

# **PROGRAMAÇÃO,**

## ALGORITMOS E ESTRUTURAS DE DADOS



**JOÃO PEDRO NETO**

# **PROGRAMAÇÃO, ALGORITMOS E ESTRUTURAS DE DADOS**

**ESCOLAR EDITORA**

*Título: Programação, Algoritmos e Estruturas de Dados*

# **Índice**

---

Prefácio .....	11
Introdução .....	15
1 – Tipos e Variáveis.....	21
1.1 Tipos Primitivos .....	22
1.2 Variáveis e Expressões.....	29
1.3 Tipos Compostos * .....	34
2 – Estrutura de Controlo .....	43
2.1 As Três Ferramentas.....	43
2.2 Instruções Java .....	44
3 – Referências: Vectores.....	61
4 – Referências: Classes.....	73
4.1 O tipo Classe .....	74
4.2 Classes do Java .....	94
5 – Recursão.....	107
5.1 Recursão vs. Iteração.....	109
5.2 Recursão Terminal .....	112
5.3 Resolução de Problemas por Retrocesso *	114
6 – Excepções e Asserções.....	121
6.1 Excepções.....	125
6.2 Asserções.....	131

7 – Entrada e Saída de Dados .....	135
7.1 As Classes Gerais de Entrada/Saída .....	136
7.2 Os Fluxos Básicos .....	138
7.3 Os Fluxos Avançados .....	143
7.4 Origem e Destino da Informação .....	148
7.5 Usando a classe Scanner .....	151
8 – Especificação .....	159
8.1 Tipo de Dados Abstracto .....	164
8.2 Desenho .....	169
9 – Abstracção .....	173
9.1 Classes Abstractas .....	174
9.2 Encapsulamento .....	175
9.3 Tipos e Interfaces .....	176
10 – Herança .....	187
10.1 Relações de Herança .....	188
10.2 Herança no Java .....	193
10.3 A Classe Object .....	203
10.4 As Excepções Revisitadas .....	203
10.5 Herança Múltipla .....	204
11 – Desenho por Contrato .....	209
11.1 Correcção .....	210
11.2 Desenho por Contratos .....	214
11.3 A Linguagem de Contratos .....	217
11.4 O Desenho Revisitado .....	223
11.5 Contratos e Herança .....	226
12 – Programação por Eventos .....	231
12.1 Eventos em Java .....	232
12.2 Máquinas de Estados .....	238
13 – Análise de Algoritmos .....	247
13.1 Introdução .....	247
13.2 Notação O .....	250
13.3 Análise de Programas .....	254
14 – Listas .....	261
14.1 Especificação da Lista .....	261
14.2 O Desenho e os Contratos do TDA Lista .....	266
14.3 Uma Implementação Dinâmica da Lista .....	274
14.4 Outros tipos de Listas .....	289

---

## ÍNDICE

15 – Pilhas e Filas .....	293
15.1 Pilha.....	294
15.2 Fila.....	301
16 – Conjuntos .....	311
16.1 A Especificação do Conjunto .....	312
16.2 A Interface Conjunto.....	314
16.3 Uma Implementação Estática.....	318
17 – Árvores.....	327
17.1 Árvores Binárias.....	328
17.2 Árvores Binárias de Pesquisa .....	345
17.3 Árvores Genéricas .....	362
17.4 Amontoados .....	366
18 – Tabelas .....	383
18.1 Especificação da Tabela .....	384
18.2 A Interface Tabela.....	385
18.3 Tabelas de Dispersão.....	387
19 – Ordenação .....	403
19.1 Algoritmos de Ordenação.....	403
19.2 A Classe Sort.....	413
20 – Programação Dinâmica .....	417
20.1 Jogo de Bachet .....	419
20.2 A maior subsequência comum .....	422
20.3 Multiplicação de Matrizes .....	427
Anexo A – Programas Java .....	435
A.1 A máquina virtual.....	435
A.2 O SDK.....	436
A.3 Pacotes.....	439
A.4 Comentários e Documentação.....	441
A.5 Arquivos JAR.....	445
Bibliografia Selecciónada .....	447
Índice Remissivo .....	449



*Em memória do Prof. Fernando Moura Pires (1948-2003)*



# Prefácio

---

Desde que nascemos regemo-nos por um conjunto de compromissos assumidos sem a nossa opinião: família, cultura, ambiente, uma miríade de outros factores. Um dos mais importantes é a língua. A língua que aprendemos (neste caso, o português) tem um aspecto formativo sobre o nosso processo cognitivo. A forma como pensamos é, até certo ponto, estruturada pela forma como aprendemos a falar e a exprimir as nossas opiniões. Quando foi tomada a decisão de escrever um manual de introdução à programação surgiu uma questão semelhante. Uma linguagem de programação é uma notação para escrever programas. A linguagem escolhida como veículo de transmissão dos conceitos fundamentais da programação forma uma percepção – ou se quisermos, um conjunto de preconceitos – sobre “o que é” e “como se faz” computação. Este é um assunto de difícil resolução. É sempre preciso uma dose mínima de convenções para aprender, só não devemos fechar-nos à possibilidade de as mudar, se for esse o caso<sup>1</sup>.

Esta escolha recaiu sobre a linguagem de programação Java que nos leva na direcção da programação centrada em objectos e da programação procedural (historicamente relacionada) deixando para trás outros modelos de programação (como por exemplo, a programação lógica ou a programação funcional). O Java é uma linguagem com um sucesso e expansão sem muitos paralelos no mundo da Informática; é uma linguagem

---

<sup>1</sup> Mesmo a própria Ciência funciona assim. Um paradigma, uma ideia estruturante, tem uma quantidade de inércia associada (representada pelos especialistas da Escola que criou esse paradigma) que só perante um conjunto de problemas fundamentais não explicáveis, se deixa substituir por outro.

ainda em evolução com funcionalidades apropriadas às necessidades modernas como a comunicação por rede, a concorrência e a robustez; é centrada em objectos facilitando o uso de determinadas disciplinas que ajudam na construção e manutenção de grandes aplicações informáticas (parte do que se denomina por **engenharia de software**). Não sendo a melhor alternativa do ponto de vista teórico (por exemplo, não esgota os conceitos da programação centrada em objectos) é uma escolha equilibrada entre a teoria e a prática. Quando for apropriado apresentaremos conceitos que, apesar de não se encontrarem na linguagem Java, são relevantes para uma melhor compreensão das possibilidades da programação.

## Para o Leitor

A estrutura do texto é a seguinte:

- A 1<sup>a</sup> parte apresenta os conceitos fundamentais (como memória, algoritmo, recursão, classes e objectos, ficheiros, exceções) através de múltiplos exemplos em código fonte Java.
- A 2<sup>a</sup> parte introduz algumas técnicas da engenharia de *software* que permitem controlar a inevitável complexidade inerente ao desenvolvimento de programas mais complexos, nomeadamente o processo de especificação através da noção de Tipos de Dados Abstractos e do desenho suportado pela Programação por Contrato. O processo de implementação baseia-se na programação centrada em objectos (através de mecanismos como a abstracção, a herança e o polimorfismo).
- A 3<sup>a</sup> parte descreve vários algoritmos comuns (por exemplo, como ordenar um conjunto de valores, quando se pode utilizar a programação dinâmica) e estruturas de dados (pilhas, filas, conjuntos, árvores, tabelas). A apresentação segue os conceitos da programação estruturada – onde se inclui a especificação e desenho discutidos na 2<sup>a</sup> parte – servindo igualmente como um conjunto de exemplos completos e documentados.
- É ainda incluído um anexo que descreve o pacote de desenvolvimento disponibilizado pela empresa *Sun* para compilar, documentar, organizar e executar programas Java.
- As principais referências bibliográficas foram reunidas numa secção final para não prejudicar a exposição e, consequentemente, o ritmo de leitura dos capítulos.

São usados alguns símbolos com os seguintes significados:

<input checked="" type="checkbox"/>	Algo correcto.
<input checked="" type="checkbox"/>	Algo incorrecto.
	O digitado no teclado.
	O impresso no monitor.
	A descrição do erro ocorrido.
	O conteúdo do ficheiro <nome>.
	Mudança de linha.
	Assunto opcional numa primeira leitura

Foram utilizadas diferentes fontes para melhor identificar o texto do uso de código fonte Java.

Em relação aos conhecimentos dos leitores assume-se um conhecimento prévio dos conceitos básicos de Álgebra e Lógica estudados no secundário. Assume-se algum conhecimento de Informática, do ponto de vista do utilizador, suficiente para descarregar, instalar e executar programas, em especial o pacote de desenvolvimento Java.

O endereço [www.di.fc.ul.pt/~jpn/java](http://www.di.fc.ul.pt/~jpn/java) contém informação complementar e actualizada.

## Para o Docente

O conteúdo deste manual foi concebido para apoiar duas disciplinas/módulos de introdução à programação. Consoante o foco seja a programação procedural ou a programação centrada em objectos, o Java é uma escolha interessante devido à disciplina de programação que impõe e aos mecanismos de segurança e robustez que possui. Esta organização ajuda o estudante, no início da aprendizagem, à compreensão de um conjunto estruturante de conceitos essenciais.

A primeira parte descreve a matéria de uma disciplina de *Introdução à Programação* (IP), onde são referidos os conceitos de tipo, estrutura de controlo, referências, classes, vectores, recursão, ficheiros, exceções, entre outros. No contexto do relatório<sup>2</sup> ACM/IEEE com recomendações sobre a estrutura de um Curso de Computação (CC2001), esta matéria cobre os tópicos seguintes:

- PF1 – Fundamental Programming Constructs
- PF2 – Algorithms and Problem Solving
- PF3 – Fundamental Data Structures
- PF4 – Recursion
- PF5 – Event-Driven Programming (*em relação às exceções*)
- PL4 – Declarations and Types

A segunda parte inclui noções relevantes da programação centrada em objectos e do desenvolvimento de *software*, cobrindo os seguintes tópicos do CC2001:

- PL5 – Abstraction Mechanisms
- PL6 – Object-Oriented Programming
- PF5 – Event-Driven Programming
- SE1 – Software Design (*parte introdutória*)
- SE5 – Software Requirements and Specifications (*parte introdutória*)

---

<sup>2</sup> Versão electrónica em <http://www.acm.org/sigcse/cc2001/>

A terceira parte cobre a disciplina de *Algoritmos e Estruturas de Dados* (AED) e trata as unidades seguintes:

- AL1 – Basic Algorithm Analysis
- AL2 – Algorithmic Strategies
- AL3 – Fundamental Computing Algorithms

O conceito de especificação e da programação por contratos é necessário para os Tipos de Dados Abstractos (TDA) utilizados na 3<sup>a</sup> parte. Num contexto de duas disciplinas (IP e AED), a programação por contratos pode ser inserida no fim de IP (depois das excepções e asserções) enquanto os TDA são apresentados no início de AED. As noções de interface e herança devem ser discutidas (mesmo que superficialmente) para as concretizações dos TDA em classes Java.

## **Agradecimentos**

O problema de agradecer a um conjunto de pessoas que o merecem é deixar inadvertidamente alguém de lado. Desde já peço desculpa às vítimas da minha memória.

Pela revisão dos capítulos o meu especial obrigado ao Francisco Martins, Ana Paula Maldonado e Isabel Nunes pelas extensas páginas revistas e pela paciência necessária. Também queria dirigir um agradecimento às restantes pessoas que leram partes e ajudaram a diminuir o número de incorrecções, nomeadamente ao António Casimiro, Ana Luísa Respicio, Beatriz Carmo, Francisco Couto, Graça Gaspar, João Sarmento, Henrique Rocha, Alexandra Mota e Andreia Veiga.

Não só da bibliografia editada se estuda e aprende. Muito contribuíram as notas, acetatos e exercícios produzidos pelo esforço (geralmente não contabilizado e muitas vezes desvalorizado na avaliação da carreira docente) de vários docentes do Departamento de Informática da FCUL, nomeadamente do José Luís Fiadeiro, Vasco Vasconcelos, Beatriz Carmo, Antónia Lopes, Isabel Nunes e Pedro Rodrigues.

Ao Fernando Moura Pires, José Félix Costa e Helder Coelho pelos conselhos, apoios e orientações dados na última década e dos quais é já impossível particularizar, o meu reconhecimento e obrigado!

# Introdução

---

À medida que os computadores abrangem cada vez mais tarefas e (normalmente) facilitam – porque automatizam – parte do tecido social em que vivemos, as metáforas computacionais tomam o devido lugar na cultura vigente. É comum dizer que uma pessoa é composta por corpo e mente. Por analogia, um computador divide-se em **suporte físico** (do inglês, *hardware*) e **suporte lógico** (do inglês, *software*). O *hardware* é o componente físico do computador. O *software* é o componente lógico, uma virtualidade sustentada por impulsos eléctricos. Da mesma forma que a mente é o principal factor da individualidade humana, o *software* determina a funcionalidade do computador. O *hardware* é “simplesmente” o suporte que permite a execução das tarefas determinadas pelo *software* que o utiliza (apesar de em certas máquinas mais específicas, as limitações do *hardware* determinarem as potencialidades do *software*).

A estrutura exacta do *hardware* depende de cada fabricante, como os detalhes mecânicos de cada carro variam com os diferentes modelos. Mas, como para os carros, a estrutura geral do *hardware* dos vários computadores é semelhante porque os objectivos são semelhantes. Basicamente, divide-se o suporte físico de um computador em:

- Processador – a **unidade central de processamento** (do inglês, *central processing unit* ou *CPU*) é responsável pela correcta execução de uma ou mais tarefas. O processador consegue executar tarefas dado que a sua arquitectura é associada a uma linguagem específica – a **linguagem máquina** – capaz de expressar essas mesmas tarefas. O processador é o principal responsável pelo desempenho do computador. Empresas como a Intel ou a AMD concorrem para encontrar novas técnicas e arquitecturas que permitam construir processadores progressivamente mais rápidos. De facto, nos últimos trinta anos, a evolução do desenho dos

processadores tem respeitado a Lei de Moore: Gordon Moore (em 1965) previu que o número de componentes constituintes dos processadores duplicaria aproximadamente a cada ano e meio! Ou seja, ocorreu um crescimento tecnológico exponencial nas últimas décadas sem indicações de desaceleração.

- Memória – a capacidade de armazenar informação por um determinado período de tempo. A memória é composta fundamentalmente por **memória primária** (do inglês, *main memory*, usualmente *Random Access Memory, RAM*) e por **memória secundária** (do inglês, *auxiliary memory*). A memória primária é muito mais rápida pois é suportada electronicamente, enquanto a memória secundária é suportada mecanicamente (através de disquetes, discos rígidos, CD-ROMs). Porém, a memória primária só contém informação enquanto o computador está activo, a informação é temporária. Apenas na memória secundária a informação tem um carácter permanente. Existe ainda a **memória fixa** (do inglês, *Read Only Memory, ROM*) inerente ao *hardware* e independente do *software* utilizado, não podendo ser alterada por este.

Existem ainda **periféricos**, todos os sistemas auxiliares que permitem a comunicação/interface entre o processador e o exterior. Por exemplo: o teclado para comunicar com o utilizador, as placas de rede para comunicação em redes locais ou na Internet, os suportes mecânicos para armazenar memória secundária, etc.

O suporte lógico pode-se classificar nas áreas seguintes:

- Sistema Operativo – Um **sistema operativo** (do inglês, *operating system*), S.O., é uma tarefa especial que quando executada por um computador permite a gestão dos recursos do mesmo, a execução de outras tarefas, a gestão da memória e dos periféricos existentes. Entre os exemplos mais comuns de S.O. encontramos o *Windows* e o *Linux*.
- Ficheiros – Um **ficheiro** é um conjunto de dados agrupados sob um determinado nome que o identifica. O sistema operativo é capaz de reconhecer um ficheiro e de manipulá-lo (cria, modifica, apaga ficheiros). Um ficheiro que guarde outros ficheiros designa-se por directória ou pasta.
- Programas – Um **programa** é um ficheiro que descreve uma tarefa específica através de uma linguagem conhecida pelo sistema (na maioria dos casos, na linguagem máquina do processador). O S.O. é capaz de reconhecer um programa e de executá-lo, ou seja, é capaz de executar a tarefa descrita no seu conteúdo.

De que forma o S.O. reconhece ficheiros e programas? Toda a informação está armazenada na memória do computador, sendo identificada e gerida pelo S.O. através de um conjunto de **endereços de memória**. Como é que a informação é armazenada? Devido à forma como o suporte físico está construído, cada componente do *software* é representado por um conjunto de desactivações/activações eléctricas a que se convencionou designar por zeros e uns. Deste modo, a representação da informação é baseada num **código binário** (dado que o alfabeto possui apenas dois símbolos: 0/1). Esta unidade atómica de representação é

designada por **bit**, i.e., um bit é igual a 0 ou igual a 1. Convencionou-se usar o termo **byte** (em português, **octeto**) para designar um conjunto de oito bits. Enquanto um bit pode representar dois valores diferentes, um byte representa  $2^8 = 256$  valores diferentes.

A memória de um computador é organizada em conjuntos de bytes. Cada endereço de memória é constituído por X bytes e referencia um byte de memória. No caso da maioria dos S.O. actuais, um endereço ocupa quatro bytes, ou seja, 32 bits. Assim, é possível organizar a memória disponível em  $2^{32}$  endereços diferentes, ou seja, quatro Gbytes (aproximadamente quatro mil milhões de bytes).

Os programas contêm a descrição de uma tarefa na linguagem máquina do processador em questão. No entanto, esta forma de expressar tarefas é pouco adequada para ser interpretada por uma pessoa, dado ser uma linguagem direcionada à arquitectura do processador. Diz-se uma **linguagem de programação de baixo nível** porque tem um baixo nível de abstracção<sup>3</sup>.

Actualmente, existem linguagens mais abstractas que permitem descrever essas tarefas de uma forma mais adequada à compreensão humana, designando-se por **linguagens de programação de alto nível**. Entre as centenas (milhares?) de exemplos, encontramos linguagens como Lisp, Fortran, C, Pascal, Ada, C++, Java, etc. No entanto, os programas descritos por estas linguagens de programação não são reconhecidos directamente pelo processador. Esta descrição textual da tarefa designa-se por **código** ou **código fonte** (do inglês, *source code*). É necessário realizar uma tradução da descrição de alto nível para a linguagem máquina do processador. Este processo é designado por **compilação** e é executado por programas especiais designados por **compiladores**. O processo de compilação tem uma desvantagem: entre cada linguagem de programação e cada processador (i.e., cada linguagem máquina) é necessário uma tradução diferente, ou seja, um compilador diferente. Para X linguagens e Y processadores, são necessário XY compiladores e um compilador é um programa muito complexo que exige recursos para ser construído.

Uma outra forma de resolver o problema é através da criação de um passo intermédio entre a linguagem de alto nível e a linguagem máquina. Este passo intermédio requer a construção de uma **máquina virtual** que reconhece uma dada linguagem e executa tarefas descritas nessa linguagem. Assim, cada nova linguagem precisa de um compilador que traduza os seus programas para a linguagem reconhecida pela máquina virtual. Como cada processador possui um programa que executa a máquina virtual, para lidar com X linguagens e Y processadores são somente necessários X compiladores diferentes e Y

---

<sup>3</sup> A abstracção é o grau no qual um sistema descreve a funcionalidade relevante a uma determinada tarefa (quanto mais abstracta for a descrição, menos detalhes são descritos). Este conceito, aparentemente hermético e por isso assustador no início, é uma das ferramentas mais poderosas que a nossa mente possui. Falaremos sobre a abstracção no decorrer do manual.

máquinas virtuais. São assim precisos menos programas (e menos recursos). A desvantagem é uma efectiva perda de desempenho, dado que os programas são executados pela máquina virtual (em si um programa) e não directamente na linguagem máquina reconhecida pelo processador. Uma vez traduzido o programa da linguagem de alto nível para a linguagem da máquina virtual, esta nova descrição pode ser executada em qualquer processador. O processo da compilação é separado do processo da execução, tornando os programas mais portáveis entre processadores diferentes. Esta foi a atitude tomada pelos arquitectos do Java.

# PARTE I

## Programação Procedimental



# 1 – Tipos e Variáveis

---

Existem dois conceitos fundamentais na computação: o algoritmo e a memória. O algoritmo é uma medida de tempo. A memória é uma medida de espaço. O **algoritmo** é um conjunto preciso de ordens para a resolução de uma dada tarefa. A **memória** é a informação lida e alterada pela execução do algoritmo. Esta informação é essencial porque contém os dados necessários para a resolução do problema incluindo os resultados temporários provenientes da execução. Por exemplo, para multiplicar 234 por 132 podemos usar um método aprendido no ensino primário:

$$\begin{array}{r} 234 \\ \times 132 \\ \hline 468 \\ 702 \\ + 234 \\ \hline 30888 \end{array}$$

Inconscientemente, ao mesmo tempo que aplicamos um algoritmo (multiplicar cada dígito da 2<sup>a</sup> parcela pela 1<sup>a</sup> parcela e para o *i*-ésimo resultado multiplicar *i*-1 vezes por 10), usámos memória temporária (no papel) para guardar essas multiplicações sucessivas. Porquê? Porque a soma dessas multiplicações devolve-nos o resultado final. A própria soma usa memória temporária (dizemos “e vão dez...”, por exemplo na passagem da 4<sup>a</sup> para a 5<sup>a</sup> coluna da soma).

As noções de algoritmo e memória acompanham a Humanidade desde há muito. A existência do Ábaco é milenar (existem vestígios de ábacos na Babilónia que remontam a

300 A.C), bem como métodos para os mais diversos cálculos: a soma, a subtracção, a multiplicação, a divisão...

## 1.1 Tipos Primitivos

A informação que recebemos só faz sentido se soubermos o contexto de onde provém. Por exemplo, o número 24.1 pode indicar a altura de um edifício em metros ou a velocidade de um corredor em quilómetros por hora. Do ponto de vista do computador (uma máquina de processamento baseada no sistema binário) a informação é sempre composta por uma sequência finita de bits. Mas que tipo de bits são, ou seja, que tipo de informação é representada?

As linguagens de programação permitem utilizar as funcionalidades do computador a partir de um nível de abstracção maior que o da linguagem máquina do processador. Esta abstracção é útil porque esconde um conjunto de detalhes de implementação da máquina, na maioria dos casos, desnecessários para a resolução da tarefa que pretendemos solucionar. Uma das abstracções que as linguagens de programação fornecem é a possibilidade de representar e manipular tipos de informação, como por exemplo, inteiros, valores lógicos, reais, frases. Esta abordagem facilita a construção dos algoritmos porque para cada tipo de dados existe um conjunto de operações para manipular esses dados e é mais fácil detectar os problemas que ocorrem quando os dados não são do tipo esperado.

Consideramos um **valor** como uma unidade informativa armazenada em memória e manipulável quando necessário. Um **valor atómico** é um valor que não pode ser decomposto em unidades mais simples (ao contrário de um **valor composto**). Exemplos de valores atómicos: o inteiro zero, o real 0.5, a letra 'a'... Exemplos de valores compostos: as coordenadas  $\langle x,y \rangle$  de um ponto a duas dimensões, um registo de identificação de uma pessoa, uma conta bancária...

É conveniente agrupar conjuntos de valores que consideramos relacionados (os números naturais 1, 2, 3, 4, 5... têm algo em comum. O quê?). Essa relação é constituída por um conjunto de operações apropriadas sobre esses valores (por exemplo, a soma e a subtracção para os números naturais). Usamos esta ideia para definir tipo:

*Um tipo é um conjunto de valores relacionados por um conjunto de operações.*

Quando um tipo contém somente valores atómicos denomina-se por **tipo primitivo** ou **atómico**.

O Java suporta oito tipos primitivos apresentados na tabela seguinte:

**Tabela 1.1 – Tipos Primitivos em Java**

Tipo	Valor Inicial	Bits Usados	Valor Mínimo	Valor Máximo
boolean	false	1	–	–
char	\u0000	16	\u0000	\uFFFF
int	0	32	-2147483648	2147483647
byte	0	8	-128	127
short	0	16	-32768	32767
long	0	64	-9223372036854775808	9223372036854775807
float	0.0	32	$\pm 1.45E-45$ até $\pm 3.4028235E+38$	
double	0.0	64	$\pm 4.9E-324$ até $\pm 1.7976931348623157E+308$	

Cada tipo primitivo possui um conjunto de **operações**, ou **operadores** (por exemplo, a soma é uma operação sobre valores inteiros). Uma **expressão** é um valor ou uma combinação de valores e operações sobre valores (por exemplo,  $4+5$  é uma expressão). Veremos de seguida as operações disponíveis a cada tipo.

## Tipo Boolean

O tipo `boolean` (em português, booleano) inclui os dois valores lógicos: verdadeiro (`true`) e falso (`false`). As operações do tipo booleano são as seguintes:

- **Negação** – operador `!` (em inglês, *not*). A operação de negação que troca o valor da expressão associada.
- **Conjunção** – operadores `&` e `&&` (em inglês, *and*). A operação de conjunção. As expressões  $A \& B$  e  $A \& B$  são verdadeiras se e só se  $A$  e  $B$  forem ambas verdadeiras, caso contrário as expressões são falsas. Enquanto o operador `&` avalia os dois lados da expressão, o operador `&&` somente avalia o lado direito se o esquerdo for verdadeiro.
- **Disjunção** – operadores `|` e `||` (em inglês, *or*). A operação de disjunção. As expressões  $A | B$  e  $A | B$  são verdadeiras se e só se pelo menos  $A$  ou  $B$  for verdadeira, caso contrário as expressões são falsas. Enquanto o operador `|` avalia os dois lados da expressão, o operador `||` somente avalia o lado direito se o esquerdo for falso.
- **Ou Exclusivo** – operador `^` (em inglês, *xor*). A operação “ou exclusivo”. Uma expressão  $A ^ B$  é verdadeira se só um entre  $A$  e  $B$  for verdadeiro, caso contrário a expressão é falsa. Não existem dois operadores pois é sempre necessário avaliar os dois lados da expressão.

As respectivas tabelas de verdade são:

**Tabela 1.2 – Tabelas de Verdade**

A	true		false	
B	true	false	true	false
<b>!A</b>	false		true	
<b>A&amp;B</b>	true	false	false	false
<b>A&amp;&amp;B</b>	true	false	false	false
<b>A B</b>	true	true	true	false
<b>A  B</b>	true	true	true	false
<b>A^B</b>	false	true	true	false

## **Tipo Char**

O tipo alfanumérico `char` inclui os valores constituídos por um único caractér. A lista de caracteres possível corresponde à lista de caracteres Unicode. O Unicode<sup>4</sup> é um protocolo internacional com a finalidade de representar os caractéres de (quase) todos os alfabetos do Mundo. Esta lista é definida por uma tabela de códigos que associa cada símbolo a um determinado número. Como cada caractér é representado por 16 bits é possível representar  $2^{16}$ , ou seja, 65536 símbolos diferentes.

Em Java, cada caractér Unicode é representado por um número com quatro dígitos hexadecimais (por exemplo, o símbolo  $\pi$  possui o código `03C0` que em Java se representa por `\u03C0`). No caso de pretender-se usar letras, dígitos ou símbolos da tabela ASCII basta incluir entre aspas simples (por exemplo `'a'`, `'+'`, `'4'`). Outros símbolos especiais são precedidos pelo símbolo `\`.

**Tabela 1.3 – Alguns Caractéres Especiais**

Símbolo	Significado
<code>\b</code>	<i>Apagar último caractér</i>
<code>\t</code>	<i>Tabulação horizontal</i>
<code>\n</code>	<i>Mudança de linha</i>
<code>\"</code>	<i>Aspa dupla</i>
<code>\'</code>	<i>Aspa simples</i>
<code>\\"</code>	<i>Símbolo \</i>

## **Tipos Inteiros: Byte, Short, Int e Long**

Estes tipos incluem valores numéricos inteiros. Entre os quatro tipos a única diferença é o número de bits necessários às suas representações. Esse número determina os números máximos e mínimos que cada tipo pode conter (ver tabela 1.1). O programador deve usar o

---

<sup>4</sup> As tabelas Unicode estão disponíveis em [www.unicode.org/charts/](http://www.unicode.org/charts/)

tipo mais económico de modo a não desperdiçar memória. Por exemplo, se é necessário armazenar a idade de uma pessoa, o tipo `short` é suficiente (ninguém vive mais que 32767 anos). Comparado com o uso do tipo `int` poupar-se-ia  $32-16=16$  bits.

Para representar um número em Java escreve-se a sequência de dígitos que o define podendo esta ser precedida pelo símbolo `-` para números negativos, ou (opcionalmente) pelo símbolo `+` para números positivos. Alguns exemplos de números inteiros: `0`, `1`, `123456`, `-1000`, `+100`.

É possível representar valores inteiros de base hexadecimal (base 16) e de base octal (base 8). Para números hexadecimais preceder o número com `0x` (zero xis) sendo os seis últimos dígitos representados pelas letras A a F. Para números octais preceder o número com um zero e utiliza-se apenas usa os dígitos de 0 a 7. Alguns exemplos:

`0xFF`      número 255 ( $15 \times 16 + 15$ ) em base hexadecimal

`0764`      número 500 ( $7 \times 8^2 + 6 \times 8 + 4$ ) em base octal

Por omissão, um número é interpretado como um inteiro de 32 bits. Se colocarmos um `L` no fim do número é interpretado como um inteiro de 64 bits, ou seja, um `long`.

`100L`      um número de tipo `long`

`0xA9L`      outro número de tipo `long`

Os operadores dos tipos inteiros são os seguintes:

---

**Tabela 1.4 – Operadores Inteiros**

---

Operação	Significado
<code>+</code>	<i>Adição</i>
<code>-</code>	<i>Subtração</i>
<code>*</code>	<i>Multiplicação</i>
<code>/</code>	<i>Quociente da Divisão</i>
<code>%</code>	<i>Resto da Divisão</i>
<code>==, !=</code>	<i>Igual e Diferente</i>
<code>&lt;, &lt;=</code>	<i>Menor e Menor ou Igual</i>
<code>&gt;, &gt;=</code>	<i>Maior e Maior ou Igual</i>
<code>&lt;&lt;</code>	<i>Deslocamento para a esquerda</i>
<code>&gt;&gt;, &gt;&gt;&gt;</code>	<i>Deslocamento para a direita</i>
<code>&amp;</code>	<i>AND bit a bit</i>
<code> </code>	<i>OR bit a bit</i>
<code>^</code>	<i>XOR bit a bit</i>
<code>~</code>	<i>Complemento bit a bit</i>

Como a divisão de dois inteiros não tem necessariamente de resultar num inteiro, o Java possui dois operadores que permitem retirar a informação do quociente e do resto dessa divisão, nomeadamente os operadores `/` e `%`.

$1+6$	igual a 7
$234 * 132$	igual a 30888
$1 + -5$	igual a -4
$10 / 2$	igual a 5
$10 \% 2$	igual a 0
$1 / 2$	igual a 0 (um a dividir por dois tem quociente 0)
$1 \% 2$	igual a 1 (um a dividir por dois tem resto 1)

Os operadores `>>`, `>>>` e `<<` manipulam a sequência binária que representa o número: empurram n bits numa determinada direcção. Por exemplo, o byte 57 é representado pelos 8 bits seguintes:

0	0	1	1	1	0	0	1
---	---	---	---	---	---	---	---

Ao executar  $57 << 1$  obtemos a sequência:

0	1	1	1	1	0	0	1	0
---	---	---	---	---	---	---	---	---

Ou seja, 114. Cada deslocamento de um bit para esquerda resulta numa multiplicação por 2 (porquê?). Os novos bits da direita são preenchidos por zeros. Porém existe um detalhe na representação dos números negativos: o bit mais à esquerda guarda o sinal (0 para números positivos, 1 para números negativos). Para os números negativos inverte-se o significado dos outros bits. Esta representação é designada complemento 2 e tem como objectivo facilitar as operações aritméticas binárias. Por exemplo, o byte  $-1$  é representado por:

1	1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---	---

Quando se executa  $-1 << 1$ ...

1	1	1	1	1	1	1	1	0
---	---	---	---	---	---	---	---	---

...obtém-se -2.

No caso do deslocamento para a direita, o comportamento é similar para os positivos. Já para os negativos pode-se optar por empurrar o dígito do sinal (operador `>>>`) ou não (operador `>>`).

$-4 >> 2$	igual a $-1$
$-4 >>> 2$	igual a 1073741823

Os restantes operadores servem igualmente para manipulação de inteiros bit a bit. Como existem quatro operadores lógicos para os booleanos, existem quatro operadores que tratam cada par de bits como se fossem uma expressão booleana (o bit 1 sendo igual a `true` e o bit 0 igual a `false`). Alguns exemplos:

a	1	1	0	0	1	1	0	0
b	1	0	1	0	1	0	1	0

a&b	1	0	0	0	1	0	0	0
a b	1	1	1	0	1	1	1	0
a^b	0	1	1	0	0	1	1	0
~a	0	0	1	1	0	0	1	1

Para cada tipo inteiro é implementada uma aritmética módulo  $2^b$  (onde  $b$  é o número de bits do tipo em questão). Se, ao executar um comando de incremento, for ultrapassado o valor máximo (em inglês, *overflow*) o valor “dá a volta” ao intervalo e passa a ser igual ao valor mínimo. Por exemplo, se um valor `byte` igual a 127 for incrementado numa unidade passa a ser -128. O inverso também ocorre: se estiver no valor mínimo e for decrementada uma unidade (em inglês, *underflow*) passa para o valor máximo.

Caso ocorra uma divisão por zero (exemplo, `1/0`) é levantado um erro (estes erros serão referidos no capítulo 6).

## Tipos Reais: `Float` e `Double`

Os tipos primitivos `float` e `double` incluem números reais com precisão finita (finita porque a memória de qualquer computador nunca é infinita). A diferença entre estes dois tipos reside nos bits necessários das respectivas representações: 32 bits para o tipo `float`, 64 bits para o tipo `double`. Ambos os formatos respeitam a norma padrão IEEE 754-1985 para números de vírgula flutuante. Esta norma define como se representam os números, como se realizam as operações e como lidar com arredondamentos, infinitos e outras exceções.

Um real é uma sequência de dígitos onde existe obrigatoriamente um ponto decimal (pode-se usar a notação exponencial). Alguns exemplos:

0.0	igual a 0,0
3.001	igual a 3,001
3.1415926	uma aproximação de $\pi$
-4.1234	igual a -4,1234
1.2e6	igual a $1,2 \times 10^6 = 1200000$
-41.2e-2	igual a $-41,2 \times 10^{-2} = 0,412$

Por omissão um número real é do tipo `double`. Para dizer explicitamente que o número é do tipo `float` deve-se colocar a letra F a seguir ao número.

3.0F	igual ao <code>float</code> 3,0
------	---------------------------------

Como se usa um número limitado de bits para representar números reais, nunca se deve esquecer que os números armazenados em `float` e `double` são apenas aproximações. Por

exemplo, o resultado de  $1.0/3.0$  é  $0.3333333333333333$  e não  $0.(3)$ . Por este motivo deve-se evitar o operador de igualdade  $==$  para comparar dois números reais. É melhor definir que dois números são iguais se o módulo da diferença entre ambos é menor que um dado valor  $\Delta$ :

*igual(x, y) se e só se  $|x - y| < \Delta$*

Os operadores dos tipos reais são os mesmos que dos inteiros exceptuando  $\%$ ,  $>>$ ,  $>>>$ ,  $<<$  e o operador  $/$  que devolve o valor aproximado da divisão.

No caso de uma operação ilegal ambos os tipos possuem valores especiais que permitem identificar o problema que ocorreu.

```
1.0/0.0    igual ao valor Infinity  
-1.0/0.0   igual ao valor -Infinity  
0.0/0.0    igual ao valor NaN (Not a Number, i.e., não é um número)
```

## Tipos Enumerados e Subtipos

Em certas linguagens é possível definir um conjunto finito de valores para criar um novo tipo primitivo. A este tipo de dados denomina-se por **tipos enumerados**. Segue-se um exemplo na linguagem de programação PASCAL onde se define um novo tipo **Dias** constituído por sete novos valores atómicos (cada um é descrito por um literal de três letras que identifica um dia da semana):

```
type                      {código Pascal}  
  Dias = (Dom, Seg, Ter, Qua, Qui, Sex, Sab);  
var  
  d : Dias;  
begin  
  d := Seg;  
end.
```

Por vezes, existe uma relação de ordem implícita entre esses elementos, como ocorre no PASCAL onde se determina que  $\text{Dom} < \text{Seg} < \text{Ter} < \dots < \text{Sab}$ . Assim, **Dias** é um **tipo enumerado ordenado**.

Os tipos enumerados foram incluídos na versão Java 5. O exemplo PASCAL seria traduzido nas seguintes instruções:

```
enum Dias {Dom, Seg, Ter, Qua, Qui, Sex, Sab};  
...  
Dias d = Dias.Seg;
```

Um **subtipo** é a definição de um novo tipo contido num tipo já existente. Um exemplo novamente em PASCAL define um tipo **Nota** a partir de um conjunto limitado de valores do tipo de dados inteiro.

```
type                                {código Pascal}
  Nota = 1..20;
var
  n, m : Nota;
begin
  n := 15;
  m := n+1;
end.
```

Um subtipo utiliza as funcionalidades do tipo que o define. Neste sentido, existe uma noção muito rudimentar de herança associada ao subtipo (o subtipo herda as operações do tipo). Falaremos de herança no capítulo 10.

## 1.2 Variáveis e Expressões

Aprendemos que se pode representar informação escrevendo-a explicitamente (por exemplo, `-0.98`, `'a'`, `12`) – estas representações designam-se por **literais**. Um literal corresponde a uma informação bem definida de um dado tipo e é imutável.

Uma **variável** é o nome que designa uma unidade de memória de onde se pode aceder a uma informação de um dado tipo. Uma variável pode conter diferentes valores durante a sua existência.

Cada variável, em Java, é identificada por uma palavra de uma ou mais letras, números e símbolos `_`, não podendo começar por um dígito. As palavras seguintes são nomes válidos para variáveis: `a`, `i10`, `manual`, `proximo_inteiro`; e os seguintes são inválidos: `1a`, `a+a`, `proximo-inteiro`. No Java, as minúsculas e maiúsculas são consideradas letras diferentes, logo os nomes `aa`, `aA`, `Aa` e `AA` são todos diferentes.

Para evitar problemas com nomes de tipos e variáveis usaremos a notação que obriga que cada tipo não primitivo comece sempre com uma maiúscula e cada variável comece com uma letra minúscula. Se os nomes dos tipos e variáveis tiverem uma ou mais palavras, as primeiras letras das palavras subsequentes começam por uma maiúscula. Exemplos desta notação: `T`, `NovoTipo`, `OutroTipo`, `i`, `umaVariavel`, `umaOutraVariavel`.

Uma variável Java tem de ser declarada, ou seja (i) relacionar a futura variável com um tipo de dados conhecido; (ii) reservar memória para armazenar a informação e (iii) associar um nome a esse novo espaço. Tudo se resume a algo como:

```
int i;
```

A palavra `i` fica associada a um endereço de memória que armazena um inteiro – diz-se que `i` é uma variável inteira. Qual endereço? Não interessa realmente, essa gestão é uma

tarefa do computador. O ponto e vírgula final é um elemento sintáctico que indica o fim de um comando.

Para colocar um valor de um literal no endereço associado a uma variável é usado o operador de atribuição =. No próximo exemplo, o literal inteiro 10 é armazenado na variável i.

i = 10;

Isto constitui uma instrução (uma ordem) de atribuição. Falaremos mais sobre instruções no próximo capítulo.

O valor de uma variável é o conteúdo armazenado na memória associada a essa variável. Pode-se combinar os operadores da secção anterior com literais e variáveis para obter expressões mais complexas. Por exemplo, a instrução seguinte armazena na variável i o seu valor mais um.

i = i + 1;

Quando se observa a linha anterior pela primeira vez, há quem a interprete como uma equação impossível (qual é o número igual a si próprio mais um?). Mas a linha anterior não é uma equação, é uma ordem que determina a avaliação da expressão  $i+1$  para depois a armazenar na variável i.

Se uma variável surge à direita do operador de atribuição ocorre uma leitura do valor da variável. Quando a variável encontra-se à esquerda do operador de atribuição é usada para armazenar o valor da expressão. Em resumo: à direita a variável é lida, à esquerda a variável é escrita.

Na instrução que se segue, o resultado é armazenado noutra variável mas o mecanismo de avaliação e atribuição é o mesmo. Avalia-se a expressão  $i+1$  e guarda-se esse valor na posição de memória associada à variável j.

j = i + 1;

Esta atribuição é igualmente uma expressão cujo valor é dado pela expressão da direita. Isto significa que se pode atribuir o valor da mesma expressão a diversas variáveis numa única linha:

i = j = 1;  
a = b = 1.0 + c;

É possível construir expressões arbitrariamente complexas (com ou sem variáveis):

5+6-60  
10\*(5-6%(4+j/(4-i)))  
40.0/y \* (23.3+(6.1-z\*y))

Nas instruções seguintes, juntou-se a declaração de uma variável com uma atribuição inicial, i.e., inicializou-se as variáveis (o último caso declara e inicializa duas variáveis).

```
double a = 1.0;
boolean eVazio = true;
int i = 0, j = 1;
```

É possível combinar o operador atribuição com outras operações. Exemplos:

x += 1;	igual a x = x + 1
x *= y;	igual a x = x * y
x %= z+1;	igual a x = x % (z+1)

Há um operador ternário no Java, o operador condicional `? :`. A expressão `a?b:c` tem a interpretação seguinte: se a expressão booleana `a` for verdadeira, a expressão é igual a `b`, senão é igual a `c`. Por exemplo:

```
i = 100;
x = i<20? 1.0 : -1.0;      é atribuído -1.0 a x, porque i<20 é falso
```

Existem ainda dois operadores incluídos nos tipos inteiros que incrementam e decrementam uma variável por uma unidade: `++` e `--`. O uso destes dois operadores depende da posição onde são colocados. Se estiverem à esquerda da variável, o incremento/decremento é realizado antes do acesso ao valor da variável, ou seja, um pré-incremento/pré-decremento. Se estiverem à direita, é realizado depois do acesso, i.e., um pós-incremento/pós-decremento. No exemplo seguinte, as cinco instruções foram executadas em sequência:

v = 1;	é atribuído 1 a v
w = ++v;	é atribuído 2 a w e 2 a v
x = --v;	é atribuído 1 a x e 1 a v
y = v++;	é atribuído 1 a y e 2 a v
z = v--;	é atribuído 2 a z e 1 a v

Ao contrário da maioria dos operadores, estes produzem **efeitos secundários**: a avaliação da expressão altera o conteúdo da própria variável. O operador de atribuição, `=` e os operadores relacionados, `+=`, `-=` ... produzem efeitos secundários semelhantes.

## Conversões entre Tipos

Por vezes queremos colocar um dado tipo de informação numa variável de um tipo diferente. Em certos casos é possível utilizar um mecanismo denominado de **conversão** (do inglês, *casting*) ou **coercão** (do inglês, *coercion*).

```
double x = 2.0;
int i = (int)x;
```

Neste exemplo, o valor da variável real `x` é armazenado numa variável inteira `i`, mas para isto ser possível fez-se a conversão do tipo `double` para o tipo `int`. Nem todas as conversões são possíveis: o tipo booleano não se pode converter. Os tipos numéricos (inteiros e reais) podem ser convertidos uns nos outros mas perde-se informação quando se passa de um valor de maior precisão para outro de menor precisão. Alguns exemplos:

<code>float a = (float)1;</code>	é atribuído 1.0 a a
<code>int b = (int)1.7;</code>	é atribuído 1 a b
<code>byte c = (byte)1000;</code>	é atribuído -24 a c
<code>byte d = (byte)(int)25.2;</code>	é atribuído 25 a d

## Precedência e Associatividade

Se nada mais for dito, a expressão seguinte encerra uma ambiguidade:

$$1 + 2 * 3$$

Qual a operação que se aplica primeiro? Se for a soma, obtemos 9 como valor final. Se for o produto, obtemos 7. Este tipo de ambiguidade resolve-se definindo **precedências** sobre os operadores. Esta noção já é usada na matemática quando se define que produtos e divisões são realizadas antes de adições e subtrações. A tabela seguinte mostra a ordem de precedência dos operadores Java (alguns destes operadores só serão explicados nos próximos capítulos).

Tabela 1.5 – Precedências (do maior para o menor)

Operador	Tipo	Assoc.	Descrição
<code>++ --</code> <code>.</code> <code>[ ]</code> <code>( )</code>	variável objecto vector método	E	pós-incremento / pós-decremento acesso a um elemento de um objecto acesso a um elemento de um vector invocação de um método
<code>++ --</code> <code>+ -</code> <code>~</code> <code>!</code>	variável número inteiro booleano	D	pré-incremento / pré-decremento operadores unários + e - complemento bit a bit negação
<code>new (tipo)</code>	referência qualquer tipo	D	criação de objectos ou vectores conversão de tipos
<code>* / %</code>	número	E	operações aritméticas
<code>+ -</code> <code>+</code>	número string	E	operações aritméticas concatenação de strings
<code>&gt;&gt; &gt;&gt;&gt;</code> <code>&lt;&lt;</code>	inteiro	E	deslocamentos
<code>&lt; &lt;= &gt; &gt;=</code> <code>instanceof</code>	número referência	E	operadores relacionais comparação de tipos
<code>== !=</code> <code>== !=</code>	tipo primitivo referência	E	igualdade entre valores igualdade entre referências
<code>&amp;</code>	inteiro	E	conjunção bit a bit

<code>&amp;</code>	booleano		conjunção lógica
<code>^</code>	inteiro		ou exclusivo bit a bit
<code>^</code>	booleano	E	ou exclusivo lógico
<code> </code>	inteiro		disjunção bit a bit
<code> </code>	booleano	E	disjunção lógica
<code>&amp;&amp;</code>	booleano	E	conjunção condicional
<code>  </code>	booleano	E	disjunção condicional
<code>? :</code>	booleano, qualquer tipo	D	operador condicional
<code>=</code>			atribuição
<code>*=</code> <code>/=</code> <code>%=</code> <code>+ =</code> <code>--</code> <code>&lt;&lt;=</code> <code>&gt;&gt;=</code> <code>&gt;&gt;&gt;=</code> <code>&amp;=</code> <code>  =</code> <code>^=</code>	variável, qualquer tipo	D	atribuição com operador

Assim, a expressão  $1+2*3$  deixa de ser ambígua: o valor da expressão é igual a 7. A precedência padrão pode ser alterada com o uso de parênteses: a expressão  $(1+2)*3$  é igual a 9.

Outra questão importante é saber como se interpretam operadores com a mesma precedência dentro da mesma expressão: é necessário definir a **associatividade** dos operadores. Na maioria dos casos, essa associatividade é à esquerda (i.e., da esquerda para a direita). Por exemplo,  $4*5/2$  é igual a 10, não a 8. Outros, como o operador de atribuição, têm associatividade à direita (i.e., da direita para a esquerda). Por exemplo,  $a=b=1$  armazena o valor 1 nas duas variáveis. A associatividade está descrita na tabela anterior: E indica associatividade à esquerda e D associatividade à direita.

Verificámos que os tipos servem para organizar conceitos (nomeadamente conjuntos de valores semelhantes e operações sobre esses mesmos valores) aos quais é associado um nome (o nome do tipo). Esta nomeação facilita a interpretação do programa e a gestão da memória necessária ao processamento desses valores. Ao convencionar limites aos valores e às operações contidas no tipo é possível atribuir uma dimensão máxima em memória para armazenar um valor desse tipo – por exemplo, um inteiro Java necessita de 32 bits.

Associando o nome do tipo à variável é mais fácil detectar erros no programa. Quais os erros do exemplo seguinte?

```
boolean b;
double d;

☒ b = true || 1;
☒ d = b;
```

Foram declarados *b* como variável de tipo booleano e *d* como real. Primeiro erro: a operação OU lógico não é usada entre valores lógicos e inteiros. Segundo erro: uma

variável real não pode receber valores lógicos. O acto de associar um nome a um tipo permite detectar problemas numa fase muito inicial da implementação onde o custo de correção do problema é muito pequeno (estas questões são discutidas no capítulo 6).

## Referências

Para além dos tipos primitivos descritos existe um conjunto especial de tipos denominados por referência. Enquanto uma variável de um tipo primitivo armazena um valor, uma variável de um tipo por referência armazena *o endereço de memória no qual se encontra um objecto*. Qual endereço de memória? Não sabemos. É uma tarefa do sistema que executa o programa. O que interessa é que a partir de uma referência é possível aceder à informação independentemente da sua posição de memória.

Existem dois tipos de referência: a classe e o vector (do inglês, *array*). Com o tipo classe é possível criar novos tipos de dados para além dos oito referidos (ver capítulo 4). O conceito de classe é tão importante que cada programa Java é obrigatoriamente um conjunto de classes e a linguagem foi estruturada à volta deste conceito. O tipo vector permite criar conjuntos de elementos do mesmo tipo agrupados e acedidos através de um índice inteiro (ver capítulo 3).

Por exemplo, é possível criar uma nova classe `Ponto2D` capaz de armazenar pontos num plano XY e de manipular esses pontos através de um conjunto de operações. Pode-se criar um vector de booleans escrevendo-se `boolean[]`. É ainda possível combinar os dois conceitos para criar um vector de pontos, `Ponto2D[]`. O número de combinações é ilimitado. Ao conjunto de todas essas possibilidades designamos **tipos de referência**.

Uma diferença importante em relação aos tipos primitivos é que estes apenas armazenam um determinado valor enquanto os tipos por referência podem armazenar múltiplos valores do mesmo tipo (num vector) ou de tipos diferentes (numa classe). As referências ocupam 32 bits sendo descritas nos capítulos 3 e 4.

## 1.3 Tipos Compostos\*

Os tipos primitivos são conjuntos de valores atómicos (i.e., valores que não podem ser decompostos em valores mais simples) associados a um conjunto de operações. Um **tipo composto** representa um conjunto de valores construídos a partir de valores atómicos (e eventualmente de outros tipos compostos) com um conjunto de operações para manipulação desses valores. Veremos nesta secção algumas formas utilizadas por outras linguagens de programação para criar tipos compostos sem recorrer à noção de classe.

### Tuplos

No caso dos tuplos, o tipo composto é o resultado do produto cartesiano de zero ou mais tipos (primitivos ou compostos). Dados os tipos U e V, o tipo construído pelo produto

cartesiano é  $T = U \times V = \{ (u,v) : u \in U \wedge v \in V \}$ . O novo tipo  $T$  é constituído por dois valores: um componente de tipo  $U$  e um componente de tipo  $V$ .

Esta forma de tipo composto existe em várias linguagens de programação como no PASCAL (designado `record`) ou no C (designado `struct`) onde cada componente incluído no tuplo é identificado por um nome. Um tuplo na linguagem C é definido da forma seguinte:

```
#include <stdio.h>      /* código C */
typedef struct {
    int x;
    int y;
} Point2D;                define o tipo Point2D como um tuplo
void main() {
    Point2D p = { 2, 1 };  declara a variável p do tipo Point2D e
                           inicializa os componentes com x=2 e y=1
    printf("%d", p.x);    imprime o valor do componente x de p
}
□ 2
```

Quando consideramos o produto cartesiano de zero componentes obtemos um tipo especial (por vezes designado `Unit`) com um só elemento (o tuplo vazio designado por `()`) e nenhuma operação. Nas linguagens historicamente relacionadas com o C (como o C++ e o próprio Java) este tipo denomina-se `void`.

## Uniões Disjuntas

Nas uniões disjuntas um elemento do tipo composto admite um e um só valor pertencente a um dado conjunto de tipos. Em vez de uma combinação de valores é escolhido um valor de um desses tipos. Dados os tipos  $U$  e  $V$ , o novo tipo construído pela união disjunta é  $T = U \cup V = \{ u : u \in U \} \cup \{ v : v \in V \}$ . Este tipo composto existe, por exemplo, no PASCAL (designado `record variante`) ou no C (designado `union`). Um exemplo em C:

```
#include <stdio.h>      /* código C */
union Valor {           os componentes partilham a mesma memória
    int ival;            ... só pode ser usado um de cada vez
    char cval;
};
void main() {
    Valor v;
    v.cval = 'a';        armazena como sendo um caracter
    printf("%c ", v.cval);  imprime como se fosse um caracter
}
```

```
v.ival = 70;           armazena como sendo um inteiro
printf("%c ",v.cval); ...a letra F tem valor 70
printf("%d ",v.ival); imprime como se fosse um inteiro
}
□ a F 70
```

## Conjuntos

Outra possibilidade de tipo composto define um tipo como o conjunto de todos os subconjuntos de um outro tipo. Dado o tipo U, o novo tipo T é igual ao conjunto potência (do inglês, *power set*) de U, i.e.,  $T = 2^U = \{ V : V \subseteq U \}$ . Este tipo composto existe, por exemplo, no PASCAL (*set of*) com operadores de união, intersecção, união, pertença e contém. Um exemplo em PASCAL a partir de um tipo enumerado:

```
type                                { código Pascal}
  Dias = (Dom, Seg, Ter, Qua, Qui, Sex, Sab);
  Trabalho = set of Dias;      o novo tipo é um conjunto de dias

var
  D, Dia : Dias;
  T       : Trabalho;

begin
  D := Seg;
  T := [D] + [Ter] + [Qua];    união de conjuntos
  T := T - [Ter];            subtração de conjuntos
  for Dia := Dom to Sab do
    if Dia in T then         operação de pertença
      write(output, 's')
    else
      write(output, 'n');
  end.
  □ nsnsnnn
```

## Tipos Recursivos

Um **tipo recursivo** T é um tipo composto onde parte da sua composição é constituída por valores do tipo T. Um exemplo é o tipo lista: uma lista ou é vazia ou é constituída por um valor e por outra lista. Esta noção recursiva permite definir estruturas sem dimensão máxima.

Um exemplo da definição de uma lista de inteiros na linguagem funcional Haskell:

```
data IList = Vazia |          -- código Haskell
            Cons Int IList
```

Duas descrições de listas:

Cons 3 (Cons 4 Vazia)	-- uma lista com 3 e 4
Vazia	-- uma lista vazia

Como esta definição, os valores possíveis para listas são:

Vazia	
Cons i <sub>1</sub> Vazia	
Cons i <sub>2</sub> (Cons i <sub>1</sub> Vazia)	
Cons i <sub>3</sub> (Cons i <sub>2</sub> (Cons i <sub>1</sub> Vazia))	
Cons i <sub>4</sub> (Cons i <sub>3</sub> (Cons i <sub>2</sub> (Cons i <sub>1</sub> Vazia))))	
...	
$\forall n \in \text{int}$	

De facto, as listas possíveis são infinitas. Entre esses valores existem listas finitas (como as listas apresentadas) e mesmo listas infinitas. Do ponto de vista computacional, apenas interessam as listas com uma dimensão máxima finita<sup>5</sup>. O conceito de tipo recursivo será usado na 3<sup>a</sup> parte, na definição de tipos de dados como listas ou árvores.

## Tipos Parametrizados

Quando definimos um tipo composto determinámos exactamente qual o tipo de informação armazenada (no exemplo da secção anterior definimos uma lista de inteiros). Em certas linguagens é possível deixar o tipo da informação indefinido no momento da definição do tipo porque as funcionalidades desse tipo não dependem da informação armazenada.

No exemplo da lista existem operações como inserir um elemento, devolver o número de elementos, inverter a lista, etc. Em nenhuma destas operações é preciso saber se a lista armazena inteiros, booleanos ou qualquer outro tipo de informação. No momento da definição do tipo apenas queremos descrever a estrutura e as respectivas operações. No momento da declaração de uma variável desse tipo é que nos comprometemos em armazenar uma determinada informação. Com esta atitude escusamos de definir e declarar uma lista de inteiros, definir e declarar uma lista de booleanos, definir e declarar uma lista de reais, etc. Apenas definimos uma lista parametrizada (i.e., o tipo de informação armazenada é um parâmetro da definição) e declaramos posteriormente as listas necessárias.

---

<sup>5</sup> Para a definição recursiva de um tipo existem infinitas soluções que a satisfazem. Como escolher entre todas essas soluções? Qualquer definição recursiva possui uma solução mínima que é a intersecção de todas as soluções possíveis. É essa solução mínima que se escolhe como significado da descrição recursiva. Este assunto está fora do âmbito do texto.

Generalizando o exemplo anterior ( $\alpha$  é o parâmetro do tipo List):

```
data List  $\alpha$  = Vazio |  
          Const  $\alpha$  (List  $\alpha$ )
```

Definiu-se que List é uma lista de um **tipo genérico**  $\alpha$ . Agora List pode ser usado em vários contextos:

Const True Vazio	-- uma lista de booleanos
Const 3 Vazio	-- uma lista de inteiros
Const 1.0 Vazio	-- uma lista de reais
Vazio	-- uma lista vazia de tipo $\alpha$

É possível criar funções que manipulem a lista de forma genérica. No exemplo seguinte, a função tamanho calcula recursivamente o número de elementos da lista independentemente do tipo de informação armazenado. Se a lista for vazia o resultado é zero, senão o resultado é um (o primeiro elemento da lista) mais o tamanho do resto da lista:

```
tamanho Vazio = 0  
tamanho (Const primeiro resto) = 1 + tamanho resto
```

Desde o Java 5 existe um mecanismo semelhante para criar **classes genéricas** (do inglês, *generics*) às quais se pode, posteriormente, adicionar um tipo definido. No exemplo seguinte (à exceção dos genéricos, os detalhes sintáticos e semânticos do exemplo são explicados nos próximos capítulos), o tipo genérico diz respeito a uma informação consultada através da operação consultar():

```
class Padrao<T> {  
    private T info;  
    public Padrao(T x) { info = x; }  
    public T consultar() { return info; }  
}  
...  
Padrao a = new Padrao<Integer>(127);  
Padrao b = new Padrao<Double>(4.5);  
System.out.print(a.consultar()+"|"+b.consultar());  
□ 127|4.5
```

Os tipos genéricos permitem criar uma implementação genérica sobre os tipos utilizados e ao mesmo tempo específica quando se utiliza essa implementação. Observámos isso no exemplo anterior: T é o tipo genérico da declaração, Integer e Double são tipos específicos usados.

## Exercícios

1. Declare duas variáveis do tipo `double`. As variáveis deverão chamar-se `valor` e `taxa`. A primeira deverá ser inicializada a `325.0` e a segunda não deverá ser inicializada. Consegue declarar as variáveis numa única instrução?

2. Declare duas variáveis, `angulo` e `velocidade`. A variável `angulo` armazena valores reais, sendo inicializada a `47.6`. A variável `velocidade` armazena valores inteiros sendo inicializada a `120`. Será possível declarar ambas as variáveis numa única instrução?

3. Assuma que `a` é uma variável do tipo `int` cujo valor é `3`. Assuma ainda que `d` é uma variável do tipo `double` cujo valor é `2.19`. Diga qual é o valor e o tipo de cada uma das expressões seguintes:

- i)  $a + 3 * a$
- ii)  $(a + 3.0) * a$
- iii)  $45 - a + 23$
- iv)  $4 - d + a / 2$
- v)  $(d + 2) / a$
- vi)  $3.24 + a * 3 \% (5 - a)$
- vii)  $2 * 5.0 / a + 3$
- viii)  $2 * 5 / a + 3$

4. Assuma que `i` e `j` são duas variáveis de tipo numérico e que `b` é uma variável do tipo `boolean`. Elimine os parênteses desnecessários de cada uma das expressões.

- i)  $((3 * i) + -4) / 2$
- ii)  $((3 * j) / (7 - i)) * (i + (-23 * j))$
- iii)  $((((i + j) + 3) + j) * (((i - 4) / j) + -323))$
- iv)  $(3 >= (j - 3)) == ((323 - (j * -7)) != (43))$
- v)  $((i > 4) != (!b)) \& (45 + 3 < (j - 4))$
- vi)  $((3 >= 5) == (!b || b))$
- vii)  $(23 + (j * 2) == ((4 * i) - 3)) != (false || (b \& (!b)))$
- viii)  $(b || (! (b \&& (3 == (i * 2)))))$

5. Para cada uma das expressões, elimine ou acrescente parênteses para que o valor das mesmas seja igual à coluna da direita. Assuma que `i` e `j` são variáveis do tipo `int` cujo valor é `3` e `10`, respectivamente. Assuma ainda que `d` é uma variável do tipo `double` cujo valor é `2.15`.

- |                                 |      |
|---------------------------------|------|
| i) $3 * i + j / 2$              | 19   |
| ii) $i * (j / 7 - i)$           | 0    |
| iii) $j + 3 \% (i - 2) + 1 * 7$ | 1    |
| iv) $(j - 2) + (7.75 / 3)$      | 5.25 |
| v) $d * (6 / (3 + 2))$          | 6.3  |
| vi) $d - 2 * (3 + 2 * d)$       | 4.75 |

6. Qual a diferença das seguintes instruções? Poder-se-ia alterar a ordem das mesmas?

```
int num = 0;  
num = 0;
```

7. Escreva uma atribuição que coloque na variável `media` a média dos valores das variáveis `nota1`, `nota2` e `nota3`. Assumindo que `nota1` armazena o valor 11, `nota2` armazena o valor 15 e `nota3` armazena o valor 14, qual será o valor presente na variável `media` após a atribuição? (sugestão: preste atenção aos tipos das variáveis envolvidas)

8. Determine o valor da variável `i` após cada uma das sequências de instruções:

- i) 

```
int j, i = 1;  
j = 3 + i * 2;  
i = j / 2 * i + 3;  
i++;
```
- ii) 

```
int i;  
double count = 3;  
count -= 2.3;  
i = (int)count;
```
- iii) 

```
double b = 3.1, c = 0;  
c += 2;  
b *= c + 3;  
int i = (int) (c + b);  
i--;
```
- iv) 

```
double a = 1.1;  
a *= 1.1;  
int b = (int) a;  
int i = ++b;
```
- v) 

```
int a = 107;  
int i = 50;  
i &= a;
```
- vi) 

```
int a = 107;  
int i = 50;  
i &= --a;
```

9. Suponha que precisamos dos seguintes dados: uma idade, uma altura, um número da lotaria, um salário, o género (feminino ou masculino) de uma pessoa, o estado civil (solteira, casada, divorciada ou viúva). Declare e inicialize variáveis adequadas para estes dados.

10. Identifique os erros existentes em cada um dos excertos de código.

- i) 

```
int a, b = 0.0;
a = b + 4;
int c; d;
d += a + 2 * b;
```
- ii) 

```
double d = 2;
int j = d + 3 * 2;
d--;
int a = ++j--;
```
- iii) 

```
int e = 2.0;
double j = e == (e<3.0);
```
- iv) 

```
boolean b = True;
int j = 3 ** 2;
b &= false;
```
- v) 

```
int count = 3;
count += 7;
int j = (double) count;
```



# 2 – Estrutura de Controlo

---

O conjunto de ordens que descreve objectivamente (sem ambiguidades nem incoerências) como partindo dos dados iniciais se pode chegar à solução de um problema denomina-se por **algoritmo**. Uma linguagem de programação é uma notação para descrever algoritmos. Designamos essas descrições por **programas**. A **sintaxe** descreve a estrutura dos programas (onde se incluem o vocabulário e a gramática da linguagem) sem se preocupar com o seu significado. A **semântica** fornece uma interpretação à sintaxe (ou seja, dá um significado ao que foi escrito), associando cada programa a um algoritmo.

## 2.1 As Três Ferramentas

À medida que o estudo da computação foi amadurecendo, chegou-se à conclusão de que existiam três conceitos básicos que, quando compostos, eram suficientes para representar qualquer algoritmo: (i) **sequência**; (ii) **condição**; e (iii) **iteração**.

### Sequência

A sequência é a capacidade de executar um conjunto de instruções sequencialmente. Já a usámos no capítulo anterior, por exemplo quando se escreveu:

```
double x = 2.0;  
int i = (int)x;
```

Intuitivamente, considerámos que a declaração e inicialização que ocorrem na primeira linha foram executadas antes da declaração, inicialização e conversão da segunda linha.

## Condição

A condição é a capacidade de escolher entre executar ou não executar um conjunto de instruções, consoante a resposta a uma dada pergunta. Por exemplo, ao dizer “vamos ao cinema se houver bilhetes” condicionamos uma acção (ir ao cinema) à resposta de uma pergunta (há bilhetes?). A instrução mais comum para exprimir esta capacidade é designada por instrução condicional. A instrução condicional, consoante a resposta (verdadeira ou falsa) a uma pergunta, executa uma de duas acções. Um exemplo em Java:

```
if (i>0)
    j = i;
else
    j = -i;
```

Esta instrução significa: calcular a resposta da expressão `i>0`; se for verdadeira (`i` é positivo) armazenar na variável `j` o valor de `i`, senão (`i` é negativo ou zero) armazenar em `j` o valor simétrico de `i`. No fim da execução, o valor de `j` é igual ao módulo do valor de `i`.

## Iteração

O terceiro e último conceito, a iteração, é a capacidade de executar repetidamente um conjunto de instruções. Esta repetição é determinada pela resposta a uma dada pergunta (de forma semelhante à da condição). Por exemplo:

```
while (i>0)
    i--;
```

Esta instrução significa: enquanto a variável `i` for maior que zero subtrai o valor de `i` em uma unidade. De notar, que se no início da iteração a variável já fosse menor ou igual a zero, a subtracção nunca seria realizada.

Denomina-se por **guarda** a expressão lógica que determina a execução ou não da próxima iteração.

## 2.2 Instruções Java

Uma **instrução** (ou **comando**) representa uma acção parte de um programa. O **estado** da execução de um programa é o conjunto dos valores de todas as variáveis num determinado momento juntamente com a próxima instrução a ser executada<sup>6</sup>. As instruções que determinam a sequência de estados quando o programa é executado também se designam

---

<sup>6</sup> É necessário incluir certas variáveis implícitas à execução do programa, como a pilha de invocações (que armazena informação sobre os métodos ainda por terminar). Falaremos mais sobre métodos no capítulo 4. As pilhas serão descritas na 3<sup>a</sup> parte do livro.

por **estrutura de controlo**. Veremos em seguida as instruções disponibilizadas pela linguagem Java. Por motivos de clareza, os exemplos são apresentados isoladamente sem pertencerem a um programa (ou seja, tal como estão não podem ser compilados). O anexo A explica como se constrói, compila e executa um programa Java.

## Instruções de Declaração

São os comandos que declaram uma nova variável associando-lhe um nome, um tipo de dados e uma posição de memória. A sintaxe Java é:

*[modificadores opcionais] tipo nome [= valor];*

Alguns exemplos:

```
int i;  
float f;  
aMinhaClasse c1;
```

Uma variável só existe depois de declarada. É possível e, na maioria dos casos, aconselhável inicializar a variável no momento da declaração.

```
int i = 1;  
float f = 0.0, g = 1.0;
```

Para declarar uma constante, i.e., uma variável com um valor fixo, prefixa-se a declaração com a palavra **final**. A declaração é acompanhada com o valor da constante.

```
final double PI = 3.1415926535;
```

Convenciona-se nomear as constantes com letras maiúsculas de modo a se salientarem, no código produzido, das restantes variáveis. Um outro exemplo:

```
final boolean SIM = true,  
NAO = false;
```

Apesar de se poder declarar uma variável muito antes dela ser usada, é aconselhável declará-la o mais tarde possível, o que na maioria dos casos é no exacto momento em que ela é utilizada pela primeira vez.

## Instruções de Atribuição

São as instruções que modificam o valor de uma ou mais variáveis, i.e., que modificam o estado do programa.

```
i = 1;  
f *= g;
```

Uma expressão com efeitos secundários é uma instrução de atribuição.

```
i++;
```

Nesta secção incluem-se igualmente a criação de novos objectos e as invocações de métodos de classes e objectos. Alguns exemplos:

```
c1 = new aMinhaClasse();
c1.oMeuMetodo();
System.out.println("olá mundo!");
```

Esta última instrução imprime literais ou conteúdos de variáveis no monitor. Voltaremos a estas instruções nos capítulos seguintes.

## Instruções de Bloco

Uma instrução de bloco (ou simplesmente, um bloco) é um conjunto de instruções entre um par de delimitadores. No caso do Java os delimitadores são as chavetas: '{' e '}'.

```
{
    int k = j*i;
    i += k + k*j;
}
```

Qualquer variável declarada dentro de um bloco é apenas visível nesse bloco (incluindo os eventuais blocos internos a esse). O espaço de visibilidade de uma variável denomina-se **âmbito** (ou contexto) da declaração dessa variável (do inglês, *variable scope*). Assim, uma variável só é visível dentro do âmbito onde foi declarada. Esse âmbito é definido pelo bloco de código onde se inclui a declaração. É possível usar o mesmo nome para variáveis em contextos distintos, sem problemas de compatibilidade entre esses nomes. Os nomes das variáveis internas de um determinado programa não entram em conflito com outros nomes, sendo um factor essencial para a modularização dos programas.

Porém, na sintaxe da linguagem Java não é permitido que um bloco com uma dada variável contenha um outro bloco com uma declaração de variável com o mesmo nome. O exemplo seguinte produz um erro de compilação:

```
{
    int i;
    {
        int i;           erro!!
    }
}
```

Já o uso de variáveis com o mesmo nome em blocos consecutivos é válido:

```
{  
    {  
        int i;  
    }  
    {  
        int i;  
    }  
}
```

## Instruções Condicionais

Estas são as instruções que fazem depender a execução de um conjunto de instruções da resposta a uma certa pergunta. A instrução mais simples deste tipo é a instrução **if**:

```
if (guarda)  
    instrução;
```

Por exemplo:

```
if (i>0)  
    i++;
```

Se a guarda for verdadeira, a instrução que se segue é executada; se for falsa, a instrução não é executada.

Para escolher entre a execução de duas instruções opta-se pela instrução **if-else**:

```
if (guarda)  
    instrução;  
else  
    instrução;
```

Por exemplo:

```
if (i>0)  
    i++;  
else  
    i--;
```

Neste caso há sempre uma instrução a ser executada: ou a que se segue à guarda se esta for verdadeira, ou a que se segue ao **else** se a guarda for falsa.

Podem-se agrupar duas ou mais instruções **if-else** para decidir entre três ou mais possibilidades:

```
if (i>0)
    x = 200;
else if (i<0)
    x = 100;
else           se i não for positivo nem negativo, i é igual a zero
    x = 150;
```

Há um problema que surge quando se combina a instrução `if` com a instrução `if-else` (em inglês, este problema designa-se por *dangling else*). No exemplo seguinte, o `else` pertence a que instrução `if`?

```
if (i1>0)
    if (i2<0)
        x = 200;
else
    x = 100;
```

Por definição, considera-se que o `else` pertence ao `if` anterior mais próximo. Para associar o `else` ao primeiro `if` teremos de utilizar uma instrução de bloco:

```
if (i1>0) {
    if (i2<0)
        x = 200;
}
else
    x = 100;
```

Existe ainda o comando `switch`. Esta instrução revela-se útil quando é preciso reunir um conjunto de condicionais.

```
switch (expressão) {
    case literal:
        instruções;
    case literal:
        instruções;
    ...
    default:
        instruções;
}
```

A expressão deste comando deve ser do tipo inteiro (`byte`, `short` ou `int`) ou um `char`, o que limita o `switch`. Cada caso está associado a um conjunto de instruções ao qual se adiciona (no fim) uma instrução `break`. Se o `break` não existir são executadas as

instruções do próximo `case`. Se nenhum dos casos descritos for igual à expressão é executado o conjunto de instruções associado ao `default`. Por exemplo:

```
switch (n) {  
    case 1:  
        m = 1.0;           se n for igual a 1, m é igual a 1.0  
        break;  
    case 2: // continua...  
    case 3:  
        m = -1.0;         se n for igual a 2 ou 3, m é igual a -1.0  
        break;  
    default:  
        m = 0.0;          senão m é igual a 0.0  
}
```

## Instruções de Iteração

As instruções de iteração (ou de ciclo) executam repetidamente uma dada instrução. É comum as linguagens de programação fornecerem três tipos de instruções de ciclo: (i) um ciclo que execute zero ou mais vezes; (ii) um ciclo que execute uma ou mais vezes; e (iii) um ciclo que execute um número determinado de vezes.

No caso do Java, a instrução (i) denomina-se `while`.

```
while (guarda)  
    instrução;
```

Por exemplo:

```
while (i>0) {  
    j*=i;  
    i--;  
}
```

O que acontece? Suponhamos que a variável inteira `j` foi inicializada a 1 e que a variável inteira `i` armazena inicialmente o valor 4. A primeira acção é avaliar o valor da expressão `i>0` que constitui a guarda: como `i` é igual a 4, a expressão é verdadeira. Sempre que a guarda é verdadeira, a instrução seguinte é executada (nesto caso, uma instrução de bloco constituída por duas instruções de atribuição) – esta instrução é designada por **corpo do ciclo**. Depois da execução do corpo de ciclo, a guarda volta a ser avaliada. Quando a guarda for falsa, o ciclo termina.

Instrução corrente	$i > 0$	$i$	$j$
Início do ciclo		<b>4</b>	<b>1</b>
$i > 0?$	true		
$j *= i;$			4
$i--;$		3	
$i > 0?$	true		
$j *= i;$			12
$i--;$		2	
$i > 0?$	true		
$j *= i;$			24
$i--;$		1	
$i > 0?$	true		
$j *= i;$			24
$i--;$		0	
$i > 0?$	false		
Fim do ciclo		<b>0</b>	<b>24</b>

Assim, a variável  $j$  armazena o conjunto de multiplicações sucessivas desde 4 (inclusive) até 1. No final do ciclo  $j$  contém o factorial de 4.

No Java, a instrução de tipo (ii) que repete uma ou mais vezes denomina-se `do-while`. Em vez da guarda se encontrar no início, está no fim (a execução do corpo do ciclo é efectuada pelo menos uma vez).

```
do
  instrução
  while (guarda)
```

Vejamos o exemplo seguinte:

```
do {
  j *= i;
  i--;
} while (i>0);
```

Para os valores iniciais  $j=1$  e  $i=4$ , este exemplo faz exactamente o mesmo que o exemplo do `while` (apenas difere o momento em que a guarda é avaliada). Porém o comportamento é diferente se  $i$  for igual a zero (porquê?). Para ser equivalente é necessário incluir uma instrução condicional extra:

```
if (i>0)
  do {
    j *= i;
    i--;
  } while (i>0);
```

Se *i* for igual a zero, a variável *j* não é alterada e é igual a 1, o que está correcto com o exemplo do ciclo while (afinal, o factorial de 0 é 1).

Em muitos ciclos existe uma variável que determina o número de iterações, sendo designada por **contador** ou **variável de progresso**. Esta variável é normalmente usada para definir a guarda (nos dois exemplos anteriores, a variável de progresso era a variável *i*).

A instrução de tipo (iii) é a instrução **for** com a seguinte sintaxe:

```
for (declarações e atribuições iniciais;
      guarda;
      instrução de incremento)
      instrução
```

Em muitos casos, o uso deste ciclo é conveniente porque organiza as instruções relativas à variável do progresso numa única linha.

Qualquer uma das três partes do ciclo **for** é opcional. As instruções de inicialização (é possível declarar variáveis locais ao ciclo) são executadas em primeiro lugar (e somente esta vez durante a execução do ciclo). De seguida é avaliada a expressão booleana que representa a guarda. Se esta for falsa, o ciclo termina. Se a guarda for verdadeira, o corpo do ciclo é executado. Em seguida é executado a instrução de incremento. Depois, volta a ser testada a guarda, o corpo, o incremento, novamente a guarda e assim sucessivamente. Quando a guarda resultar no valor falso, o ciclo terminará, como ocorre com as restantes instruções de ciclo.

Um exemplo:

```
int soma = 0;
for (int i=0; i<100; i++)
    soma++;
System.out.print(soma);
□ 100
```

A variável de progresso *i* foi usada para repetir cem vezes o corpo do ciclo, *soma++*.

A variável de progresso pode ser usada nos cálculos a efectuar no corpo do ciclo. Calculando o factorial (e considerando que as variáveis *i* e *j* foram já declaradas):

```
for (j=1, i=4; i>0; i--)
    j*=i;
System.out.print(j);
□ 24
```

Outro exemplo:

```
for (int i=0; i<20; i++)
    System.out.print(i*3 + " ");
□ 0 3 6 9 12 15 18 21 24 27 30 33 36 39 42 45 48 51 54 57
```

Este poderia ser reescrito da seguinte forma:

```
for (int i=0; i<60; i+=3)
    System.out.print(i + " ");
□ 0 3 6 9 12 15 18 21 24 27 30 33 36 39 42 45 48 51 54 57
```

O corpo do ciclo é composto por uma instrução Java (ou por várias incluídas numa instrução de bloco). Isto significa que podemos encadear ciclos. Por exemplo:

```
for (int i=0; i<3; i++)
    for (int j=1; j<4; j++)
        System.out.print("(" + i + "," + j + ")");
□ (0,1) (0,2) (0,3) (1,1) (1,2) (1,3) (2,1) (2,2) (2,3)
```

Se a guarda de um ciclo é sempre verdadeira, diz-se um **ciclo infinito**. Exemplos de ciclos infinitos:

```
int i = 0;
while (true)
    i++;

int soma=1;
do
    soma+=2;
while (soma!=0);
```

## Instruções de Salto

São instruções que quando executadas determinam a próxima instrução a executar (ao contrário das restantes instruções que, após serem executadas, ‘passam’ a vez para a instrução imediatamente a seguir).

A instrução `break` interrompe a execução da instrução `switch` (já vimos esse caso) ou do ciclo `while`, `do` ou `for` onde estiver inserida. Por exemplo:

```
for (int i=0; i<100; i++)
    if (i==resposta) {
        ...
        existeResposta = true;
        break;
    }
```

Quando a guarda do `if` é satisfeita, o `break` é executado e o ciclo `for` pára, sendo executada a próxima instrução depois do ciclo.

Na maioria dos ciclos, esta instrução é substituída complementando a guarda. Procura-se assim, concentrar todas as condições de saída na guarda tornando o código fonte mais legível. O exemplo anterior poderia ser rescrito como:

```
for (int i=0; i<100 && !existeResposta; i++)
    if (i==resposta) {
        ...
        existeResposta = true;
    }
```

O segundo comando nesta secção é o `continue`. Este comando quando executado faz com que o ciclo (onde está inserido) seja novamente iterado, não executando as restantes instruções do corpo do ciclo. No exemplo seguinte são somados apenas os valores ímpares, dado que para os pares (o resto da divisão por dois é igual a zero) a atribuição não é executada.

```
for (int i=0; i<100; i++) {
    if (i%2==0)
        continue;
    soma += i;
}
```

Do mesmo modo, na maioria dos ciclos é aconselhável rescrever o código para evitar este tipo de saltos. A seguinte versão do exemplo anterior é preferível:

```
for (int i=0; i<100; i++)
    if (i%2!=0)
        soma += i;
```

O comportamento do `continue` é ligeiramente diferente para cada uma das instruções de ciclo. Para o `while` e para o `do-while`, a execução salta para o teste da guarda. Para o `for`, a execução passa primeiro para a instrução de incremento e só depois para o teste da guarda.

A terceira instrução de salto é a instrução `return` que termina a execução do método onde se insere. Falaremos em pormenor desta instrução no capítulo 4.

## A Instrução Vazia

É a instrução que não faz nada. É representada pelo ponto e vírgula. Por exemplo:

```
while (estado.executaProximo())
    ;
```

Esta instrução é útil, por exemplo, para ciclos onde o trabalho é realizado na guarda.

## Ler e Escrever Informação

Nos exemplos anteriores utilizou-se a instrução `System.out.print()` para escrever informação no monitor. Por exemplo:

```
System.out.print(100);  
□ 100
```

A instrução `System.out.println()` produz o mesmo efeito mas adiciona uma mudança final de linha.

```
System.out.println(100);  
□ 100 ↵
```

Desde a versão Java 5 que está disponível a instrução `System.out.printf()` para escrita formatada. Esta instrução possui vários argumentos, sendo o primeiro uma *string* com a descrição da formatação desejada pelo programador. Dentro dessa string identificam-se os próximos argumentos através do símbolo % seguido de um número (que determina o espaço disponível para o argumento) mais um identificador de uma letra que determina o tipo do argumento (s para *strings*, d para inteiros, ). Alguns exemplos:

```
System.out.printf("|%5d|", 100);  
□ | 100|
```

```
System.out.printf("%s tem %d anos\n", "pedro", 24);  
□ pedro tem 24 anos ↵
```

```
System.out.printf("char:%c, int:%d, octal:%o, hex:%x",  
                  65, 65, 65, 65);  
□ char: A, int: 65, octal: 101, hex: 41
```

```
System.out.printf("%4.1f ", 20.541);  
System.out.printf("%7.2f ", 20.541);  
System.out.printf("%8.4f ", 20.541);  
□ 20.5 20.54 20.5410
```

Não existe um método igualmente directo para ler informação do teclado (mas inclui a classe `Scanner` que facilita o processo). Um exemplo do seu uso:

```
Scanner sc = new Scanner(System.in);  
System.out.println("Como se chama?");  
String nome = sc.next();  
System.out.println(nome + ", introduza um inteiro");  
int i = sc.nextInt();  
System.out.print(i*100);  
□ Como se chama? ↵
```

- Pedro
- Pedro, introduza um inteiro ↵
- 5
- 500

Os processos de leitura e escrita serão explicados em detalhe no capítulo 7.

---

## Exercícios

1. Qual o valor final da variável `bissexto`?

```
int      diasAno   = 365;
boolean bissexto = diasAno == 366;
```

2. Qual o valor final da variável `x` depois da execução de:

```
int x=0;
x += x++;
```

3. Considere o excerto de código abaixo. Se o utilizador inserisse o valor 5, que resultado seria apresentado? E se o valor introduzido fosse 6?

```
int limite = 6;
Scanner scanner = new Scanner(System.in);
System.out.println("Introduza um nº de 0 a 10:");
int resposta = scanner.nextInt();
if (resposta < limite)
    System.out.println("Abaixo do Limite");
else
    System.out.println("Acima do Limite");
```

4. Qual dos excertos de código lhe parece mais legível?

- (a) `if (x == 1) y++; else if (x > 1) y--; else y = x;`
- (b) `if (x == 1) y++
 else if (x > 1) y--
 else y = x;`
- (c) `if (x == 1)
 y++;
 else if (x > 1)
 y--;
 else
 y = x;`

```
(d) if (x == 1)
    y++;
else if (x > 1)
    y--;
else
    y = x;
```

Representam estas alíneas o mesmo algoritmo? Que algoritmo é esse?

5. Escreva um excerto de código que, dado uma variável inteira dia com valor entre 1 a 7, imprima no monitor o respectivo dia da semana (considere o Domingo como tendo valor 1). Será melhor usar o condicional `if...then...else` ou a instrução `switch`?

6. Rescreva a instrução somente com instruções `if`.

```
switch (c) {
    case 1:
        m = 1.0;
        break;
    case 2:
        p = 2.0;
    case 3:
        m = 3.0;
        break;
    default:
        m = 0.0;
}
```

7. O problema do *dangling else*: Qual o valor de x se y for igual a 0?

```
int x = 0;
if (y > 0)
    if (y < 5)
        x = 10;
else
    x = 5;
```

A identação do `else` não está correcta. Sabendo a resposta da pergunta anterior, devemos acrescentar ou remover um espaço nas duas últimas linhas deste código?

8. Considere o seguinte excerto de código. Qual destas instruções provocará um erro de compilação?

```
{  
    int i = 0;  
    System.out.println(i);  
    {  
        int j = 1;  
        System.out.println(i + j);  
    }  
    System.out.println(j);  
}
```

9. Considere os excertos de código abaixo. Qual o valor da variável teste quando cada ciclo terminar?

- (a) 

```
int teste = 10;  
while (teste > 5)  
    teste -= 2;
```
- (b) 

```
int teste = 10;  
do  
    teste -= 2;  
while (teste > 5);
```

E se o valor inicial de teste for igual a 4?

10. Rescreva o código seguinte usando (a) um ciclo *while*, (b) um ciclo *do...while*.

```
int soma = 1;  
for (int i = 1; i < 100; i++)  
    soma += i;
```

11. Considere o excerto de código abaixo. Qual o valor final da variável resultado?

```
int resultado = 0;  
for (int i = 1; i < 4; i++)  
    for (int j = 1; j < 5; j += 2)  
        resultado += j;
```

12. Considere o excerto de código abaixo. Qual o valor final da variável `x`?

```
int x = 0;
for (int i = 1; ; x++)
    if (i == 10)
        break;
else
    i++;
```

A variável de progresso do ciclo (a variável `i`) é usada de forma confusa. Poderia «arrumar» este código sem alterar o seu significado?

13. Escreva um excerto de código que calcule o somatório de 1 a 1000.

14. Escreva um excerto de código que mostre os múltiplos de 7 menores que 500.

15. Escreva um excerto de código que calcule o produto de dois números apenas com a operação soma.

16. Rescreva a operação da multiplicação usando o algoritmo aprendido na Escola Primária.

17. Repita os dois exercícios anteriores para o cálculo do quociente e do resto da divisão inteira (deve usar a subtração).

18. Escreva um excerto de código que, dado um inteiro positivo `n`, calcule se `n` é primo (um número primo só é divisível por si e pelo número 1).

19. Escreva um excerto de código que, dado um positivo `n`, calcule se `n` representa um ano bissexto (um ano é bissexto se é divisível por 4, exceptuando os anos divisíveis por 100 mas não por 400).

20. Assuma que a variável `media` contém a média das notas obtidas pelos alunos que compareceram a exame numa determinada cadeira. Escreva uma sequência de instruções (ou uma única instrução) que coloque no monitor a mensagem:

Média de exame = <nota\_média> valor(es)

Nota: onde aparece <nota\_média> deverá constar o valor da variável `media`.

21. Escreva um excerto de código que leia um inteiro do teclado para uma variável e que apresente no monitor o valor entrado elevado ao cubo, como no exemplo seguinte:

4

O cubo de 4 é 64

22. Escreva um excerto de código que leia quatro valores inteiros e apresente a média dos valores entrados.

6  
 4  
 7  
 5

A média é 5.5

23. Escreva um excerto de código que leia três números inteiros do teclado e apresente o maior número lido:

43  
 12  
 6

O maior número é 43

24. Escreva um excerto de código que leia um número entre um e dez e imprima a respectiva tabela de multiplicação:

4

$1 \times 4 = 4$   
 $2 \times 4 = 8$   
 $3 \times 4 = 12$   
 $4 \times 4 = 16$   
 $5 \times 4 = 20$   
 $6 \times 4 = 24$   
 $7 \times 4 = 28$   
 $8 \times 4 = 32$   
 $9 \times 4 = 36$   
 $10 \times 4 = 40$

25. Escreva um excerto de código que leia um valor inteiro entre 1000 e 9999 e que apresente no ecrã, o número lido, escrito da direita para a esquerda (assuma que o valor entrado está sempre dentro dos limites enunciados). Exemplo da interacção:

Introduza um valor compreendido entre 1000 e 9999  
 4325  
 Número invertido: 5234

26. Escreva um excerto de código que leia o nome de um aluno e a nota (0 a 20) obtida pelo mesmo num exame e coloque no ecrã a mensagem “O aluno <nome\_do\_aluno> foi <aprovado/reprovado> com <nota> <valor/valores>”. Note que a mensagem deverá estar de acordo com a nota obtida pelo aluno: se o João Miguel teve nota superior ou igual a 10 valores, suponhamos 15, a mensagem será “O aluno João

Miguel foi aprovado com 15 valores”; se o João Miguel obteve apenas 1 valor, a mensagem será “O aluno João Miguel foi reprovado com 1 valor”.

27. Qual o valor da variável total no fim da execução das instruções seguintes?

```
int total = 42;  
while (total>1)  
    if (total%2==0)  
        total /= 2;  
    else  
        total = 3*total + 1;
```

Este exercício é inspirado do Problema de *Collatz*. De facto, ninguém sabe se existe uma inicialização para a variável total que torne o ciclo infinito (já foram testados todos os números até  $7 \cdot 10^{11}$ ).

## 3 – Referências: Vectores

---

No capítulo 1 foram referidos dois tipos não primitivos: os vectores e as classes. Neste capítulo falaremos do primeiro caso enquanto as classes serão apresentadas no capítulo seguinte.

A Álgebra é o ramo da matemática que, entre outras coisas, ocupa-se da aplicação de operações aritméticas sobre sistemas de equações com o objectivo de as resolver ou simplificar. Neste processo, por vezes é conveniente organizar os números que multiplicam as variáveis do conjunto das equações num único quadro, organizando-os por linhas e colunas. A esta estrutura é costume designar-se matriz (uma matriz com uma coluna designa-se por vector). Por exemplo:

$$\begin{array}{rcl} 3x - 3y + 6z & = & 15 \\ 7x + y - 2z & = & 11 \\ 8x - 11z & = & 0 \end{array} \quad \Rightarrow \quad \left| \begin{array}{ccc|c} 3 & -3 & 6 & x \\ 7 & 1 & -2 & y \\ 8 & 0 & -11 & z \end{array} \right| = \left| \begin{array}{c} 15 \\ 11 \\ 0 \end{array} \right|$$

Muitas linguagens de programação, incluindo o Java, definem um tipo de dados apropriado para este género de informação a qual designamos por **vector** (do inglês, *array*). Uma das características dos vectores é armazenarem elementos do mesmo tipo. Que tipo? Define-se o tipo da informação quando se declara um novo vector através do operador `[ ]`. Por exemplo:

```
int[] a;  
double[] b;  
boolean[] c;
```

declara *a* como um vector de inteiros  
declara *b* como um vector de reais  
declara *c* como um vector de booleanos

Não existe restrição sobre os tipos armazenados em vectores. Nos exemplos anteriores, foram declarados vectores com elementos que pertencem a tipos primitivos Java. Se existir um novo tipo de dados Ponto2D (para representar pontos a duas dimensões) a declaração seguinte seria válida:

Ponto2D[] d; declara um vector de pontos 2D

Qualquer tipo primitivo ou tipo de referência construído com classes e/ou vectores pode ser agrupado num vector. Não esquecer, porém, a restrição que todos os elementos do vector devem ser do mesmo tipo.

## Criação de Vectores

Aqui reside uma diferença fundamental entre os tipos primitivos referidos no capítulo 1 e os tipos de referência (i.e., vectores e classes): enquanto as variáveis de tipos primitivos referem-se à informação armazenada, as variáveis de tipos de referência referem-se indirectamente, armazenando o endereço de memória onde a informação está armazenada. Por exemplo:

`int i;` declara uma variável inteira  
`int[] a;` declara uma referência para um vector de inteiros

Poderíamos dizer que *i* é o nome da memória onde se encontra um inteiro, enquanto a armazena o endereço da memória onde está guardado o vector de inteiros. Em diagrama:

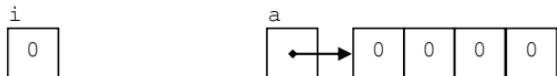


A seta na caixa da variável a refere que a informação não é armazenada nesse local, apenas é armazenado o endereço da informação. Os pontos de interrogação identificam a dúvida seguinte: qual endereço? De forma geral a resposta é: não sabemos que endereço é esse. O importante é que a máquina que executa o programa seja capaz de aceder à informação. Esses detalhes são transparentes para o programador e geridos automaticamente pelo computador.

Se tentássemos armazenar uma informação no vector, essa acção provocaria um erro. Porquê? Porque declarámos apenas uma referência para um vector de inteiros, ou seja, onde armazenar o endereço desse vector. *Ainda não reservámos a memória para armazenar o próprio vector!* Como fazer? Por exemplo:

```
a = new int[4];
```

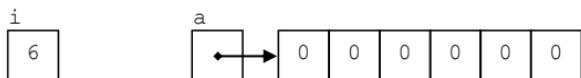
O comando `new` reserva memória suficiente para armazenar um objecto do tipo de referência que se segue. Este comando, depois de reservar a memória, devolve o endereço onde essa memória se encontra. Na instrução anterior foi criado um vector de 4 inteiros, armazenando o endereço desse vector na variável `a`. No diagrama:



Todos os elementos de um vector são inicializados com o valor por defeito do tipo a que pertencem.

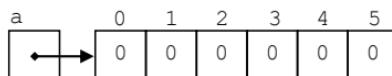
Um vector é uma estrutura de dados estática: uma vez definida, a dimensão permanece inalterada enquanto o vector existir. Porém, esta dimensão é determinada durante a execução, no exacto momento da criação do novo objecto vector (i.e., quando o operador `new` é executado). É possível definir a dimensão do vector a partir do valor de uma variável.

```
i = 6;
a = new int[i];
```



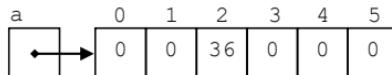
## Acesso aos elementos de um Vector

Uma vez criado o espaço de memória para armazenar os elementos do vector, é possível aceder a esses mesmos elementos. Na linguagem Java o acesso faz-se através do operador `[ ]`. Cada elemento do vector é identificado por um índice inteiro. Por convenção, considera-se que o primeiro elemento do vector é identificado pelo índice de valor zero, o segundo elemento pelo índice um, o terceiro pelo índice dois e assim sucessivamente.



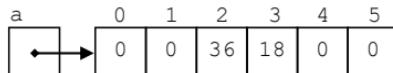
Por exemplo, para atribuir um valor ao terceiro elemento do vector a escreve-se:

```
a[2] = 36;
```



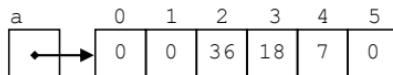
Para armazenar no quarto elemento, a metade do terceiro elemento:

```
a[3] = a[2] / 2;
```



É igualmente possível usar expressões inteiras (o índice de um vector é um inteiro) dentro do operador de acesso:

```
i = 9;  
a[ a[2]/i ] = 7;
```



Estas expressões não podem ser do tipo `long`, o que limita a dimensão máxima de um vector à dimensão máxima do maior inteiro positivo:  $2^{31} - 1$ . Esta limitação é relativa pois mesmo assim, um vector pode conter mais de dois mil milhões de elementos.

É possível conhecer a dimensão de um vector através do atributo `length`. O exemplo seguinte calcula o somatório de todos os elementos do vector de inteiros `a`.

```
int soma = 0;  
for (int i=0; i<a.length; i++)  
    soma += a[i];
```

Qualquer acesso a partir de um índice negativo (por exemplo `a[-1]`) ou de índice maior que a dimensão do vector é detectado e resulta no lançamento de uma exceção `ArrayIndexOutOfBoundsException` (as exceções são referidas no capítulo 6).

Foi inserido, a partir da versão Java 5, uma nova versão do comando `for`. Este comando itera automaticamente sobre as posições de um vector. O exemplo anterior poderia ser descrito da seguinte forma:

```
int soma = 0;  
for (int elemento : a)  
    soma += elemento;
```

Onde cada índice do vector `a` seria percorrido automaticamente durante o ciclo, armazenando o respectivo valor, em cada iteração, na variável `elemento`.

## Literais de Vectores

Da mesma forma que é possível representar literais de tipos primitivos (por exemplo, `1000` é o literal inteiro que representa o número mil), é possível representar literais de vectores.

A palavra `null` representa a ausência de vector. A instrução seguinte declara um vector de inteiros /e indica que não há nenhum vector associado a esta referência.

```
int[] a = null;
```

Em diagrama, representamos essa situação da forma seguinte:



Outra forma de representar vectores é descrever o conteúdo de cada um dos elementos do vector na declaração do mesmo. Alguns exemplos:

```
int[] a = { -4, -3, -2, -1, 0, 1, 2, 3, 4 };
double[] b = { 1.0, -20.1, 0.0 };
boolean[] c = { true, true, false };
```

Neste caso não se usa a palavra `new`. A reserva da memória é implícita no acto da inicialização.

Existe ainda uma outra forma de usar literais de vectores. Esta opção é útil quando se cria um vector para uma dada utilização mas não interessa o seu nome – são denominados por **vectores anónimos**. No exemplo seguinte é invocado o método *MasterMind* com a sequência `aadcb` (veremos o que é um método no capítulo seguinte).

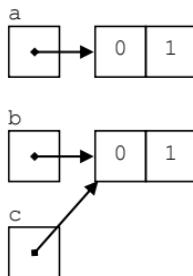
```
masterMind(new char[] {'a', 'a', 'd', 'c', 'b'});
```

Neste exemplo, como não era necessário associar o vector a um nome, utilizou-se um vector anónimo.

## Operações sobre Vectores

O tipo vector é constituído pelo conjunto dos objectos vectores juntamente com dois operadores: a atribuição e a comparação. A atribuição associa a referência de um vector a um conteúdo. A comparação compara duas referências (não conteúdos). Considere o exemplo seguinte:

```
int[] a = {0, 1};
int[] b = {0, 1};
int[] c = b;
```



Depois de executar estas três instruções, a expressão `a==b` seria falsa (apesar dos conteúdos dos dois objectos vector referenciados serem iguais) e a expressão `b==c` seria verdadeira (as variáveis `b` e `c` referenciam o mesmo objecto).

Ao contrário, por exemplo, do tipo inteiro onde existem várias operações, não é possível encontrar muitas operações sobre endereços para além de poder compará-los e copiá-los (respectivamente com os operadores `==` e `=`).

## Vectores Multidimensionais

É possível em Java criar estruturas vectoriais com mais do que uma dimensão, denominadas por **vectores multidimensionais** (costuma-se designar por matriz se for um vector de duas dimensões). Para declarar vectores multidimensionais adiciona-se um operador `[]` por cada dimensão do vector.

As instruções seguintes declaram referências para um vector inteiro bidimensional, um vector real tridimensional e um vector booleano tetradimensional:

```
int[][] a;
double[][][] b;
boolean[][][] c;
```

Para criar um vector multidimensional utiliza-se, novamente, o operador `new`.

```
int[][] quatroEmLinha = new int[7][6];
```

O que faz esta instrução Java?

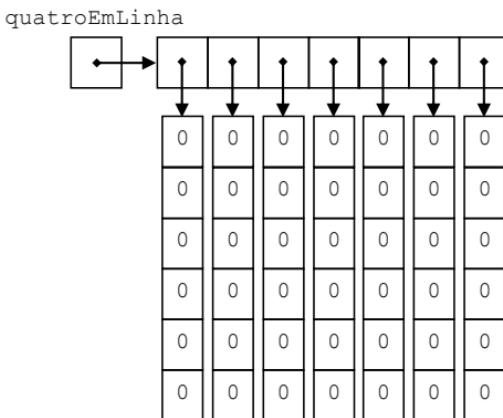
- Em primeiro lugar, declara a variável `quatroEmLinha` como sendo uma referência para um vector bidimensional de inteiros.
- A seguir, cria em memória um vector de 7 referências para vectores de inteiros unidimensionais.
- Depois, cria em memória 7 vectores de 6 inteiros cada e armazena essas referências no vector anterior.
- Em quarto lugar, inicializa todos os 42 elementos desses 7 vectores com o valor por defeito do tipo primitivo `int` (o valor zero).

- Finalmente, devolve a referência do endereço de memória do primeiro vector e armazena-a na variável `quatroEmLinha`.

De outra forma, é o mesmo que:

```
int[][] quatroEmLinha = new int[7][];
for (int i=0; i<7; i++)
    quatroEmLinha[i] = new int[6];
```

Em diagrama:

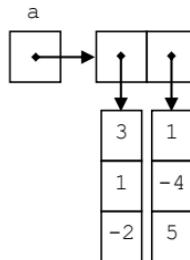


Todas as dimensões de um vector, excepto a última, são referências para vectores com dimensões progressivamente menores até ao último nível onde se armazena efectivamente a informação. É admissível definir parcialmente um vector multidimensional reservando memória para as primeiras dimensões. No exemplo seguinte, as duas primeiras linhas são válidas, mas as duas últimas não são (tenta-se armazenar referências de memória em variáveis que ainda não possuem memória definida).

- `int[][] a = new int [10][];`
- `double[][][] b = new double [3][50][];`
- `int[][] a = new int [] [10];`
- `boolean[][][][] c = new boolean [2] [] [2] [10];`

É possível inicializar um vector multidimensional com um literal:

```
int[][] a = { {3, 1, -2}, {1, -4, 5} };
```



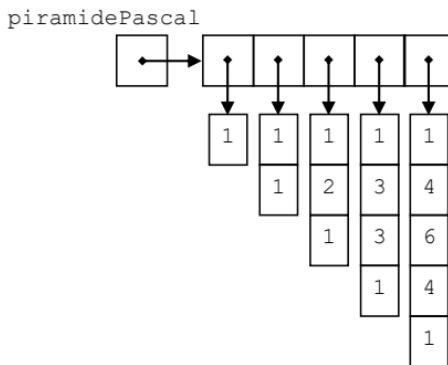
Ou com um vector anónimo:

```
int a = new int[][] { {3, 1, -2}, {1, -4, 5} };
```

Temos sempre referido vectores rectangulares. Tendo em conta a forma como é organizada a memória dos vectores, é possível criar estruturas vectoriais não rectangulares.

```
int[][] piramidePascal = { {1},  
                           {1, 1},  
                           {1, 2, 1},  
                           {1, 3, 3, 1}  
                           {1, 4, 6, 4, 1}  
};
```

Em diagrama:



## Exercícios

1. Declare: (a) um vector de inteiros, (b) um vector de booleanos, (c) um vector capaz de armazenar médias de alunos.

2. Qual o objectivo do operador `new`?

3. Qual a diferença entre os seguintes pares de linhas?

- (a) `int v1;`  
`int[] v2;`
- (b) `int[] v1;`  
`long[] v2;`
- (c) `int[] v1;`  
`int[] v2 = new int[10];`
- (d) `int[] v1 = new int[70];`  
`int[] v2 = new int[10];`

4. Considere o seguinte código:

```
int[] v1 = new int[100];
int[] v2 = new int[10];
```

- (a) Como atribuir o valor inteiro 1 ao último índice do vector `v1`?
- (b) Como guardar no primeiro índice de `v1` a soma do primeiro e último índice de `v2`?
- (c) Que valor devolve a expressão `v1.length + v2.length`?
- (d) Escreva um programa que peça ao utilizador valores inteiros para inicializar `v2` (começando a preencher por `v2[9]` e terminando em `v2[0]`).

5. Considere o seguinte código. A que valor é iniciada a variável `x`? Se no primeiro índice do vector estivesse guardado o valor 4, a execução do programa produziria um erro. Qual?

```
int[] v = {3, 1, 4, 2};
int x = v[v[0]];
```

6. Qual o valor de todas as variáveis após a execução do seguinte código?

```
int[] a = {1, 2, 3},
      b = {4, 5, 6},
      c = b;

boolean d = c == a,
       e = c == b,
       f = a == null;

c[0] = 7;
int g = b[0];
```

7. Escreva um ciclo que apresente no monitor todos os elementos de um vector unidimensional.
8. Escreva um ciclo que some os elementos de um vector unidimensional.
9. Escreva um excerto de código some os elementos de um vector bidimensional.
10. Dado o vector  $v$  iniciado da seguinte forma:

```
int[] v = new int[100];  
for(int i=0; i<100; i++)  
    v[i] = i*i;
```

declare e inicialize o vector  $u$  com o conteúdo de  $v$  mas de forma invertida (o primeiro elemento de  $v$  na última posição de  $u$ , o segundo na penúltima e assim por diante).

11. Escreva uma sequência de instruções capaz de encontrar o elemento máximo e mínimo de um vector de inteiros.
12. Escreva uma sequência de instruções capaz de encontrar o último índice do maior elemento de um vector de inteiros (ou seja, assuma a existência de repetições).
13. Escreva uma sequência de instruções capaz de encontrar o segundo maior elemento de um vector de inteiros (sem contar com repetições).
14. Escreva uma sequência de instruções capaz de verificar se existem dois zeros consecutivos num vector de inteiros.
15. Dado um vector de inteiros ordenados de forma crescente descubra a média, a mediana e a moda desse vector (A *média* é igual à divisão entre a soma dos valores do conjunto e o número total dos valores; A *mediana* é o valor situado na posição que separa o conjunto – o conjunto deve estar ordenado – em dois subconjuntos com o mesmo número de elementos. A *moda* é o valor que ocorre com maior frequência na lista dos valores).
16. Dado um vector de inteiros, crie um novo vector de reais com a mesma dimensão e com os valores normalizados (dividem-se todos os valores da lista pelo valor máximo da lista).
17. Dado um vector de inteiros `notasAlunos`, calcule um vector `histograma`, onde a sua  $i$ -ésima posição contém o número de notas iguais a  $i$  contidas no vector inicial.
18. Usando o vector `notasAlunos` do exercício anterior, crie um vector booleano `aprovou`, onde a sua  $i$ -ésima posição é verdadeira se e só se a  $i$ -ésima posição de `notasAlunos` conter uma nota maior ou igual a 10.

19. Considere o seguinte código:

```
int[][] a;  
int[][] b = new int[5][];  
int[][] c = new int[5][3];  
int[][] d = { {1, 2, 3}, {-1,-2,-3} };
```

Represente graficamente qual a estrutura da memória criada por cada uma destas declarações/inicializações.

20. Dado um vector tridimensional de inteiros determinar o número de elementos que esse vector efectivamente armazena em memória.

21. Escreva um excerto de código capaz de iniciar um vector triangular como o da página anterior, com os valores do triângulo de Pascal, para N linhas.



# 4 – Referências: Classes

---

Desde muito cedo, quando aprendemos a ler, a escrever e a estruturar o nosso pensamento, aprendemos que certas palavras referem-se a objectos que conseguimos identificar. Outras vezes, essas mesmas palavras servem para referir conceitos mais abstractos, no sentido em que as entidades a que as palavras se referem não existem realmente.

Por exemplo, quando dizemos “o meu carro”, estamos a referir-nos a um veículo específico: de uma dada marca, de uma dada cor juntamente com outras características. É um objecto concreto e identificável. No entanto, se dissermos “o carro é o principal meio de transporte da sociedade moderna” não nos referimos a qualquer carro em particular, mas sim ao conceito que identifica o automóvel que, com maior ou menor precisão, guardamos na nossa mente. Neste segundo caso, a palavra refere-se a uma noção abstracta, a uma classe de objectos que classificamos como carros.

Numa linguagem de programação, uma noção informal de classe não serve. À partida, qualquer objecto de uma classe partilhará o mesmo tipo de propriedades e terá o mesmo tipo de comportamento. Neste contexto, uma classe representa todos os objectos possíveis que partilham um conjunto comum de **componentes** (ou características) composto por **atributos** (ou propriedades) e por **operações**.

Considere o conjunto simplificado de atributos e operações que definem um carro:

Tabela 4.1 – A classe Carro

Atributos	Operações
<i>matrícula</i>	<i>arrancar</i>
<i>modelo</i>	<i>travar</i>

<i>ano</i>	<i>acelerar</i>
<i>bonsTravões</i>	<i>verIdade</i>
<i>velocidade</i>	

A classe define um tipo. Porquê? Porque define um conjunto de valores possíveis para os elementos desse tipo (dados pelos atributos) e um conjunto de operações possíveis sobre esses elementos. Neste exemplo, qualquer elemento X que pertença ao tipo Carro diz-se um objecto da classe Carro, ou simplesmente, o objecto X é um Carro.

Cada objecto do tipo Carro possui um conjunto de atributos próprios que o caracteriza (e eventualmente o distingue dos outros objectos). O produto cartesiano dos tipos dos atributos de uma classe designa-se por **configuração** dos objectos dessa classe. Um conjunto específico de valores pertencente à configuração designa-se por **estado**. Assim, um objecto X da classe Y – diz-se que X é uma **instância** de Y – é caracterizado por um estado que pertence à configuração da classe. Este estado não é necessariamente fixo, ele costuma variar durante a execução do programa.

Por exemplo, a configuração da classe Carro descrita pela tabela anterior é o produto cartesiano `String × String × int × boolean × double`. O objecto que representa um *Renault Clio* com bons travões, matrícula 43-21-UV de 2003 que se desloca à velocidade de 65.3 Km/h, tem o estado:

*matrícula*: 43-21-UV  
*modelo*: Renault Clio  
*ano*: 2003  
*bonsTravões*: true  
*velocidade*: 65.3

Para além do conjunto de atributos, cada objecto carro partilha um determinado conjunto de operações que se dividem entre **interrogações** e **comandos**:

- Uma interrogação devolve uma informação sobre o estado do objecto (no exemplo, *verIdade* devolve a idade de um dado carro).
- Um comando corresponde à alteração do estado do objecto (a operação *travar* diminui o atributo *velocidade*).

Uma interrogação não altera o estado do objecto. Os comandos modificam um ou mais atributos do estado do objecto. O conjunto das operações de uma dada classe representa o **comportamento** desses objectos. Assim, cada objecto é constituído por um estado (pertencente à configuração da sua classe) e pelo comportamento da classe a que pertence.

## 4.1 O tipo Classe

Uma **classe** (do inglês, *class*) em Java é uma estrutura de dados com uma configuração definida pelo produto cartesiano de zero ou mais tipos (eventualmente diferentes) e com um conjunto de zero ou mais operações que definem o comportamento dos objectos da

classe. A classe é um conceito que inclui memória (a configuração no formato de atributos) e computação (o comportamento no formato de operações).

Três diferenças importantes em relação aos vectores: (i) cada nova classe define um novo tipo (nos vectores não são criados novos tipos); (ii) a configuração da classe pode ser constituída por tipos diferentes (nos vectores, todos os elementos são do mesmo tipo); (iii) as diversas operações são implementadas pelo programador (nos vectores, as operações são fixas).

A classe é o conceito mais fundamental das linguagens centradas em objectos, onde se inclui o Java. Cada programa Java é constituído por uma ou mais classes. As variáveis e instruções têm, obrigatoriamente, de ser incluídas dentro de classes. A classe é a unidade básica de um programa (há programas só com uma classe, mas não há programas sem a definição de pelo menos uma classe).

## Declaração

O seguinte programa Java declara uma referência para objectos da classe Carro:

```
public class Start {           // ficheiro Start.java
    public static void main(String[] args) {
        Carro oMeuCarro;
    }
}
```

O compilador entende a nossa intenção: declarar uma nova variável `oMeuCarro` do tipo `Carro`. Mas onde está a definição deste tipo? Qual é a estrutura? Antes de declarar uma variável de um tipo não primitivo precisamos definir esse tipo. Para isso, é necessário criar um novo ficheiro `Carro.java` referente à definição desta nova classe. Cada classe inicia-se com um cabeçalho semelhante ao seguinte:

```
public class Carro {           // ficheiro Carro.java
    ... // definição da classe
}
```

Dentro do bloco adicionamos os atributos e operações que forem convenientes.

Segue a definição da configuração da classe `Carro` (veremos o porquê das palavras `public` e `private` no capítulo 9).

```
public class Carro {
    private String matricula;
    private String modelo;
    private int ano;
    private boolean bonsTravoes;
```

```
private double velocidade;  
...  
}
```

Além de definir a configuração é necessário incluir o comportamento dos futuros objectos da classe. Este comportamento é definido por um conjunto de métodos apropriados.

## Métodos

Um **método** (do inglês, *method*) é uma instrução de bloco com um cabeçalho. Qualquer método pertence a uma classe. Cada método corresponderá a uma operação incluída no comportamento da classe. Por exemplo:

```
public class Carro {  
    ...  
    public void arrancar() {  
        velocidade = 10;  
    }  
}
```

O método `arrancar()` pertencente à classe `Carro` modifica o atributo `velocidade` para o valor 10. Mas quem é modificado? Não a classe, porque a classe apenas define a configuração e o comportamento dos objectos. Quem é modificado é o objecto específico alvo da **invocação** do método.

```
public class Start {  
    public static void main(String[] args) {  
        Carro oMeuCarro;  
        ...  
        oMeuCarro.arrancar();  
    }  
}
```

A forma de acesso a um componente de um objecto é através do operador ponto, `.`, que permite o acesso (aos atributos e métodos) do objecto referenciado pela variável. O acesso aos componentes acima designa-se por **acesso qualificado** dado se explicitar o objecto alvo da invocação (por isso, no exemplo anterior, a instrução `velocidade = 10` refere-se ao atributo de um objecto bem definido, o referenciado por `oMeuCarro`). Já no âmbito dos métodos da classe não é necessário indicar o objecto, dado ser implícito no contexto da execução. Nesses casos, os acessos aos atributos e aos métodos são **acessos não qualificados**.

Alguns métodos precisam de valores auxiliares para definir correctamente o problema em questão. Por exemplo, para conhecer a idade do carro não basta saber o ano da sua construção, é igualmente necessário saber em que ano nos encontramos.

```
public class Carro {  
    ...  
    private int ano;  
    ...  
    public int verIdade(int anoActual) {  
        return (anoActual+1) - ano;  
    }  
}
```

Deste modo diz-se que a variável `anoActual` é um **parâmetro** do método `verIdade()`.

A invocação de um método com parâmetros necessita de um conjunto apropriado de **argumentos** que definem os valores dos parâmetros do método invocado. Uma invocação do método anterior:

```
public class Start {  
    public static void main(String[] args) {  
        Carro oMeuCarro;  
        ...  
        int i = oMeuCarro.verIdade(2003);  
    }  
}
```

O método `verIdade()` mostra também ser possível devolver um valor para quem invocou o método. O tipo de informação devolvido é previamente definido no cabeçalho da definição do método. O valor devolvido é calculado para cada invocação do método e é dado pela expressão associada à instrução `return` executada. O tipo `void` é usado para os métodos que não devolvem qualquer tipo de informação. Por vezes, denomina-se por **função** um método que devolve um valor e por **procedimento** um método que não devolve um valor.

Certos métodos necessitam de variáveis auxiliares. No próximo exemplo, a variável `v` é utilizada para ajudar a determinar quando se deve travar com maior ou menor intensidade.

```
public class Carro {  
    ...  
    public void travar() {  
        double v = bonsTravoes ? 10 : 20;  
        if (velocidade / v >= 5)  
            velocidade /= 1.5;  
        else  
            velocidade /= 1.1;  
    }  
}
```

Resumindo, um método é constituído por:

- Uma **assinatura**, i.e., um nome e uma lista dos tipos e nomes dos parâmetros.
- Uma instrução de bloco.
- O tipo de valor devolvido. Se o método não devolve um valor específico, o tipo devolvido é `void`.
- Um conjunto de exceções que pode lançar (ler capítulo 6)
- Um conjunto de modificadores de acesso do método (ler capítulo 9).

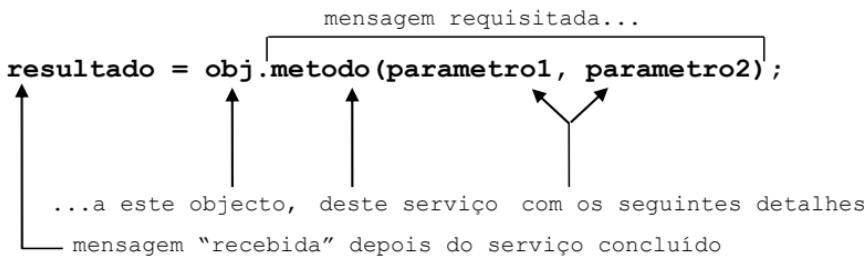
A estrutura sintáctica destas componentes é a seguinte:

```
[modificadores de acesso]
tipo-devolvido nome-método ( [tipo-1 parametro-1], [tipo-2 parametro-2] ... )
[excepções-lançadas]
{ [instruções java] }
```

Designa-se por **cabeçalho** toda a informação do método (assinatura, tipo devolvido, modificadores de acesso e exceções lançadas) antes da instrução de bloco.

## Acessos como Troca de Mensagens

Pode-se interpretar o acesso de um objecto X aos componentes de outro objecto Y como uma acção de troca de informação entre os dois objectos: X pede algo a Y e Y responde. A base dessa troca de informação (e consequentemente de todo o fluxo de dados do sistema) é a **mensagem**, i.e., a invocação de um método de um objecto, juntamente com os argumentos. Se o método devolver um valor, esse valor é visto como uma mensagem de resposta.



## Criação de Objectos e Construtores

Considere novamente a classe em que se declarou uma referência para um carro:

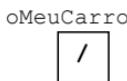
```
public class Start {
    public static void main(String[] args) {
```

```

Carro oMeuCarro;
...
oMeuCarro.arrancar();
}
}

```

As reticências não foram postas por acaso. Como nos vectores, o que ocorreu com a declaração foi a criação de uma referência para um objecto do tipo Carro doravante conhecida como oMeuCarro. Porém, ainda não foi reservada a memória para armazenar o objecto.



Para criar um novo objecto é necessário utilizar o operador new (como nos vectores).

```

public class Start {
    public static void main(String[] args) {
        Carro oMeuCarro;
        oMeuCarro = new Carro();
        oMeuCarro.arrancar();
    }
}

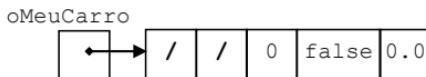
```

O **construtor** de uma classe (no exemplo `Carro()`) é um método especial executado de cada vez que um objecto dessa classe é criado. Este método contém normalmente as inicializações dos atributos do novo objecto, mas pode incluir outras instruções conforme as necessidades. Um construtor tem o mesmo nome da classe. Por omissão, é executado um construtor que inicializa os atributos do novo objecto com os valores por defeito dos tipos dos atributos (inicializa a 0 se for um tipo inteiro, a null se for um tipo por referência, etc.).

O operador new realiza as tarefas seguintes:

- Reserva memória suficiente para armazenar um objecto do tipo em questão,
- Inicializa os atributos desse objecto com os valores por defeito dos respectivos tipos,
- Executa o construtor indicado na instrução,
- Devolve o endereço da memória reservada.

No exemplo anterior, depois de executar `oMeuCarro = new Carro();`:



O programador pode definir os seus próprios construtores. Uma vez definido um novo construtor, o compilador deixa de executar o construtor por omissão. Podem existir diversos construtores com listas de parâmetros diversas – são essas assinaturas que distinguem qual o construtor desejado no momento da criação. É comum ter um construtor que inicializa os atributos a partir dos argumentos do construtor (ver o exemplo seguinte).

A classe Carro com a definição dos atributos, dos métodos e do seu construtor:

```
public class Carro {  
    private String matricula;  
    private String modelo;  
    private int ano;  
    private boolean bonsTravoes;  
    private double velocidade;  
  
    public Carro(String mat, String mod,  
                 int a,         boolean bt) {  
        matricula = mat;  
        modelo   = mod;  
        ano       = a;  
        bonsTravoes = bt;  
        velocidade = 0;      // vel. inicial é sempre zero  
    }  
  
    public void arrancar() {  
        velocidade = 10;  
    }  
  
    public void acelerar() {  
        velocidade *= 1.1;  
    }  
  
    public void travar() {  
        double v = bonsTravoes ? 10 : 20;  
  
        if (velocidade/v>=5)  
            velocidade /= 1.5;  
        else  
            velocidade /= 1.1;  
    }  
  
    public int verIdade(int anoActual) {  
        anoActual++;  
        return anoActual - ano;  
    }  
} //endClass Carro
```

Agora já é possível uma criação mais personalizada de um objecto carro:

```
public class Start {
    public static void main(String[] args) {
        Carro oMeuCarro;
        oMeuCarro =
            new Carro("43-21-UV", "Clio", 2003, true);
        oMeuCarro.arrancar();
    }
}
```

O objecto depois da criação e da invocação do método `arrancar()`:



Existem diferenças fundamentais entre os conceitos de classe e objecto. A classe é um tipo, um molde ou padrão previamente desenhado e posteriormente usado para a criação de objectos. Uma classe é um modelo definido durante a construção de um programa. Um objecto é uma instância de uma classe, criado e manipulado durante a execução desse programa.

## A Referência “this”

Na invocação de cada método é preciso explicitar o objecto alvo da invocação. No exemplo seguinte, o objecto referente à invocação do método `igual()` é o objecto referenciado por `conjuntoA`.

```
conjuntoA.igual(conjuntoB);
```

Pode-se referir dentro do método a esse objecto pela palavra `this`.

Considere que a seguinte classe `Conjunto` inclui um método `contem()` que determina se um conjunto está contido noutra. O método que verifica se dois conjuntos são iguais poderia ser implementado da seguinte forma:

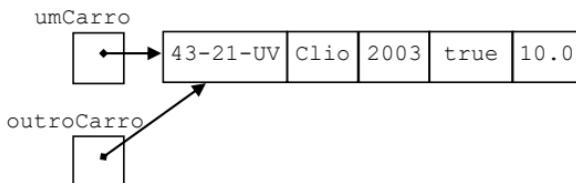
```
public class Conjunto {
    ...
    public boolean igual(Conjunto s) {
        return contem(s) && s.contem(this);
    }
}
```

## Atribuição vs. Cópia

É necessária atenção com a manipulação de referências para objectos. Considere o programa:

```
public class Start {
    public static void main(String[] args) {
        Carro umCarro =
            new Carro("43-21-UV", "Clio", 2003, true);
        Carro outroCarro = umCarro;
    }
}
```

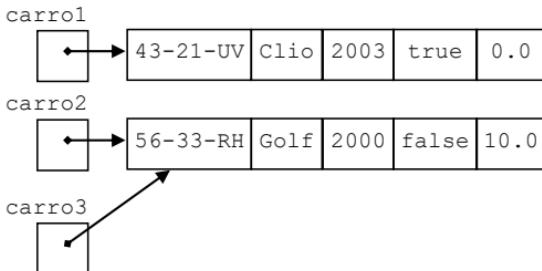
Depois da executar a segunda declaração e inicialização, a variável `outroCarro` possui a mesma referência que `umCarro`. Não existe um segundo objecto.



Existem duas variáveis (`umCarro`, `outroCarro`) que referem o mesmo objecto. Em inglês (à falta de um termo apropriado em português) diz-se que as variáveis são *aliases*. Este mecanismo (*aliasing*) é comum em muitas linguagens de programação e é extensivamente utilizado na programação. De notar que a invocação de um método a partir de uma variável produz alterações sobre todos os seus *aliases*. Por exemplo, a execução de `outroCarro.arrancar()` afecta igualmente o objecto referenciado por `umCarro` (afinal, é o mesmo objecto).

A execução do próximo programa cria os seguintes objectos:

```
public class Start {
    public static void main(String[] args) {
        Carro carrol =
            new Carro("43-21-UV", "Clio", 2003, true);
        Carro carro2 =
            new Carro("56-33-RH", "Golf", 2000, false);
        Carro carro3 = carro2;
        carro3.arrancar();
    }
}
```

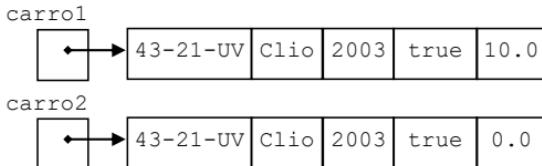


Para duplicar – ou **clonar** – um objecto é preciso fazê-lo de forma explícita.

```
public class Carro {
    ...
    public Carro copiar() {
        Carro copia =
            new Carro(matricula,modelo,ano,bonsTravoes);
        copia.velocidade = velocidade;
        return copia;
    }
}
```

Para o próximo programa obtemos os objectos abaixo:

```
public class Start {
    public static void main(String[] args) {
        Carro carrol =
            new Carro("43-21-UV", "Clio", 2003, true);
        Carro carro2 = carrol.copiar();
        carrol.arrancar();
    }
}
```



Para uma referência não estar associada a qualquer objecto, deve-se associar a referência ao valor **null**. Diz-se então uma **referência vazia** ou nula. Este valor especial pode ser armazenado em qualquer variável de tipo referência (vector ou objecto). Nunca se deve tentar aceder a um componente de um objecto a partir de uma referência nula. Porquê?

Porque não existe objecto para ser acedido. Tentar aceder a um componente a partir de uma referência vazia provoca um erro de execução.

## Identidade vs. Equivalência

É possível utilizar o operador booleano de igualdade, `==`, entre objectos da mesma classe. Não são comparadas as configurações dos objectos, apenas as referências. Se a comparação for verdadeira, diz-se que os objectos referenciados são **idênticos**. No exemplo seguinte, depois da execução das três primeiras instruções, a expressão `carrol==carro3` é verdadeira, mas `carrol==carro2` é falsa.

```
public class Start {  
    public static void main(String[] args) {  
        Carro carrol =  
            new Carro("43-21-UV", "Clio", 2003, true);  
        Carro carro2 = carrol.copiar();  
        Carro carro3 = carrol;  
        ...  
    }  
}
```

Para comparar o estado de dois objectos, i.e., verificar se são **equivalentes**, é necessário programá-lo explicitamente.

```
public class Carro {  
    ...  
    // dois objectos Carro são equivalentes se  
    // tiverem a mesma matricula  
    public boolean equals(Carro outroCarro) {  
        return matricula.equals(outroCarro.matricula);  
    }  
}
```

Curiosamente, as expressões `a.equals(b)` e `b.equals(a)` não têm necessariamente de devolver o mesmo resultado. Porquê? Se, por exemplo, a variável `a` referenciar um objecto válido e `b` armazenar a referência vazia, a primeira expressão é falsa e a segunda expressão produz um erro de execução (não existe objecto para invocar o método `equals()`).

## Parâmetros por Referência e por Valor

Outra questão importante é saber o que acontece quando alteramos no corpo do método os valores dos parâmetros. Revendo o método `verIdade()`:

```
public class Carro {  
    ...  
    public int verIdade(int anoActual) {  
        anoActual++;  
        return anoActual - ano;  
    }  
}
```

O parâmetro `anoActual` só existe durante a execução do método, o seu valor foi alterado pela instrução `anoActual++`. Mas o que acontece ao argumento usado na invocação do método? Por exemplo:

```
public class Start {  
    public static void main(String[] args) {  
        Carro oMeuCarro;  
        ...  
        int esteAno = 2003;  
        int i = oMeuCarro.verIdade(esteAno);  
        // aqui, qual o valor de 'esteAno'? 2003 ou 2004?  
    }  
}
```

Nas linguagens de programação existem dois mecanismos de passagem de argumentos para métodos: a **passagem por valor** e a **passagem por referência**.

- Na passagem por valor o sistema cria uma cópia do argumento e associa-a ao parâmetro do método. Assim, qualquer modificação afecta apenas a cópia, não o original. A execução do método não produz efeitos sobre os argumentos usados na sua invocação.
- Na passagem por referência, o associado ao parâmetro do método é a referência da variável usada na invocação. Assim, qualquer alteração do parâmetro durante a execução do método reflecte-se no valor do argumento correspondente.

No caso do Java, *todos os argumentos são passados por valor*. Existe, porém, um ponto importante: nos casos dos vectores e dos objectos é realizada somente a cópia da referência do argumento, *não a cópia do objecto referenciado*.

No exemplo anterior, foi criada uma cópia do argumento `esteAno` e foi sobre essa cópia que a modificação ocorreu (`anoActual++`). Logo, a variável `esteAno` mantém-se inalterada.

Não existe passagem por referência em Java, mas ao passar-se um objecto de uma classe como parâmetro de um método e este, quando executado, modificar o estado do objecto as alterações permanecem. Considere o exemplo:

```
void modificar(C o) {  
    o.atributo++;  
}  
...  
obj.atributo = 1;  
modificar(obj);      depois da invocação, obj.atributo é igual a 2
```

Este tipo de programação é uma fraca abordagem à programação centrada em objectos. Cada método deverá modificar somente o estado do objecto da invocação. Assim, no exemplo anterior, o método deveria ser incluído na classe C:

```
class C {  
    ...  
    void modificar() {  
        atributo++;  
    }  
}  
...  
obj.modificar();
```

Igualmente, se um método modifica dois ou mais objectos então deve ser dividido em dois ou mais métodos para que cada objecto altere o seu estado. Sempre que possível, cada método deve ser desenhado para realizar uma única tarefa, sendo designado por **método coeso** (do inglês, *cohesive method*).

No C e no C++ é possível passar qualquer valor, vector ou objecto por valor ou por referência. Uma estrutura muito extensa pode necessitar de muitos recursos para ser passada por valor (há uma duplicação dos objectos parâmetros: se o original já gasta bastante memória, a cópia vai gastar outro tanto). No Java não existe esse problema porque o passado por valor é apenas a referência do objecto, nunca o próprio objecto.

## Objectos mutáveis e imutáveis

Na secção anterior foi referido que a execução de certos métodos alteram o objecto alvo da invocação. Quando isso ocorre diz-se que esses objectos são **mutáveis** (do inglês, *mutable objects*). Existem classes onde os seus objectos são **imutáveis**. A invocação de métodos sobre um objecto imutável não altera o seu estado inicial. Em vez disso, é devolvido um novo objecto com as devidas alterações (que por sua vez é imutável e não pode ser mais alterado).

Qual a vantagem? A gestão do mecanismo de *aliasing* é complicado e com tipos imutáveis esse problema não ocorre. Não é necessário gerir quais as variáveis que devem reflectir as

modificações dos objectos, dado que objectos imutáveis nunca mudam. Igualmente, a implementação de tipos imutáveis pode ser realizada de forma mais eficiente que a dos tipos mutáveis (ver página 94 sobre uma classe de objectos imutáveis, a classe `String`).

## Sobreregar Métodos

Não podem existir métodos na mesma classe com a mesma assinatura. Não quer dizer que não existam dois métodos com o mesmo nome. Basta que as listas de parâmetros sejam diferentes para que as assinaturas serem diferentes. Esta capacidade denomina-se por **sobreregar** métodos (do inglês, *method overloading*). Os métodos seguintes poderiam pertencer à mesma classe:

```
public class Carro {
    ...
    public void arrancar() {
        velocidade = 10;
    }
    public void arrancar(int velInicial) {
        velocidade = velInicial;
    }
}
```

## Destrução de Objectos (e Vectores)

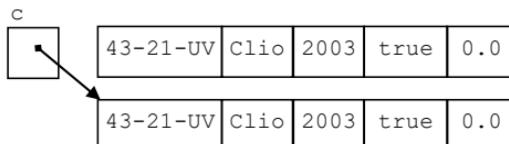
Depois da criação de um objecto (ou vector) a memória disponível diminui. Ao criar objectos sem nunca os destruir, i.e., sem libertar a memória reservada na sua criação, corre-se o risco de ficar sem memória disponível para criar novos objectos comprometendo a execução do programa<sup>7</sup>. Por exemplo, se não houvesse forma de libertar memória, a próxima instrução esgotaria rapidamente a memória disponível:

```
while (true)
    c = new Carro("43-21-UV", "Clio", 2003, true);
```

O que acontece? A cada iteração, a referência `c` é usada para armazenar o endereço de memória de um novo objecto. Ao fim de um certo número de iterações (milhares, milhões, depende da memória inicial) surgiria um erro associado à falta de memória. Mas observemos melhor. Na primeira iteração, a referência `c` armazena o endereço do primeiro objecto criado pelo operador `new`. Na segunda iteração `c` armazena o endereço do segundo objecto criado. Mas quem referência agora o endereço do primeiro objecto?

---

<sup>7</sup> Resumindo, reserva-se memória para criar objectos, destrói-se objectos para libertar memória.

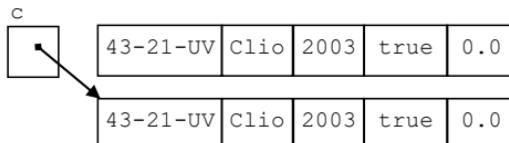
*Depois da 1<sup>a</sup> iteração**Depois da 2<sup>a</sup> iteração*

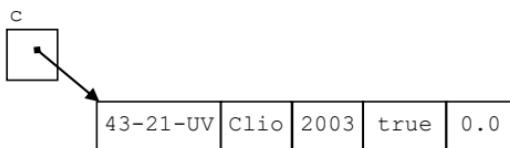
Depois da 2<sup>a</sup> iteração é impossível aceder ao primeiro objecto. Não existe uma referência que contenha o endereço de memória do primeiro objecto. A única forma de acesso era através da referência `c`.

Este é um problema sério em linguagens como o C ou o C++, porque o programador tem de se preocupar em destruir o primeiro objecto antes de usar `c` para referenciar outro objecto. Usando a linguagem Java, este trabalho de gestão do espaço disponível é realizado automaticamente. A este processo designa-se **recolha de lixo** (do inglês, *garbage collection*), ou seja, a recolha das referências perdidas.

Esta recolha tem vantagens e desvantagens. A desvantagem é que quando o processo é activado (libertando memória não referenciada) pode atrasar a execução do programa – existe uma penalização em termos de desempenho. Porém, a vantagem é que ao libertar o programador desta tarefa, facilita a programação e diminui o número de erros relacionados aumentando, assim, a robustez do programa.

Voltando ao exemplo anterior:

*Depois da 1<sup>a</sup> iteração**Depois da 2<sup>a</sup> iteração*



*Depois da recolha de lixo*

Em linguagens em que este mecanismo não existe é necessário explicitar a libertação de memória. Por exemplo, no C e no C++ existem operadores inversos ao operador `new` para libertar a memória dinamicamente reservada. Segue um exemplo em C:

```
#include <stdlib.h>      /* biblioteca do malloc, free */
int *p;
p = (int*)malloc(sizeof(int)); /* reserva memória */
...
free(p);                  /* liberta essa memória */
```

O C++, uma linguagem de programação centrada em objectos, possui um método especial designado destrutor. Ao inverso do construtor que é executado quando o objecto é criado, o destrutor é executado quando o objecto recebe a ordem de ser destruído. Esta linguagem possui o comando de remoção de objectos `delete` com a função inversa do comando `new`.

No Java existe um método que efectua uma função semelhante a este destrutor: o método `finalize()`. Este método não tem argumentos e devolve `void`. A diferença fundamental em relação ao destrutor do C++ é que não existe garantia alguma do momento em que o método `finalize()` possa ser invocado. Apenas é garantida a invocação *depois* do objecto perder todas as referências e *antes* da execução da recolha de lixo. Ou seja, o método pode nunca ser activado se a recolha de lixo não for activada. Não se garante qual a ordem da execução no caso de existirem vários objectos a destruir<sup>8</sup>. É conveniente que este método trate apenas questões relacionadas à memória, como fechar ficheiros ou apagar referências de objectos para que possam ser eliminados na futura recolha de lixo. Este tipo de funcionalidade é especialmente útil quando se interage com um programa de outra linguagem, para o qual a recolha de lixo não é automática e deve ser explicitamente referenciada.

A recolha de lixo pode ser forçada através do comando `System.gc()`. Como a recolha de lixo força as invocações dos métodos `finalize()` dos objectos a serem apagados, pode-se, assim, forçar a execução destes métodos.

---

<sup>8</sup> Curiosamente, a execução do método `finalize()` pode criar uma nova referência para o objecto (por exemplo, guardando a referência do próprio objecto num atributo da classe).

```
public class C {  
    public void finalize() {  
        System.out.print("a libertar recursos...");  
    }  
}  
...  
C obj = new C();  
obj = null;      // o objecto referenciado perdeu-se  
System.gc();  
□ a libertar recursos...
```

## Referências Finais

É possível usar o modificador `final` em referências (objectos e vectores). Porém, se nas variáveis é o valor armazenado que é constante, nas referências é a referência ao objecto que é constante. O objecto pode ser modificado, mas a referência não.

`final C obj;`  
 `obj = new C();` // fixa-se nesta inicialização.  
 `obj.atributo = 3;`  
 `obj = new C();`

Tanto nos tipos primitivos como nas referências, o valor final não precisa ser conhecido no momento da compilação. O que se garante é que depois da inicialização, o valor armazenado numa variável final não é alterado. Este mecanismo permite uma maior flexibilidade, pois alguns valores constantes podem depender da execução do programa.

## Métodos com parâmetros variáveis

Desde a versão Java 5 que também é possível criar métodos com um número variável de parâmetros. Nos parâmetros do cabeçalho da função define-se um tipo T seguido de reticências e de um nome X. Para cada invocação, o compilador transforma o número variável de argumentos num vector de tipo T com nome X. Um exemplo desta funcionalidade:

```
public void m(String ... args) {  
    for (String e : args)      // args é um vector de Strings  
        System.out.print(e);  
}  
...  
m("olá", " ", "mundo", "!");  
□ olá mundo!
```

Esta invocação é equivalente a: `m(new String[] {"olá", " ", "mundo", "!"});`

## Componentes de Classe

Sempre que falámos de atributos referimos que o seu tipo faz parte da configuração da classe e para um dado objecto dessa classe, o valor desse atributo faz parte do estado do objecto. Também se referiu que um método é invocado explicitando qual o objecto alvo da invocação.

É possível definir componentes que dizem respeito à classe e não aos objectos da classe. São denominados **atributos de classe** (do inglês, *class fields, static fields*) e **métodos de classe**, (do inglês, *class methods, static methods*). A necessidade de um atributo de classe surge quando há algo que não diz respeito a um objecto em particular mas sim à classe em geral. No nosso exemplo, a eventual informação sobre quantos carros existem diz mais respeito à classe Carro do que aos objectos da classe. Para definir um componente de classe prefixamos o atributo ou método com a palavra `static`:

```
public class Carro {
    public static int nCarros = 0;
    ...
    public Carro(...) {
        ...
        nCarros++;
    }
    ...
}
```

O período de vida de um atributo de classe corresponde ao tempo de execução do programa que utiliza a respectiva classe.

Um atributo de classe pode ser acedido a partir do nome da própria classe, `Carro.nCarros`, ou através de um objecto da classe, `umCarro.nCarros`. Não se aconselha a segunda opção porque dá a falsa impressão de `nCarros` não ser um atributo de classe.

A inicialização de um atributo de classe deve ocorrer na sua declaração (não nos construtores, porque estes referem-se a objectos) ou numa instrução de bloco de classe. No exemplo seguinte é declarado um atributo de classe `vector` inicializado pelo bloco de classe (executado uma única vez no início do programa).

```
public class C {
    static int[] vector;
    static {
        vector = new int[100];
        for (int i=0;i<100;i++)
            vector[i] = i*i;
    }
    ...
}
```

Um bloco não precisa ser de classe. Se não for, é executado antes do construtor para cada novo objecto. Por exemplo:

```
public class Carro {  
    public static int nCarros = 0;  
    {  
        nCarros++;  
    }  
    ...  
}
```

Para além dos atributos de classe existem métodos de classe que se referem a tarefas específicas da classe. Dentro de um método de classe apenas são acessíveis atributos de classe e outros métodos de classe. Como não existe um objecto em concreto não se pode utilizar a referência `this`.

```
public class Carro {  
    ...  
    public static int nCarros = 0;  
    public static decrementar() {  
        nCarros--;           para invocar antes de um objecto ser destruído  
    }  
    ...  
}
```

## Classes Interiores★

Uma **classe interior** (do inglês, *inner class*) pode ser definida dentro de uma outra classe. Uma instância de uma classe interior é sempre associada a uma instância da classe exterior. Um objecto do tipo da classe interior tem acesso à configuração e ao comportamento da classe onde é definida.

No seguinte exemplo, a classe `Carro` foi expandida de modo a incluir a última acção realizada (travagem ou aceleração e a que velocidade). Essa informação fica armazenada numa instância de classe interior `Accao`.

```
public class Carro {  
    ...  
    private Accao ultimaAccao;
```

A referência para o objecto da classe interior. Segue a definição da classe interior:

```
public class Accao {  
    private String accao;  
    private double velocidade;
```

```
public Accao(String s, double v) {  
    accao = s;  
    velocidade = v;  
}  
  
public String toString() {  
    return "realizado uma" + accao +  
        " com velocidade " + velocidade;  
}  
} // endInnerClass Accao  
...
```

Nos dois métodos seguintes são criados novos objectos da classe interior que armazenam a informação relevante à última acção realizada pelo objecto carro.

```
public void acelerar() {  
    ultimaAccao = new Accao("aceleração",velocidade);  
    velocidade *= 1.1;  
}  
  
public void travar() {  
    double v = bonsTravoes ? 10 : 20;  
    ultimaAccao = new Accao("travagem",velocidade);  
    if (velocidade/v>=5)  
        velocidade /= 1.5;  
    else  
        velocidade /= 1.1;  
}  
} // endClass Carro
```

Serão apresentados exemplos do uso das classes interiores na secção 12.2 que apresenta uma implementação para máquinas de estados e na secção 16.2 que refere o mecanismo de iteração de estruturas de dados.

Uma **classe local** (do inglês, *local inner class*) é definida dentro de um método. Ela é inacessível para lá do contexto do bloco (como uma variável local) mas é possível criar instâncias da classe, passá-las como argumentos ou devolvê-las como resultados de métodos (as instâncias continuam a existir enquanto forem referenciadas). O único modificador que se pode aplicar é `final` (indicando nesse caso que a classe local não pode ser estendida). As instâncias da classe local têm acesso somente às variáveis locais e aos argumentos finais do método (e já tenham sido inicializados). A razão desta restrição é que uma instância da classe local pode ainda existir quando o método (onde a classe interior local foi definida) terminar a sua execução. Se não se restringisse o contexto aos valores finais, já não seria possível aceder a esses valores. Assim, o contexto do objecto local, no

momento da criação é “congelado” (dado que os valores exteriores guardados são constantes) para posterior acesso.

## 4.2 Classes do Java

Entre as múltiplas classes das bibliotecas Java apresentamos algumas das mais usadas.

### A Classe “String”

Um objecto desta classe armazena uma frase (i.e., uma sequência de caracteres). A possibilidade de comunicação entre programas e utilizadores humanos é dos factores mais comuns nas linguagens de programação. Os desenhistas do Java resolveram tratar esta noção de frase com uma classe própria com a possibilidade de exprimir um objecto desta classe como um literal. Basta incluir a frase desejada entre aspas duplas. Por exemplo:

```
String s = "Olá";
```

Esta declaração faz o mesmo que as instruções:

```
char[] c = { 'O', 'l', 'á' };  
String s = new String(c);
```

Não confundir as aspas simples (literais do tipo primitivo `char`) com as aspas duplas (literais da classe `String`).

É igualmente possível utilizar caracteres especiais como os descritos na tabela 1.3.

```
String s = "Esta \"frase\"\n mudou de linha";
```

A classe ainda tem a particularidade de possuir um operador de concatenação, `+`, entre frases. Para literais, a concatenação é executada durante o processo de compilação (não existe uma penalização no desempenho do programa).

```
String linha = "Esta frase é demasiado comprida" +  
    "para uma só linha de código";
```

Alguns dos métodos da classe `String`:

- `char charAt(int i)` – devolve o carácter no índice *i* do objecto (como nos vectores, o primeiro carácter encontra-se no índice zero).

```
String s = "abcdefg";  
System.out.print(s.charAt(0));  
 a
```

- int *length()* – devolve o número de caracteres.

```
String s = "abcdefg";
System.out.print(s.length());
□ 7
```

- boolean *equals(Object s)* – devolve true se *s* é uma string igual à frase em questão, caso contrário devolve false.

```
String s = "abcdefg";
String t = "abcdefg";
System.out.print(s.equals(t));
□ true
```

- int *compareTo(String s)* – compara lexicograficamente os conteúdos de *s* e do objecto (qual delas aparece primeiro no dicionário). Devolve negativo se *s* aparecesse depois, devolve zero se forem iguais, ou devolve positivo caso contrário.

```
String s = "abcdefg";
String t = "abcd";
System.out.print(s.compareTo(t) + " ");
System.out.print(t.compareTo(s));
□ 3 -3
```

- String *substring(int inicio, int fim)* – devolve uma subfrase do objecto começando no índice *inicio* até o índice *fim* exclusive.

```
String s = "abcdefg";
System.out.print(s.substring(2,5));
□ cde
```

- String *replace(char antigo, char novo)* – devolve uma nova frase, substituindo todas as ocorrências do carácter *antigo* pelo carácter *novo*.

```
String s = "abcdefg";
System.out.print(s.replace('d', 'X'));
□ abcXefg
```

- String *trim(String s)* – devolve uma frase que se obtém depois de remover os eventuais espaços do princípio e do fim de *s*.

```
String s = "      abcdefg      ";
System.out.print("|"+s.trim()+"|");
□ |abcdefg|
```

- static String *valueOf*(int *i*) – este método de classe recebe um inteiro *i* e devolve uma frase com esse conteúdo. Existem métodos semelhantes para os outros tipos primitivos, incluindo vectores de caracteres. No exemplo seguinte, a variável *s* armazena a String “12”.

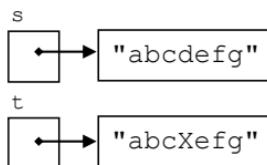
```
String s = String.valueOf(12);
System.out.print(s);
```

□ 12

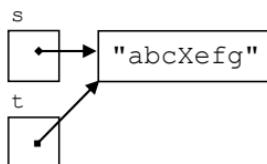
Existe um método especial denominado *toString()* invocado implicitamente para imprimir objectos. Quando se espera uma String e encontra-se a referência ao objecto *obj* de um outro tipo, a referência é substituída pela invocação *obj.toString()*.

Os objectos da classe String são imutáveis: não podem ser modificados depois da inicialização. Por exemplo, a execução das seguintes linhas produzem o efeito descrito pelo diagrama.

```
String s = "abcdefg";
String t = s.replace('d', 'X');
```



Se as strings fossem objectos mutáveis (que não são) a execução anterior produziria algo como:



Ou seja, em cada operação é criado um novo objecto. No exemplo seguinte são três os objectos criados (para além dos dois literais).

```
String t = "";
t += "olá ";
t += "mundo!";
```

Para modificar frases em tempo de execução deve-se utilizar a classe `StringBuffer`. Os objectos desta classe são mutáveis, o que torna certas tarefas, como concatenar duas frases, mais eficientes. O exemplo anterior seria descrito da forma seguinte (sendo necessário apenas um objecto):

```
StringBuffer t = new StringBuffer();
t.append("Olá ");
t.append("mundo!");
```

## A Classe “Scanner”

Esta classe surge com a versão Java 5 e serve para a leitura e processamento de dados, em especial, a leitura de dados do teclado.

No exemplo seguinte, é criado um objecto desta classe sendo utilizado para ler um inteiro.

```
Scanner sc = new Scanner(System.in);
int i = sc.nextInt();
```

O objecto `System.in` representa proveniente do utilizador pelo teclado (este objecto e outros semelhantes designam-se por fluxos. Falaremos sobre fluxos no capítulo 7).

Existe métodos semelhantes para recolher outros tipos primitivos. No exemplo seguinte, lê-se do teclado um valor do tipo `double`.

```
double x = sc.nextDouble();
System.out.println(x*2);
```

No caso da leitura de números com casas decimais é necessário ter atenção às convenções regionais que o computador utiliza. Por exemplo, há países onde um e meio se escreve 1,5 enquanto outros representa-se por 1.5.

É possível ler informação de uma *string* como se fosse do teclado:

```
String frase = "1 separador 4";
Scanner sc = new Scanner(frase);
System.out.print(sc.nextInt());
sc.next();
System.out.print(sc.nextInt());
sc.close();
```

```
□ 1 4
```

O método `next()` recolhe uma *string* e método `close()` termina o processamento do respectivo objecto.

É igualmente possível definir quais os símbolos considerados separadores para facilitar a recolha de informação (através do método `useDelimiter()`). No exemplo seguinte, o dois pontos é definido como separador na recolha de dados na *string*:

```
String frase = "ontem:hoje:amanhã";
Scanner sc = new Scanner(frase).useDelimiter(":");
System.out.print(sc.next() + " e ");
System.out.print(sc.next());
sc.close();
□ ontem e hoje
```

Consultar a documentação Java sobre a classe `Scanner` e a classe `Pattern` para mais informação.

## As Classes Envolventes dos Tipos Primitivos

Como existem métodos de classe para *strings* é normal esperar que existam métodos para os tipos primitivos que lidem com inteiros, reais ou valores booleanos. Porém, há um problema: sendo os inteiros, reais e booleanos tipos primitivos e não classes, como armazenar e posteriormente obter acesso a esses eventuais métodos? Para responder a esse problema há classes especiais que reúnem esses métodos, designadas por **classes envolventes** (do inglês, *wrapper classes*):

- Classes `Byte`, `Short`, `Integer` e `Long` – contêm métodos que lidam com os valores inteiros.
- Classes `Float` e `Double` – contêm métodos que lidam com valores reais.
- Classe `Boolean` – contêm métodos que lidam com valores lógicos.
- Classe `Character` – contêm métodos que lidam com caracteres.

Alguns dos métodos comuns da classe `Integer` (os métodos são muito semelhantes para as outras classes inteiras):

- `static final int MAX_VALUE` – o maior número representado por um inteiro: 2147483647.
- `static final int MIN_VALUE` – o menor número representado por um inteiro: -2147483648.
- `static int parseInt(String s)` – converte a frase *s* num inteiro. Se *s* não for um número, a função lança a exceção `NumberFormatException`.
- `static int parseInt(String s, int base)` – converte a frase *s* na base numérica *base*, num inteiro. Se *s* não for um número, esta função lança a exceção `NumberFormatException`. Neste e nos próximos casos, os valores válidos para *base* ocorrem entre 2 e 36. Para representar dígitos depois de 9, usar as letras do alfabeto, A para representar 10, B para 11, ..., Z para 36.

```
int a = Integer.parseInt("100");      a igual a 100
int b = Integer.parseInt("100", 8);    b igual a 64
int c = Integer.parseInt("FF", 16);    c igual a 255
```

- static String *toString*(int *n*, int *base*) – converte o inteiro *n* da base numérica *base*, numa frase.

```
String s = Integer.toString(255,16); s igual a "ff"
```

- static Integer *valueOf*(String *n*, int *base*) – converte a frase *s* na base numérica *base*, num objecto da classe Integer. Se *s* não for um número esta função lança a excepção NumberFormatException.
- static double *intValue*() – converte o objecto num valor do tipo int. Existem ainda métodos equivalentes: *floatValue*, *doubleValue*, *byteValue*, *shortValue* e *longValue*.

```
String s = "12";
int i = Integer.valueOf(s).intValue(); i igual a 12
```

Alguns dos métodos comuns da classe Double:

- static final int *MAX\_VALUE* – o maior número representado por um objecto desta classe: 1.7976931348623157e308.
- static boolean *isInfinite*(double *d*) – devolve verdadeiro se o real *d* armazenar o valor especial de infinito.
- static boolean *isNaN*(double *d*) – devolve verdadeiro se o real *d* não for um número, i.e., armazenar o valor especial *Not a Number*.
- static int *parseDouble*(String *s*) – converte a frase *s* num valor double. Se *s* não for um número, lança a excepção NumberFormatException.
- static Double *valueOf*(String *n*) – converte a frase *s* num objecto da classe Double. Se *s* não for um número lança a excepção NumberFormatException.
- static String *toString*(double *n*) – converte o real *d* numa frase.
- static double *intValue*() – converte o objecto num valor do tipo double. Existem ainda métodos equivalentes: *doubleValue*, *floatValue*, *byteValue*, *shortValue* e *longValue*. Estes métodos podem produzir perdas de precisão.

Alguns dos métodos comuns da classe Boolean:

- static final boolean *TRUE* – o valor “verdadeiro” da classe Boolean.
- static final boolean *FALSE* – o valor “falso” da classe Boolean.
- static Boolean *valueOf*(String *n*) – converte a frase *s* num objecto da classe Boolean.
- static String *toString*() – converte o objecto numa frase.

Alguns métodos da classe Character:

- static final byte *LINE\_SEPARATOR* – o valor do separador de linhas. Existem dezenas de constantes para indicar diversos caracteres especiais.
- static String *toString*() – converte o objecto numa frase.

- static char *charValue()* – converte o objecto num valor do tipo char.
- static boolean *isDigit(char c)* – verifica se o carácter é um dígito. Existem métodos semelhantes para verificar se é uma letra, *isLetter*, se é maiúscula ou minúscula, *isLowerCase* e *isUpperCase*.

Para além de servirem de “morada” aos métodos que lidam com os tipos primitivos, estas classes são úteis quando se pretende utilizar um tipo primitivo como argumento num método que só recebe tipos de referência. Primeiro cria-se um objecto que guarde a informação a armazenar para parametrizar o método em questão:

```
void fazerAlgo(Object o) { ... };  
...  
Double aMinhaInformacao = new Double(2.718281828);  
fazerAlgo(aMinhaInformacao);
```

Desde a versão Java 5 que o compilador faz esta tradução automaticamente (em inglês, *autoboxing* e *autounboxing*), ou seja, traduzir um tipo primitivo para um objecto da respectiva classe envolvente ou vice-versa. O exemplo seguinte mostra este mecanismo em funcionamento:

```
Integer ob = 5;           // autoboxing  
int n = ob + 6;          // autounboxing  
System.out.print(n);  
□ 11
```

## A Classe “Math”

A classe Math contém métodos para cálculo de funções matemáticas comuns. Alguns dos componentes da classe Math:

- static final double *PI* – o valor aproximado de  $\pi \approx 3.141592653589793$ .
- static final double *E* – o valor aproximado de  $e \approx 2.718281828459045$ .
- static int *abs(int n)* – o valor absoluto de *n*, i.e.  $f(n)=|n|$ . Existem ainda métodos com o mesmo nome para argumentos double, long e float.
- static double *ceil(double x)* – arredonda para cima.
- static double *floor(double x)* – arredonda para baixo.

Math.ceil(32.1)	igual a 33.0
Math.ceil(-32.1)	igual a -32.0
Math.floor(32.6)	igual a 32.0
Math.floor(-32.6)	igual a -33.0

- static double *round(double x)* – arredonda para o número mais próximo.

Math.round(32.1)	igual a 32.0
Math.round(32.5)	igual a 33.0

```
Math.round(-32.1) igual a -32.0  
Math.round(-32.5) igual a -32.0  
Math.round(-32.51) igual a -33.0
```

- `static int max(int n, int m)` – a função máximo.
- `static int min(int n, int m)` – a função mínimo. Para o mínimo e máximo existem métodos com argumentos `double`, `long` e `float`.
- `static double exp(double x)` – a função exponencial,  $f(x) = e^x$ .
- `static double log(double x)` – a função logarítmica,  $f(x) = \log_e x$ .

```
Math.exp(3) igual a  $e^3 \approx 20.085536923187668$   
Math.log(20.0855) igual a 2.9999981617010416  
Math.log(32) / Math.log(2) igual a  $\log_2 32 = 5.0$ 
```

- `static double pow(double x, double y)` – a função potência,  $f(x,y) = x^y$ .
- `static double sqrt(double x)` – a função raiz quadrada,  $f(x) = \sqrt{x}$ .

Existem as funções trigonométricas `sin`, `cos`, `tan`, `asin`, `acos` e `atan`. Os métodos `toDegrees(double radianos)` e `toRadians(double graus)` permitem a conversão entre as duas medidas de ângulos (uma rotação completa de 360 graus é igual a  $2\pi$  radianos).

É possível gerar números pseudo-aleatórios através do método `random()` que devolve um valor `double` entre 0 e 1 com distribuição aproximadamente uniforme (há quem considere que este gerador não é bom, sendo útil apenas em situações informais onde a distribuição exacta dos números gerados não é crítica).

Qualquer variável real (independentemente da implementação) é sempre uma aproximação do verdadeiro valor real. Não existe uma restrição ao modo como os diversos sistemas operativos manipulam os números `float` e `double`. Logo, sistemas diferentes podem produzir resultados diferentes. Por exemplo, o valor `Math.sin(pi/5)` pode ter mais casas decimais no sistema operativo X que no sistema Y. Se a aplicação necessitar de valores estritamente idênticos em todas as máquinas não se deve utilizar a classe `Math` mas sim a classe `StrictMath` que possui os mesmos métodos mas obriga todas as execuções do programa a produzirem os mesmos resultados. Com igual objectivo, o modificador de classe e/ou de métodos, `strictfp`, força o uso de algoritmos padrão sobre os tipos primitivos reais, assegurando os mesmos resultados independentemente do computador e do sistema operativo usado.

## Algumas conversões comuns entre tipos

Antes de terminar o capítulo, mostram-se alguns exemplos para converter valores entre tipos diferentes. Para além do mecanismo de coerção referido no capítulo 1 dedicado aos tipos primitivos, não é imediato, para quem se inicia na programação em Java, converter valores mais complexos (por exemplo, entre uma *string* e um inteiro) tendo-se de utilizar alguns dos métodos disponíveis nas classes Java.

- Entre inteiros e *strings*:

```
String s = Integer.toString(100);      igual a "100"  
int    i = Integer.parseInt("100");     igual a 100
```

- Entre caracteres e inteiros (com o respectivo valor do código ASCII)

```
int   i = (int) '0';                  igual a 48, código ASCII do zero  
char  c = 48;                      igual a '0'
```

- Entre caracteres e *strings*:

```
String s = String.valueOf('c');        igual a "c"  
char   c = "1234".charAt(0);         igual a '1'
```

- Valores decimais para outras bases (por exemplo, binário e hexadecimal)

```
String s1 = Integer.toString(30, 2);    igual a 11110  
String s2 = Integer.toString(30, 16);   igual a 1e  
int   i1 = Integer.parseInt("11110", 2); igual a 30  
int   i2 = Integer.parseInt("1E", 16);   igual a 30
```

---

## Exercícios

1. Um triplo pitagórico ( $x,y,z$ ) verifica a igualdade  $x^2+y^2=z^2$ . Implemente um método que recebe três inteiros positivos e verifica se eles constituem um triplo pitagórico.
2. Implemente um método que recebe dois parâmetros, um vector de inteiros e um inteiro e verifique se o segundo parâmetro encontra-se no vector.
3. Implemente um método que recebe uma frase e um caracter e que devolva o número de vezes que esse caracter ocorre na frase.
4. Implemente um método que recebe uma frase que devolva a frase invertida. Por exemplo, se receber “*Portugal é um país Europeu*” deve devolver “*ueporuE síap um é lagutroP*”.
5. Implemente um método que recebe um valor inteiro (menor que mil) que devolva uma frase com esse valor descrito por extenso. Por exemplo, para o valor 453 o método deve devolver “*quatrocentos e cinquenta e três*”.

6. Implemente um método que recebe um real que representa uma avaliação de 0 a 20 com duas casas decimais e verifique se essa avaliação tem valor positivo e se é quase arredondada para cima (considere que uma nota é quase arredondada para cima, quando o valor decimal é maior ou igual a 0,43).
7. Implemente um método que recebe dois reais e devolva o logaritmo do primeiro parâmetro na base do segundo parâmetro.
8. Implemente um método que recebe um inteiro positivo e devolva o factorial desse inteiro.
9. Implemente um método que recebe dois inteiros positivos e devolva o maior divisor comum. (sugestão: utilize o algoritmo de Euclides que descreve o seguinte algoritmo,  $\text{mdc}(a,b) = \text{mdc}(b,a \% b)$  para  $b \neq 0$  e  $\text{mdc}(a,0) = a$ ).
10. Implemente um método que dado um objecto da classe `Date`, determine quantos dias já passaram até ao presente dia.
11. Um número primo é um número somente divisível pela unidade e por si mesmo. Implemente um método que recebe um inteiro positivo maior que um e verifica se este é primo. Comente a eficiência da sua solução.
12. Dois números são primos entre si, se o maior divisor comum entre eles for igual a 1. Implemente um método que recebe dois inteiros positivos e verifica se são primos entre si.
13. Um número perfeito é um número igual à soma dos seus factores (excepto o próprio número) . Implemente um método que recebe um inteiro positivo e verifica se este é perfeito. Utilize na sua resolução (e explique) o seguinte método que devolve o vector de factores de um número inteiro dado:

```
public static int[] factors(int n) { // n>1
    int[] fact = new int[n];
    int index = 0;
    for(int i=1;i<=n;i++)
        if (n%i==0)
            fact[index++] = i;
    // construir o vector final
    int[] result = new int[index];
    for(int i=0;i<index;i++)
        result[i] = fact[i];
    return result;
}
```

14. A expansão prima de um número positivo é a sequência única de expoentes naturais usados na multiplicação de números primos para obter esse número. Por exemplo, como  $20 = 2^2 \cdot 3^0 \cdot 5^1$ , a sua expansão prima é (2,0,1). Já  $172480 = 2^6 \cdot 3^0 \cdot 5^1 \cdot 7^2 \cdot 11^1$ , logo a sua expansão prima é (6,0,1,2,1). Implemente um método que, dado um número inteiro positivo, devolve um vector de inteiros com a sua expansão prima.

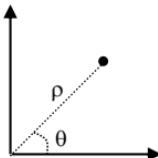
15. A expansão de Cantor de um número positivo é a sequência única de coeficientes naturais  $a_i$  usados no seguinte somatório de factoriais:  $a_n.n! + a_{n-1}.(n-1)! + \dots + a_2.2! + a_1.1!$ . Por exemplo, como  $12 = 2 \cdot 3^1$ , a expansão de Cantor é (2,0,0). Já  $743 = 6! + 3 \cdot 3! + 2 \cdot 2! + 1 \cdot 1!$ , logo a sua expansão de Cantor é (1,0,0,3,2,1). Implemente um método que, dado um número inteiro positivo, devolve um vector de inteiros com a sua expansão de Cantor.

16. Defina a classe `Primo` que inclui um construtor que recebe o índice do primo a que o objecto deve ser inicializado (e.g., `new Primo(1)` inicializa com o primeiro primo, i.e., o número 2) e os seguintes métodos: (i) `inc()` o objecto armazena o primo seguinte ao actual, (ii) `dec()` armazena o primo anterior (se existir), (iii) `valor()` devolve o primo actual, (iv) `indice()` devolve o índice do primo actual e (v) `primo(int i)` um método estático que devolve o  $i$ -ésimo primo.

17. Defina a classe `ContaTelefone` que inclui os seguintes métodos: (i) `verConta()` que devolve o valor actual da conta do telefone, (ii) `chamada(int segundos, double custoSegundo)` e (iii) `apagar()` que coloca a zero o valor actual da conta.

18. Defina uma classe que representa pontos em 2 dimensões e que utilize coordenadas cartesianas e implemente os métodos: construtores, modificação e acesso aos atributos da classe, distância ao centro, distância a outro ponto, determinar o quadrante onde o ponto se encontra.

19. Rescreva a classe anterior usando coordenadas polares como atributos.



20. Defina uma classe para representar pontos em 3 dimensões que implemente os métodos: construtores, modificação e acesso aos atributos da classe, distância ao centro, distância a outro ponto, rotação de  $x$  graus, verificar se dois pontos mais o próprio objecto são colineares (i.e., se os três formam uma linha), calcular a área do triângulo definido pelos três pontos.

21. Defina uma classe que representa segmentos de recta definidos por dois pontos. Implemente as operações comprimento, ponto médio, mover segmento e rodar segmento (em redor do ponto central do segmento). Adicione ainda um método que permita verificar a igualdade entre segmentos.
22. Defina uma classe para representar triângulos com operações para calcular a área, calcular o perímetro e para efectuar translações. A área é calculada segundo a formula  $\sqrt{s(s-a)(s-b)(s-c)}$  onde  $s$  é o semiperímetro e  $a, b$  e  $c$  os comprimentos das três arestas.
23. Defina uma classe que represente um número racional. Essa classe contém as operações: (i) consultar/modificar numerador e denominador; (ii) adicionar ao número, um outro número racional; (iii) adicionar ao número, um número inteiro; (iv) multiplicar o número por um outro número racional; (v) multiplicar o número por um número inteiro; (vi) verificar se o racional é igual a um outro número racional; (vii) verificar se o racional é inferior a outro número racional.
24. Defina uma classe que represente um número complexo. Essa classe contém as operações: (i) consultar/modificar os coeficientes da parte real e da parte imaginária; (ii) adicionar ao número, outro número complexo; (iii) subtrair ao número, outro número complexo; (iv) multiplicar o número por outro número complexo; (v) dividir o número por outro número complexo; (vi) devolver um novo número complexo igual ao corrente.
25. Defina uma classe que represente uma progressão aritmética. Essa classe contém as operações: (i) consultar/modificar o primeiro elemento e o incremento da progressão; (ii) determinar o  $i$ -ésimo elemento da progressão; (iii) devolver o valor da soma dos primeiros  $i$  elementos da progressão; (iv) determinar o valor do produto dos primeiros  $i$  elementos; (v) verificar se a progressão é crescente; (vi) determinar o índice do menor elemento da progressão superior a um dado valor (assumindo que a progressão é crescente); (vii) saber se a progressão é igual a outra progressão aritmética.
26. Rescrever o exercício anterior para progressões geométricas.
27. Defina uma classe que represente um polinómio. Esta classe contém as operações: (i) determinar o grau do polinómio; (ii) determinar o coeficiente do termo de um determinado grau; (iii) estabelecer o coeficiente do termo de um determinado grau; (iv) adicionar ao polinómio um outro polinómio; (v) multiplicar o polinómio por outro polinómio; (vi) dividir o polinómio por um polinómio da forma  $x-\alpha$  (aplicar a regra de *Ruffini*; a divisão só deverá ser realizada se a divisão der resto 0).
28. Defina uma classe que represente uma distribuição de números naturais. Esta classe contém as operações: (i) acrescentar valores à distribuição; (ii) saber quantas ocorrências existem de um dado valor; (iii) saber a média da distribuição; (iii) saber a mediana da distribuição; (iv) saber a moda da distribuição. Conferir o exercício 15 do capítulo 3.

29. Declare uma classe para cálculo de grandes factoriais. Esta classe inclui um método que recebe um valor e devolve uma frase com o valor desse factorial. Por exemplo: a invocação do método `obj факт(50)` devolve “`3044093201713378043612608166064768844377641568960512000000000000`”.
30. Construa uma classe que represente uma sequência de dígitos (0 a 9). Toda a instância da classe, quando criada, representa uma sequência vazia. Sobre qualquer instância da classe deverá ser possível: (i) acrescentar um dígito à direita da sequência; (ii) acrescentar um dígito à esquerda da sequência; (iii) concatenar à sequência uma outra sequência; (iv) saber qual é o  $i$ -ésimo elemento da sequência; (v) saber o comprimento da sequência; (vi) verificar se uma sequência é prefixo da sequência; (vii) verificar se uma sequência é sufixo da sequência; (viii) verificar se uma sequência é subsequência da sequência; (ix) saber se a sequência é igual a outra sequência; (x) devolver uma nova sequência igual à sequência corrente.
31. Defina uma classe que represente um cofre. A classe deve conter as operações: (i) iniciar a fechadura com um vector de números, (ii) determinar se o cofre está aberto ou fechado, (iii) introduzir uma frase se o cofre está aberto, (iv) recolher a frase se o cofre está aberto (ou “” se está fechado) (v) inserir o próximo número da combinação, (vi) reiniciar a fechadura (esquecendo os números da combinação já inseridos).
32. Defina uma classe que descreva o sistema de gestão duma praça de portagens, sabendo que uma instância da mesma deve ser gerada a partir do número de portagens existentes na praça. A classe deve conter as operações: (i) registar a passagem de um veículo por uma dada portagem; (ii) saber quantos veículos já passaram numa determinada portagem; (iii) saber qual a portagem mais concorrida; (iv) quantos veículos já passaram pela praça.
33. Defina uma classe geradora de números pseudo-aleatórios entre  $[0,1]$ . A classe deve conter as operações: (i) iniciar o objecto com uma semente inteira (i.e., um valor que inicia a sequência de valores), (ii) ir buscar o próximo número da sequência com distribuição uniforme, (ii) ir buscar o próximo número da sequência com distribuição Gaussiana. Procure na literatura um método adequado para gerar estas duas sequências de números pseudo-aleatórios.

# 5 – Recursão

---

O processo de iteração (visto no capítulo 2) é um dos mecanismos essenciais para a construção de programas. Esta iteração – a capacidade para executar uma tarefa zero ou mais vezes – é disponibilizada pelas instruções `while`, `do-while` e `for`. A iteração é um caso particular do conceito de **recursão**. A recursão é, no contexto da programação Java, a capacidade de um método se invocar a si próprio designando-se por **método recursivo**.

O uso da recursão é útil na resolução de problemas que podem ser decompostos em subproblemas mais simples e onde a solução final obtém-se pela composição das soluções desses subproblemas. Para aplicar soluções recursivas é necessário: (i) existir um conjunto de um ou mais casos triviais que não precisam de nenhuma computação suplementar – denominado **caso base** ou **base da recursão**, (ii) ser possível decompor o problema em subproblemas no qual pode ser construída a solução final – denomina-se esta decomposição por **passo da recursão** e (iii) os subproblemas chegam inevitavelmente ao caso base.

A estrutura de um programa recursivo é a seguinte:

```
problema(P) :  
    se o caso base B responde a P  
        devolver B  
    senão  
        decompor P em P1, ..., Pn  
        R1 = problema(P1)  
        ...
```

```
Rn = problema(Pn)
R = construir resposta com R1, ..., Rn
    devolver R
```

A esta técnica de decomposição de um problema em subproblemas relacionados também se designa por **dividir para conquistar** (do inglês, *divide and conquer*).

Os subproblemas têm de receber a informação necessária para a resolução e têm de devolver informação suficiente para a construção da resposta global. Se a decomposição dos subproblemas nunca chegar ao caso base, a recursão não tem forma de parar. É uma **recursão infinita** e o programa terminará eventualmente por falta de recursos de memória.

Considere o exemplo da função factorial. A função factorial pode ser descrita por: “a função que devolve o produtório de 1 até ao argumento inteiro dado”, i.e.,  $\text{factorial}(n) = 1 \times 2 \times \dots \times n$ , com  $\text{factorial}(0) = 1$ . Em Java:

```
public long factorial(long n) {
    int fac = 1;
    for(int i=1; i<=n; i++)
        fac *= i;
    return fac;
}
```

Esta é uma solução iterativa: o ciclo calcula o resultado através da variável `fac` iniciada a 1 (o elemento neutro do produto) e uma variável de progresso `i` de forma a iterar `n` vezes.

No entanto, a função factorial também pode ser apresentada de uma forma recursiva: “a função que devolve 1 quando o argumento é zero, caso contrário devolve o argumento multiplicado pelo factorial do antecessor”, i.e.,

$$\begin{aligned}\text{factorial}(n) &= n \times \text{factorial}(n-1) \\ \text{factorial}(0) &= 1\end{aligned}$$

Observamos que o problema geral é decomposto num caso mais simples –  $\text{factorial}(n-1)$  – e existe um caso base –  $\text{factorial}(0) = 1$ . Em Java:

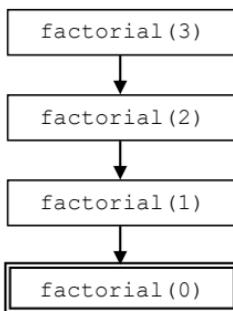
```
public long factorial(int n) {
    if (n==0) // a base da recursão
        return 1;
    return n*factorial(n-1); // o passo da recursão
}
```

Considere a seguinte definição da função factorial:

$$\text{factorial}(n) = \text{factorial}(n+1) / n$$

Qual o problema que encontramos? Os sucessivos subproblemas não convergem para uma base da recursão, tornando-se uma recursão infinita.

Uma forma de estudar um método recursivo é analisar a sua árvore de recursão. A **árvore de recursão** é uma estrutura onde se apresenta todas as invocações recursivas do método em questão. O número de elementos da árvore define o tempo de execução do método (cada elemento da árvore corresponde a uma invocação). A altura máxima da árvore determina os requisitos de memória (é igual ao número máximo de invocações à espera de resultado). A árvore de recursão para a invocação `factorial(3)` é (as caixas com linha dupla identificam os casos base):



## 5.1 Recursão vs. Iteração

Surge um potencial dilema: qual o tipo de solução (recursiva ou iterativa) que se deve escolher? A recursão é especialmente útil na resolução de problemas para os quais não existe uma solução iterativa óbvia, já que estas tendem a ser mais eficientes que as soluções recursivas. Porquê? As funções recursivas precisam de maiores recursos pois as múltiplas invocações têm de ser armazenadas e geridas em memória (o que tem um custo associado). Por outro lado, a solução recursiva tende a ser mais fácil de interpretar e codificar. Apesar de potencialmente mais lenta, verificar a sua correcção é mais acessível. A escolha de um método iterativo ou recursivo não é trivial...

Um exemplo deste problema é dado pela sequência de *Fibonacci*. Leonardo de Pisa (igualmente conhecido por *Fibonacci*) foi um matemático italiano que escreveu – em 1202 – um estudo sobre a explosão demográfica de coelhos que ocorreria numa situação idealizada.

O problema é descrito da seguinte forma:

*Cada casal de coelhos com pelo menos um mês de idade, ao fim de cada mês, acasala e produz um novo casal de coelhos. Nenhum coelho morre. Suponha a existência inicial de um casal de coelhos recém nascidos. Quantos casais de coelhos existem ao fim de um ano?*

No primeiro mês há um casal, no segundo ainda há um casal mas com idade para acasalar, no terceiro há dois casais (o casal mais velho mais o casal recém nascido), etc. Os primeiros números desta sequência são: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ... Rapidamente se verifica uma relação nesta sequência crescente: cada número é a soma dos dois anteriores, exceptuando os dois primeiros ambos iguais a 1. Esta descrição dá-nos imediatamente as bases e o passo da recursão:

$$\begin{aligned} \text{fib}(0) &= 1 \\ \text{fib}(1) &= 1 \\ \text{fib}(n) &= \text{fib}(n-1) + \text{fib}(n-2) \end{aligned}$$

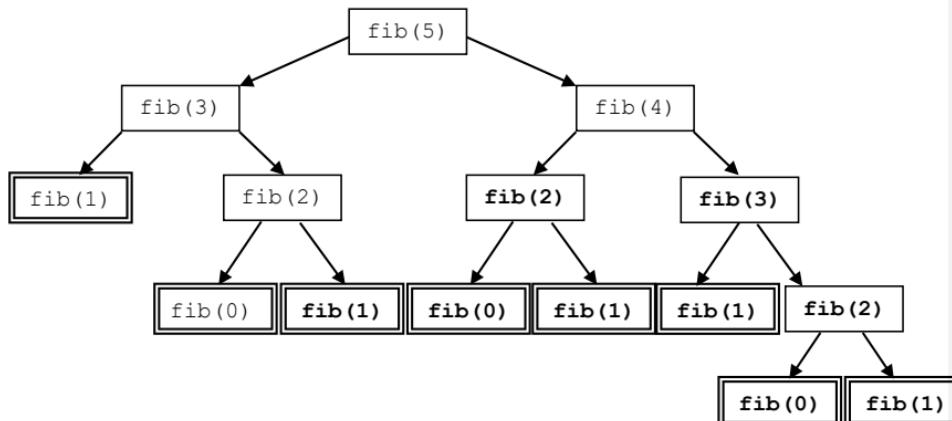
```

public long fib(int n) {
    if (n<2)
        return 1;
    return fib(n-1) + fib(n-2);
}

```

Mas, este método tem um problema grave. Quando num método existe apenas uma invocação a si próprio (como no caso do factorial) diz-se uma **recursão linear**. Quando o método invoca-se duas ou mais vezes (como no caso da função de *Fibonacci*) trata-se de uma **recursão não-linear**. Se uma recursão não-linear invocar um método, com os mesmos argumentos, mais do que uma vez, a resolução do problema pode ser ineficiente.

Considere a árvore de recursão da invocação do método de *Fibonacci* com valor inicial 5. As caixas com texto carregado indicam onde a computação foi repetida. Das dezasseis invocações, nove foram repetições. Para números maiores, esta proporção aumenta de tal modo que o cálculo de números de *Fibonacci* rapidamente se torna inviável.



Como resolver este problema? Ou opta-se por uma solução iterativa (se houver uma solução iterativa simples, o que nem sempre existe):

```
public long fib(int n) {
    int a=1, b=1;
    for(int i=2; i<=n; i++) {
        int temp = b;
        b += a;
        a = temp;
    }
    return b;
}
```

Ou reformula-se o problema de forma a obter uma recursão linear:

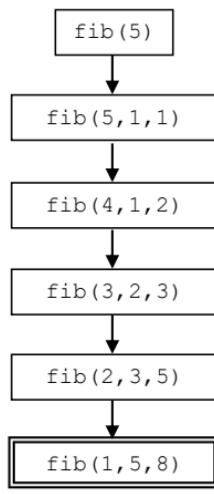
```
public long fib(int n) {
    return fib(n,1,1);
}
private long fib(int n, long a, long b) {
    if (n<2)
        return b;
    return fib(n-1,b,a+b);
}
```

Observando a nova árvore (à direita) de recursão das invocações `fib(5)` nota-se de imediato as diferenças. Esta é uma técnica muito poderosa: quando se pretende simular um ciclo através de uma recursão, utilizam-se argumentos extra na invocação recursiva para transportar os valores necessários à execução ( neste caso, os argumentos `a` e `b`).

Este exemplo mostra ser muito importante escolher uma representação recursiva adequada. Essa escolha faz a diferença entre uma solução recursiva eficiente e outra extremamente inefficiente.

Entre os problemas com uma solução recursiva há aqueles:

- para os quais existe uma solução iterativa simples e, por isso, a solução recursiva não deve ser escolhida (como o somatório de elementos de um vector, ou o cálculo do factorial)
- em que a solução iterativa não sendo trivial, os dois tipos de implementação são igualmente válidos (como a procura binária referida em 17.2).



- onde a complexidade da solução iterativa é consideravelmente maior, não compensando o eventual ganho de desempenho (como o algoritmo de ordenação *quicksort* descrito no capítulo 19).

## 5.2 Recursão Terminal

Se um método termina com uma invocação recursiva (i.e., não existem mais instruções depois da invocação) diz-se que o método tem uma invocação **recursiva terminal** (do inglês, *tail recursive call*).

Todas as invocações recursivas são executadas através do uso de uma pilha (ver capítulo 15). Esta pilha armazena as múltiplas invocações do método que não terminaram juntamente com as respectivas variáveis locais. Se o método recursivo é a última instrução a ser executada na  $n$ -ésima invocação do método, não há necessidade de manter as variáveis locais dessa  $n$ -ésima invocação (o âmbito onde eram conhecidas terminou). Desse ponto de vista, poder-se-ia eliminar da pilha a  $n$ -ésima invocação do método e substituí-la directamente pela  $(n+1)$ -ésima invocação.

Este procedimento fornece pistas para uma simplificação. Uma chamada recursiva terminal é eliminada se trocarmos os argumentos da invocação actual do método pelos argumentos da chamada recursiva terminal e voltarmos a repetir essas instruções. O exemplo seguinte possui uma recursão terminal:

```
int metodo(int n) {
    if (n<=0)           // a base da recursão
        return 1;
    return n*metodo(n/2); // o passo da recursão
}
```

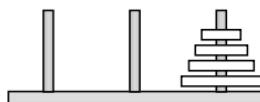
Usando a ideia do parágrafo anterior:

```
int metodo2(int n) {
    int temp = 1;
    while (n>0) {
        temp *= n;           // era o passo da recursão
        n /= 2;              // idem
    }
    return temp;            // era a base da recursão
}
```

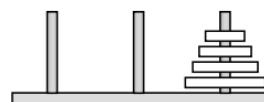
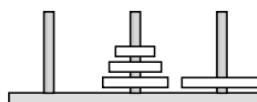
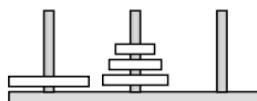
Algumas linguagens (como o Prolog ou o Haskell) detectam a recursão terminal e realizam automaticamente este tipo de optimização.

Segue outro exemplo clássico de recursão. Veremos como optimizá-lo segundo este processo. As Torres de Hanói é o nome de um *puzzle* em que existem três colunas (sejam A, B e C) onde se colocam discos de diversos tamanhos. Inicialmente, uma torre

constituída por N discos está na coluna A. Em cada turno, pode-se deslocar um dos discos que está no topo de uma das colunas para uma outra coluna desde que não se coloque por cima de um disco menor. O *puzzle* consiste em descobrir como se pode deslocar a torre inteira em A para a coluna C no menor número de passos.



Este problema é interessante porque possui uma solução recursiva relativamente simples. O raciocínio é o seguinte: deslocar uma torre de dimensão N da coluna origem para a coluna destino usando a coluna intermédia como ajuda, é o mesmo que: (i) deslocar a torre de dimensão N-1 da origem para a coluna intermédia (usando a coluna destino como ajuda), (ii) deslocar o disco maior da origem para o destino e (iii) deslocar a torre de dimensão N-1 da coluna intermédia para o destino (usando a coluna origem como ajuda).



Temos, assim, o passo da recursão. E o caso base? Uma torre de dimensão 1 (i.e., um disco) desloca-se directamente da coluna origem para a coluna destino.

```
void hanoi(int n, char origem, char destino,
           char intermedia) {
    if (n==1)      // a base da recursão
        System.out.println("Disco 1: " + origem + "->" +
                             + destino);
    else {         // o passo da recursão
        hanoi(n-1, origem, intermedia, destino);
        System.out.println("Disco " + n + ": " + origem
                           + "->" + destino);
        hanoi(n-1, intermedia, destino, origem);
    }
}
```

A invocação de `hanoi(3, 'A', 'C', 'B')` produz:

```
□ Disco 1: A->C ←  
  Disco 2: A->B ←  
  Disco 1: C->B ←  
  Disco 3: A->C ←  
  Disco 1: B->A ←  
  Disco 2: B->C ←  
  Disco 1: A->C ←
```

No método `hanoi()` existe uma invocação recursiva terminal. Ao eliminá-la, usando o processo já referido, obtemos o método `hanoi2()` mais eficiente. Uma das duas invocações recursivas do método `hanoi()` é eliminada.

```
void hanoi2(int n, char origem, char destino,  
            char intermedia) {  
  
    while (n > 1) {  
        hanoi2(n-1, origem, intermedia, destino);  
        System.out.println("Disco " + n + ": " + origem  
                           + "->" + destino);  
        n--;  
        char temp = origem;  
        origem = intermedia;  
        intermedia = temp;  
    }  
    System.out.println("Disco 1: " + origem + "->"  
                      + destino);  
}
```

## 5.3 Resolução de Problemas por Retrocesso\*

Encontra-se sempre a melhor solução se for possível percorrer recursivamente todas as hipóteses possíveis. Este tipo de estratégia designa-se por **retrocesso** (do inglês, *backtracking*). O retrocesso ao pesquisar todas as hipóteses acaba por encontrar uma solução (se esta existir) podendo parar a execução (ou alternativamente devolver todas as soluções possíveis). Existem linguagens de programação baseadas nesta técnica de procura, como a linguagem PROLOG.

O seguinte programa PROLOG descreve cinco factos (quem é pai de quem) e uma regra (um avô é um pai de um pai):

```
pai(jorge, maria).  
pai(jorge, jose).  
pai(jorge, ana).
```

```
pai(carlos,luis).  
pai(jose,joao).  
avo(X,Y) :- pai(X,Z), pai(Z,Y).
```

Quando é colocada uma questão, o PROLOG procura por retrocesso a primeira solução na sua base de conhecimento (i.e., os factos e as regras descritas no programa) que satisfaça essa questão. Baseado no programa anterior e com a questão: “*De quem o Jorge é avô?*” que se traduz como:

```
?- avo(jorge,Y).
```

Obter-se-ia o resultado:

```
Y = joao.
```

Como? No retrocesso, `avo(jorge,Y)` é substituído por `pai(jorge,Z), pai(Z,Y)`. Procura-se uma solução para `pai(jorge,Z)` e encontra-se `Z=maria` (o primeiro facto). Mas para `Z=maria` o resto da expressão, `pai(maria,Y)`, não tem solução na base de conhecimento. Então, a busca passa para a segunda hipótese de `pai(jorge,Z)` que é `Z=jose` (o segundo facto). Para `Z=jose`, a busca procura uma solução para `pai(jose,Y)`. Esta procura tem uma solução no facto `pai(jose,joao)`, logo `Y=joao`.

## Um exemplo em Java

O programa seguinte encontra a saída (se existir) de um labirinto bidimensional. Como funciona? A partir da posição inicial percorre todas as quatro direcções possíveis (este, oeste, norte e sul). A cada tentativa o método `procurar()` invoca-se recursivamente para experimentar todas as direcções (excluindo de onde veio) e assim sucessivamente. Com esta atitude garantimos visitar todas as células do labirinto (menos as paredes, claro) descobrindo a eventual saída.

```
public class Lab {  
  
    private String[] lab;  
    private boolean[][] usados;  
  
    public Lab(String[] lab) {  
        this.lab = lab;  
    }  
  
    private boolean eSaida(int l, int c) {  
        return lab[l].charAt(c) == '!';  
    }  
  
    private boolean eParede(int l, int c) {  
        return lab[l].charAt(c) == '*';  
    }  
}
```

Definiu-se um vector de *strings* que armazena a estrutura do labirinto. Assumimos que o símbolo \* representa uma parede e ! representa a saída (observe os métodos privados eSaida() e eParede()).

Quando se opta por procurar uma solução por todas as possibilidades é preciso tomar em conta que o número de hipóteses possíveis é enorme. Para melhorar o desempenho do retrocesso pode-se cortar certos caminhos de procura quando sabemos que estes não produzirão uma resposta. No exemplo do labirinto, uma casa já visitada é uma casa que escusamos visitar outra vez (a saída não passa por ela, caso contrário já a teríamos encontrado). O vector bidimensional de booleanos usados[][] marca as casas já visitadas. Segue o programa de procura por retrocesso:

```
private boolean procurar(int l, int c) {
    if (eSaida(l,c)) {
        System.out.print("SAIDA");
        return true;
    }
    // se for parede ou tiver sido visitado...
    if (eParede(l,c) || usados[l][c])
        return false;

    usados[l][c] = true; // marcar célula como visitada

    if (procurar(l,c+1)) {
        System.out.print("<-E");
        return true;
    }
    if (procurar(l,c-1)) {
        System.out.print("<-O");
        return true;
    }
    if (procurar(l+1,c)) {
        System.out.print("<-S");
        return true;
    }
    if (procurar(l-1,c)) {
        System.out.print("<-N");
        return true;
    }
    return false;
}
```

Existem três bases de recursão: (i) encontra a saída (a solução), (ii) encontra uma parede e (iii) encontra uma célula já percorrida. O passo da recursão é procurar a saída nas quatro direcções possíveis. Falta iniciar a procura:

```
public void encontrarCaminho(int linha, int coluna) {  
    usados = new boolean[lab.length][lab[0].length()];  
    if (procurar(linha, coluna))  
        System.out.println("<-COMEÇO");  
    else  
        System.out.println("NAO EXISTE CAMINHO");  
}  
} // fim da classe Lab
```

Para experimentar o programa:

```

public class Start {
    public static void main(String[] args) {
        String[] teste = { "*****!*",
                           "* *      *",
                           "* ***** *",
                           "*          ***",
                           "*   ***  ** *",
                           "*   * *   * *",
                           "*   *   *   ***",
                           "***   ***   *",
                           "*           *",
                           "*****!*"};
        Lab l = new Lab(teste);
        l.encontrarCaminho(3,1);
    }
}

□ SAIDA<-N<-N<-E<-E<-N<-E<-E<-E<-E<-COM

```

SAIDA<-N<-N<-E<-E<-N<-E<-E<-E<-E<-E<-COMEÇO<-

## Exercícios

1. Implemente um método recursivo que recebe um inteiro positivo  $n$  calcule o  $n$ -ésimo valor da seguinte sequência: 1, 3, 7, 15, 31, 63, 127, ...
  2. Implemente um método recursivo que recebe dois inteiros positivos e calcule o seu produto. Faça um outro método recursivo que calcule a potência do primeiro argumento elevado ao segundo.

3. Implemente um método recursivo que recebe um inteiro positivo  $n$  e calcule o somatório dos primeiros  $n$  inteiros positivos.
4. Implemente um método recursivo que recebe um vector de inteiros, calcule e devolva o maior dos seus elementos.
5. Implemente o algoritmo de Euclides (apresentado no exercício 9 do capítulo 4) de forma recursiva.
6. Que valor é devolvido por `met(n)`?

```
public int met(int n) {  
    if (n == 0)  
        return n;  
    else  
        return n + met(n-1);  
}
```

7. O que devolve o método `find()` assumindo que recebe um vector com inteiros positivos?

```
public int aux(int actual, int[] nums) {  
    if (actual == nums.length-1)  
        return nums[actual];  
    int x = aux(actual+1, nums);  
    if (nums[actual] > x)  
        return nums[actual];  
    else  
        return x;  
}  
  
public int find(int[] numeros) {  
    return aux(0, numeros);  
}
```

8. Implemente um método que recebe um vector de inteiros como argumento e imprime todas as permutações desse vector. Por exemplo, as permutações de  $\{1, 2, 3\}$  são 123, 132, 213, 231, 312 e 321.

9. O número de possibilidades de combinar  $r$  elementos dentro de um conjunto de  $n$  elementos é dado pela seguinte função  $C(n,r) = n! / r!(n! - r!)$ . Encontre uma definição recursiva para esta função e implemente-a.

10. Remova a seguinte recursão terminal:

```
public double power(double base, int exp) {  
    if (exp==0)
```

```
    return 1;
    return base*power(base, --exp);
}
```

11. Remova a seguinte recursão terminal:

```
public int minRec(int[] v, int index, int min) {
    if (index == v.length)
        return v[min];
    return minRec(v, index+1, v[index]<v[min]?index:min);
}

public int getMin(int[] v) {
    return minRec(v, 1, 0);
}
```

12. Considere um tabuleiro quadrado de 5 linhas por 5 colunas e um cavalo do xadrez num dado quadrado. Implemente um programa que determine como o cavalo se pode mover por todas as casas do tabuleiro sem passar duas vezes pela mesma casa. O cavalo é uma peça que se move duas casas nas 4 direções ortogonais e depois move-se mais uma casa para o lado (formando um L).

13. Considere um tabuleiro de xadrez (8 linhas por 8 colunas). Implemente um programa que determine como colocar 8 rainhas do xadrez sem se atacarem mutuamente. Uma rainha move-se uma ou mais casas (até encontrar outra peça ou o fim do tabuleiro) em todas as 8 direções ortogonais e diagonais.



# **6 – Excepções e Asserções**

---

Não existe forma viável de evitar todos os possíveis erros de um programa. Porém, podemos seguir certas regras de modo a minimizar esse potencial, mas inevitável problema. Há diversos tipos de erros: um programa pode conter erros na sua concepção (um algoritmo incorrecto) ou o programa recebe informações erradas do exterior (dados incorrectos ou inesperados).

Sempre que possível, é melhor encontrar os erros de concepção do programa durante o processo de compilação do que durante a execução. Na execução nem sempre se pode parar a aplicação sem prejuízos económicos (e não só). Muitos dos erros ocorrem devido a sintaxe incorrecta. Estes são detectados pelo compilador que exige ao programador as devidas correções antes do programa ser executado e pertencem ao conjunto dos **erros de compilação**. Um programa só é compilado e traduzido numa aplicação executável se descrever um texto sintaticamente correcto.

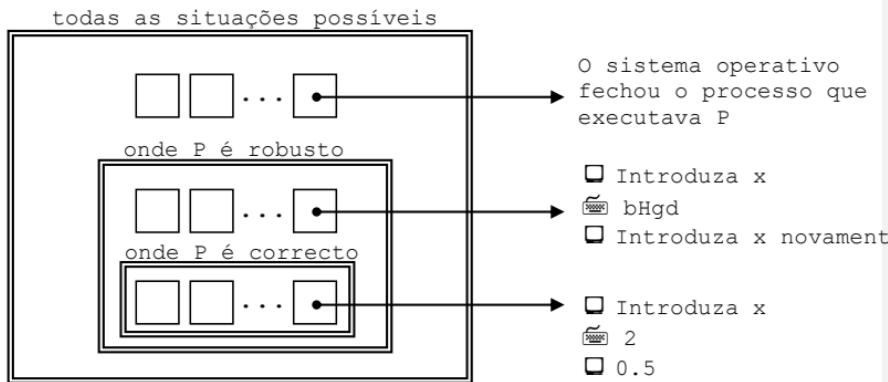
Mas um texto bem escrito não é garantia de correcção. A semântica do programa, a computação associada ao algoritmo descrito pode ser diferente do pretendido. Estes erros escapam ao processo de compilação sendo muito mais perigosos, são os **erros de execução** e, de certa forma, são bombas relógio prestes a explodir a qualquer momento (segundo a Lei de Murphy, nas piores alturas). Por este motivo, no ciclo de vida de uma aplicação há uma fase extremamente importante, depois da fase de implementação, denominada por fase de teste ou validação. Nesta fase são efectuadas simulações do

funcionamento do programa em condições “reais” para detectar quaisquer destes defeitos – os famosos *bugs*<sup>9</sup>.

Segundo Bertrand Meyer, o termo *bug* tem pelo menos três significados distintos: (i) uma escolha errada no desenvolvimento do sistema, i.e., um **erro**, (ii) uma propriedade do programa que invalida o estado do mesmo, i.e., um **defeito** ou (iii) um evento na execução do sistema que o coloca num estado inválido, i.e., uma **falha**. Assim, as falhas são consequências de defeitos que resultaram de erros. Já os erros nos dados são provenientes de acontecimentos que escapam ao controle directo da execução do programa. O programa deve ter (na medida do possível) um comportamento robusto perante esses problemas externos. Se for recebido algo incorrecto, o programa deve ser capaz de processar esse problema e resolvê-lo de alguma forma (por exemplo, pedindo a informação novamente).

Há um conjunto de situações correctas para o qual um programa tem de devolver os resultados esperados – a validação desse conjunto assegura a **correcção** do programa. Há um conjunto de situações incorrectas que o programa deve resolver adequadamente – a validação desse conjunto assegura a **robustez** do programa. A englobar as situações correctas e robustas existe o conjunto de todas as situações possíveis, onde se encontram situações para as quais nada pode ser feito e onde a aplicação se torna inútil.

Considere o seguinte exemplo: um programa P recebe um real  $x$  e executa a operação  $1/x$  (repetindo o pedido de  $x$  se for introduzido um valor inválido):



<sup>9</sup> O nascimento do termo *bug* é atribuído a uma das pioneiras da programação, Grace Hopper (que desenvolveu o primeiro compilador e muito do trabalho que levou à criação do COBOL). Em 1944, Grace trabalhava no computador MARK I quando um técnico descreveu uma falha no sistema provocada pela presença de uma traça entre os componentes do computador. Porém este termo já era conhecido antes do advento dos computadores modernos e significava um defeito industrial ou eléctrico (cf. Eric Raymond, *The New Hacker's Dictionary*).

Um programa está **correcto** se calcula as respostas correctas para o conjunto de instâncias do problema onde deve funcionar. A correcção de um programa é a propriedade mais importante no domínio da programação. Um programa que devolve respostas erradas é inútil.

Um programa **robusto** é um programa que recupera de certas situações anómalas sem entrar num estado inválido. A propriedade de ser robusto não é essencial como a correcção, mas é um sinónimo de qualidade. A robustez pode mesmo fazer parte dos requisitos do programa.

Considere o método que divide um atributo por um parâmetro real:

```
public void divide(double a) {  
    x /= a;  
}
```

Se o argumento *a* for igual a zero obtemos um erro de execução: divisão por zero. Uma forma de resolver esse problema seria:

```
public final int ARG_ZERO = -1;  
public final int ARG_OK = 0;  
  
public int divide(double a) {  
    if (a==0) return ARG_ZERO;  
    x /= a;  
    return ARG_OK;  
}
```

Quem invocasse o método teria de lidar com a situação:

```
if (obj.divide(y) != obj.ARG_OK)  
    ... // fazer algo para corrigir o problema
```

Este tipo de abordagem é usado em diversas linguagens de programação. Alguns problemas:

- A condição que verifica se não há anomalias (neste caso, se o divisor é diferente de zero) é duplicada: há um teste no método e outro na invocação do método.
- O método em si não deve ser sobrecarregado com estes detalhes. O método possui um domínio de aplicação (ou seja, o conjunto de valores admissíveis para cada um dos seus parâmetros) que deve ser respeitado por quem o invoca. A responsabilidade recai em quem invoca o método, desde que o método anuncie previamente o seu domínio. Retomaremos este assunto no capítulo 11.
- Para aplicações maiores é extremamente difícil manter todas as situações inesperadas sob controlo. Há um conjunto enorme de pequenos erros que “quase nunca ocorrem” e que por essa razão nunca são testados. Mas todos os acontecimentos raros possuem algo em comum: por vezes acontecem! Este era um

dos motivos que limitava a existência de grandes sistemas computacionais robustos e de fácil manutenção.

No Java há mecanismos integrados na linguagem que gerem estas situações de forma mais organizada minimizando potenciais problemas:

- Forçar a declaração explícita das variáveis e dos objectos usados. Em certas linguagens é possível atribuir um valor a uma variável não declarada. Por exemplo, a instrução de atribuição `xyz = 1.0` cria uma nova variável (se ainda não existir) inicializada com o valor real 1.0. O problema desta abordagem é que se escrevemos `xy = 1.0` (esquecemo-nos de digitar o `z`) o compilador não encontra nenhum erro: cria uma nova variável `xy` e não altera `xyz` como seria esperado. Foi criado um erro de execução detectável em tempo de compilação. Como o Java força a declaração, este problema não ocorre nos seus programas.
- A memória reservada para um vector é fixa. Os únicos índices válidos estão dentro do intervalo definido pela dimensão do vector. Se uma instrução tentar aceder a um índice do vector que não existe, o que acontece? Este tipo de erro não pode ser detectado em tempo de compilação, porém, pode sê-lo em tempo de execução. Existem duas atitudes distintas: (i) verificar a cada acesso se o índice é válido – se não for, parar o método actual e gerar uma mensagem de erro ou (ii) não fazer nada e esperar que o programador não cometa um erro destes. A primeira abordagem – usada pelo Java – é mais lenta (é preciso verificar cada acesso) mas mais segura (na maioria das situações é melhor um programa parar do que continuar a executar num estado inválido.). A segunda abordagem, usada em linguagens como o C ou o C++, é mais rápida mas bastante menos segura.
- Restringir o uso dos apontadores. Em Java, cada objecto ou vector é acedido através de uma variável (do tipo referência) que armazena a sua posição de memória. Cada variável deste tipo ou contém o valor especial `null` (que identifica a ausência de referência associada) ou contém uma referência para um objecto desse tipo criado pelo operador `new`. No C e C++ as estruturas de referência (os apontadores) são muito mais flexíveis. Uma referência armazena um endereço (que não sabemos nem queremos saber) de um objecto de um tipo conhecido. Um apontador armazena um endereço de memória (que conhecemos) onde (eventualmente) se encontra um objecto de um (eventual) tipo de dados. Um dos problemas no uso de apontadores é que nesse endereço de memória pode estar qualquer outro tipo de informação (ou não estar informação alguma...). É disponibilizada uma aritmética de apontadores onde se pode somar ou subtrair endereços de memória para que o apontador “deslize” pela memória disponível da aplicação. Há a possibilidade de um apontador ser redirecionado para um endereço onde se encontra um objecto de um tipo diferente, ou mesmo para uma memória onde não existe qualquer objecto. Este mecanismo, quando mal usado, pode provocar uma plethora de problemas diferentes.

- Outro tipo de problema surge quando a memória reservada em tempo de execução não é libertada convenientemente (em inglês, *memory leak*). Um endereço de memória torna-se inacessível se não existir uma referência para ele. Neste caso não existe forma da memória ser libertada, diminuindo o espaço disponível da aplicação sem qualquer contrapartida. No Java isso não é possível devido ao mecanismo automático de recolha de lixo referido no capítulo 4.
- Outro problema associado à memória ocorre nas situações onde uma dada estrutura de dados esgota a sua capacidade (em inglês, *buffer overflow*). Se o método de inserção não verificar esse facto, pode continuar a inserir dados para fora do espaço reservado e destruir outras informações. No Java isto é impossível: não é permitido armazenar informação em memória que não tenha sido reservada correctamente (tanto em relação ao espaço que ocupa como ao tipo que representa).

Ou seja, existem erros que o desenho cuidado de uma linguagem pode evitar. O preço a pagar por esta capacidade é um maior tempo de compilação – não tão grave dado que só ocorre durante a fase de implementação do programa – e um maior tempo de execução – que (excluindo situações extremas em que o tempo de resposta deve ser minimizado a todo o custo) é uma desvantagem largamente compensada pelo aumento de segurança associado. O Java adoptou esta atitude de segurança.

## 6.1 Excepções

Uma **excepção** é um objecto criado quando uma situação anómala ocorre, acompanhado por uma interrupção da execução normal de um programa. A motivação básica deste conceito é que ao descobrir um erro num método, mesmo que ainda não saibamos o que fazer, sabemos de certeza que não podemos continuar a executar as próximas instruções como se nada tivesse acontecido. O fluxo de execução é detido e é “lançada” uma excepção para alguém supostamente apto a resolver a questão. Para quem? Para o responsável pela invocação do método onde ocorreu o erro, o que corresponde à ideia da estrutura de responsabilidade hierárquica de uma instituição: quando alguém não consegue lidar com uma situação, esta é transferida para o responsável directo. Então:

- o método responsável “apanha” a excepção e resolve o problema localmente.
- o método responsável não consegue lidar com a excepção sendo transferida para quem invocou este segundo método (e assim por diante). Se ninguém conseguir resolver o problema, o programa pára enviando um relatório do erro para o canal de erro disponibilizado pelo sistema operativo.

Uma excepção é um objecto de uma classe derivada<sup>10</sup> da classe `Throwable` que define objectos a ser lançados (do inglês, *thrown*) e apanhados (do inglês, *caught*) por determinados comandos Java. Esta classe possui dois métodos úteis:

- `String getMessage()` – devolve o relatório enviado por quem lançou a excepção.
- `void printStackTrace()` – imprime para o canal de erro o caminho percorrido pela excepção desde que lançada até ser apanhada.

Derivados da classe `Throwable` existem duas subclasses: `Error` e `Exception`. Na maioria das situações, os problemas do tipo `Error` não são apanhados pelo programador dado referirem-se a situações não recuperáveis (como falta de memória).

Dentro da classe `Exception` existem as excepções do tipo `RunTimeException` e as restantes. A classe `RunTimeException` contém excepções não verificáveis (do inglês, *unchecked exceptions*), i.e., excepções que não precisam ser anunciadas pelo programador. Tipicamente referem-se a situações como divisões por zero (`ArithmeticException`), acessos a índices não existentes de vectores (`IndexOutOfBoundsException`) ou acessos a métodos através de referências nulas (`NullPointerException`). Referem-se a erros de execução do programa e implicam a terminação imediata do programa. As restantes excepções são verificáveis (do inglês, *checked exceptions*) e têm de ser consideradas pelo programador, i.e., lançadas ou apanhadas. Referem-se a situações que precisam de ser tratadas durante a execução, por exemplo, um ficheiro que não existe (`IOException`) ou um valor mal introduzido pelo utilizador.

## Lançar Excepções

Para lançar uma excepção é necessário criar um novo objecto do tipo da excepção em questão e explicitar esse lançamento com o comando `throw`:

```
if (a==0)
    throw new ArithmeticException();
```

É possível associar um objecto do tipo excepção a uma frase:

```
if (a==0)
    throw new ArithmeticException("Valor nulo!");
```

Quando a instrução é executada, o método que a contém termina e lança o objecto do tipo excepção para o método que o invocou. É preciso indicar no cabeçalho do método esta situação especial. Por exemplo:

---

<sup>10</sup> Falaremos de classes derivadas, subclasses e superclasses no capítulo 10.

```
public static double divide(double a)
    throws ArithmeticException {
    if (a==0)
        throw new ArithmeticException("Valor nulo!");
    return x / a;
}
```

O método que recebe a excepção será, eventualmente, capaz de apanhar e resolver o problema. Do ponto de vista deste método, ou é realizado o trabalho pretendido se o parâmetro estiver correcto (for diferente de zero) ou é enviado um erro se o parâmetro for inválido.

Um método pode lançar mais do que uma excepção. O cabeçalho do método deve explicitar todas as excepções que lança, separando-as por vírgulas.

```
public void metodo() throws ArithmeticException,
    NullPointerException {
    ...
}
```

Um método com um cabeçalho sem `throws` não levanta excepções que não seja capaz de tratar (exceptuando excepções do tipo `RuntimeException`). Por outro lado, é possível informar que um método lança uma determinada excepção e efectivamente a excepção nunca ocorrer.

## Apanhar Excepções

Para apanhar uma excepção é preciso inserir as instruções onde pode ocorrer o problema dentro de um bloco `try-catch`. Um exemplo que usa o método `divide()`:

```
System.out.println("Introduza um número positivo");
double x = sc.nextDouble();
try {
    System.out.print(divide(x));
}
catch (ArithmetricException e) {
    System.out.print(e.getMessage());
}
□ Introduza um número positivo←
↙ 0
□ Valor nulo!
```

A excepção foi levantada no método `divide()` e foi apanhada pelo bloco `try-catch` que recebeu o objecto criado por `new ArithmetricException("Valor Nulo!")`. A

referência e do bloco `catch` recebe o endereço da exceção para, neste exemplo, imprimir a frase associada. Desta forma, aquilo que diz respeito à execução normal do programa é separado do tratamento das situações de erro. Esta modularização ajuda a controlar a complexidade geral do programa.

Por omissão, uma exceção é um acontecimento suficientemente grave para provocar a terminação do método. Pode ser necessário desfazer o trabalho do método de modo a voltar ao estado inicial antes da tarefa se ter iniciado (em inglês, *rollback*). Por vezes, a situação não é tão grave e o bloco `try-catch` pode tentar resolver o problema invocando novamente o método onde ocorreu o problema.

Se completarmos o exemplo anterior nesta perspectiva, é possível que se realizarmos uma nova leitura do real, este esteja correcto. Quando o valor lido é igual a zero, as instruções no bloco `catch` enviam o relatório recebido para o canal de erro (através do método `getMessage()`) e pedem um novo valor. Esta forma de tratar exceções é útil em situações locais, mas pode tornar-se confusa dado que o tratamento da exceção deve conhecer o âmbito no qual se refere a recuperação da exceção – o que pode ser alterado durante a fase de implementação.

```
while (true)
{
    try {
        double x = sc.nextDouble();
        divide(x);
        System.out.println("OK!");
        break;
    }
    catch (ArithmaticException excp) {
        System.err.println("ERRO: " + excp.getMessage());
    }
}
↙ 0
↗ ERRO: Valor nulo! ↵
↙ 2
□ OK! ↵
```

Um bloco `try-catch` pode conter diversos blocos `catch`, cada um tratando um problema distinto. Por exemplo:

```
double [] vector = new double[20];
System.out.println("Número positivo");
double x = sc.nextDouble();
System.out.println("Indice do vector [1-20]");
int i = sc.nextInt();
```

```
try {
    vector[i] = divide(x);
}
catch (ArithmaticException e) {
    System.out.print(e.getMessage());
}
catch (IndexOutOfBoundsException e) {
    System.out.print("Indice do vector incorrecto!");
}
```

## O Bloco Final

Os blocos try-catch possuem opcionalmente um bloco finally onde se colocam instruções que têm de ser executadas aconteça o que acontecer<sup>11</sup> dentro do bloco, mesmo que ocorra uma exceção ou seja executada uma instrução de return.

```
tentou = false;
try {
    Tarefa t = proximaTarefa();
    if (t.vazia())
        return;
    else
        t.executar()
}
catch (Exception e) {
    System.err.println(e.getMessage());
    return;
}
finally {
    tentou = true;
}
```

Quer não exista uma próxima tarefa (o método sai através do return), quer exista um problema inesperado com a tarefa (o bloco catch é executado), quer tudo funcione correctamente, a variável booleana tentou é sempre verdadeira.

Os blocos finais são úteis para guardar instruções de “limpeza” que têm obrigatoriamente de ser executadas mesmo que ocorram exceções de algum tipo.

---

<sup>11</sup> Na realidade, existe uma forma para o bloco finally não ser executado. Com o comando System.exit() a máquina virtual do Java pára e mais nenhuma instrução é executada.

```
Classe obj = new Classe();
try {
    ...
    // código onde se usa obj
}
finally {
    obj.libertarRecursos();
}
```

## Relançar Excepções

Uma excepção, depois de apanhada, pode ser relançada pelo bloco `catch`. Este tipo de situação é útil quando o tratamento da excepção é realizado por etapas. Assim, o método onde se situa o `catch` é interrompido e o controle passa para o método acima.

```
try {
    ...
}
catch (Exception e) {
    System.err.println(e.getMessage());
    throw e;
}
```

## Lançamentos Automáticos

Entre o conjunto de excepções predefinidas, algumas lidam automaticamente com situações comuns. Por exemplo, para usar o método `m()` do objecto referenciado por `obj` escrevemos:

```
obj.m();
```

Mas o que acontece se não existir objecto, ou seja, `obj` é igual a `null`? Em princípio, para cada acesso a cada componente de cada objecto teríamos de escrever:

```
if (obj==null)
    throw new NullPointerException();
obj.m();
```

Felizmente, esta condição é realizada de forma automática pelo compilador Java. Só é necessário apanhar excepções nas situações especiais do nosso programa em que sabemos que algum problema possa acontecer. Claro que numa dada situação onde existe a possibilidade da referência ser nula pode ser conveniente evitar este tipo de excepções:

```
if (obj!=null)
    obj.m();
else
    ... // tentar resolver o problema
```

## 6.2 Asserções

Uma **asserção** expressa uma propriedade que se deve verificar num dado ponto do programa, i.e., é uma condição lógica que tem obrigatoriamente de ser verificada para o programa continuar a ser executado. O objectivo da asserção é disponibilizar ao programador a possibilidade para testar assunções sobre o estado do programa num dado momento. Se a asserção for verdadeira, o programa confirma a nossa expectativa sobre o seu estado. Caso contrário, algo incorrecto aconteceu antes da asserção e o programa deve ser terminado. Cada vez que uma asserção é verificada pelo programa aumenta a confiança sobre a correcção do próprio programa. Assim, este tipo de raciocínio é principalmente útil na fase de teste para detectar erros de execução.

Uma asserção Java tem duas formas: (i) a palavra `assert` seguida de uma expressão lógica, ou (ii) a palavra `assert` seguida de duas expressões: uma expressão lógica (para o teste da assunção) e uma outra devolvendo um valor predefinido ou uma frase (um relatório do erro, por exemplo). As asserções têm de ser inseridas em instruções de bloco.

Alguns exemplos:

```
assert x>=0;
assert v.length() == x%y;
assert x>=0 : "O valor de x é negativo!";
```

Se a expressão lógica da asserção for falsa, o método pára e é lançada a excepção `AssertionError`. Esta excepção é do tipo `Error` pertencendo à classe dos problemas que, em princípio, não possuem recuperação: o estado do programa desviou-se do esperado. Não se deve apanhar uma excepção deste tipo. O objectivo da asserção é precisamente detectar falhas de programação e fazer com que estas sejam corrigidas para o programa ser recompilado e novamente executado.

### Uso de Asserções

É normal os programadores incluírem comentários sobre certas assunções feitas informalmente. Por exemplo:

```
resultado = Math.sqrt(valor); // valor >= 0
```

No entanto este sistema não fornece nenhuma solução caso o problema ocorra: é somente um comentário sobre uma assunção não verificada. Uma solução alternativa com asserções:

```
assert valor >= 0 : "Variável \'valor\' negativa!";
Resultado = Math.sqrt(valor);
```

As asserções são úteis em comandos `switch` quando o valor recebido não é válido:

```
switch (digito) {           // digito é zero ou um
    case 0:
        ...
        break;
    case 1:
        ...
        break;
    default:                 // nunca deveria chegar aqui
        assert false : "Dígito Inválido: " + digito;
}
```

Se existe algum local no código que nunca é executado, pode-se colocar uma asserção falsa para detectar potenciais erros.

## Mau uso de Asserções

O conjunto de asserções testa a correcção do programa. As asserções nunca devem modificar o estado do programa, ou seja, nunca devem provocar **efeitos secundários**. Se isso ocorrer, o programa tem comportamentos diferentes conforme as asserções estão ligadas ou desligadas. Segue um mau uso de uma asserção:

```
public int incrementa() {
    return ++contador;
}

...
assert incrementa()>0;      // ERRADO!
```

Ao calcular o valor da expressão lógica, o atributo `contador` foi incrementado. Se a asserção for desligada, esse incremento não é realizado.

As asserções não devem ser usadas para verificar os argumentos de entrada de um método público. Para além de não funcionarem se forem desactivadas, as asserções somente lançam um tipo de excepção: `AssertionError`. É sempre mais conveniente lançar excepções especializadas que definem o problema ocorrido.

```
public int setVelocidade(double v) {
    assert v>MAX_VEL : "Vel. Inválida";           // Hmm...
    ...

}

public int setVelocidade(double v)
    throws InvalidSpeed {
    if (v>MAX_VEL)
        throw new InvalidSpeed("V. Inválida"); // Melhor!
    ...
}
```

## Ligar e Desligar as Asserções

Certas assunções podem demorar muito a serem verificadas provocando um atraso excessivo no programa, principalmente quando este já passou pela fase de teste e está a ser usado pelo cliente. É possível desligar o uso das asserções. Quando desligadas, as asserções são equivalentes a instruções vazias, não penalizando o desempenho do programa.

As asserções são uma extensão da linguagem introduzida na versão Java 2/1.4: não funcionam nas versões anteriores (em programas compilados por versões mais antigas a palavra `assert` pode, eventualmente, ter sido usada como nome para uma classe, método ou atributo).

Considere o exemplo:

```
class Assercao {
    public static void main(String[] args) {
        assert args.length > 0 : "Sem parâmetros!";
        System.out.println(args[0]);
    }
}
```

Quando se quer usar as asserções deve-se dizer explicitamente ao compilador:

```
javac -source 1.4 Assercao.java
```

Se nada for dito, o programa é executado com as asserções desligadas.

```
java Assercao
java.lang.ArrayIndexOutOfBoundsException
    at Assercao.main(assercao.java:6)
Exception in thread "main"
```

Para activar as asserções é necessário fornecer à máquina virtual, a opção `-ea`:

```
java -ea Assercao
java.lang.AssertionError: Sem parâmetros!
    at Assercao.main(assercao.java:5)
Exception in thread "main"
```

De notar que `AssertionError` é uma excepção interna e pode ser lançada explicitamente mesmo que as asserções estejam desligadas.

```
... throw new AssertionError("Erro!");
```

Se pretendermos que as asserções estejam ligadas, é possível forçar essa situação criando um efeito secundário deliberado: coloca-se as próximas instruções num bloco de inicialização.

```
static {
    boolean ligado = false;
    assert ligado = true;      // efeito secundário!
    if (!ligado)
        throw new RuntimeException("Ligar Asserções!");
};
```

Aconselha-se porém a usar este tipo de técnicas com bastante parcimónia. Efeitos secundários, como o próprio nome indica, podem produzir consequências inesperadas.

---

## Exercícios

1. Qual é a diferença entre excepções verificáveis e não verificáveis?
2. O que ocorre se executar a raiz quadrada de um número negativo? E se executar o método `Math.pow()` com ambos os argumentos iguais a zero?
3. Pode haver um bloco `try` sem um `catch`? Um `catch` pode existir sem um `try` associado?
4. Quando são criados objectos de tipo excepção?
5. Qual é a diferença entre `throw` e `throws`?
6. O exemplo seguinte está correcto? Justifique.

```
try {
    ...
}
finally {
    ...
}
```

7. Alguma destas asserções está incorrecta? Justifique.

- i) assert a == 1;
- ii) assert a = 1;
- iii) assert a > 0 : "Positivo";
- iv) assert "Inválido";
- v) assert a==0 ? b : false;

# 7 – Entrada e Saída de Dados

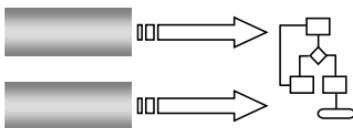
---

Um programa não representa uma entidade separada do exterior. Algum processo externo deve iniciar a execução do programa (normalmente um evento processado pelo sistema operativo). O programa também recebe e envia informação para o exterior, seja para um utilizador humano (por exemplo, recebendo informação através do teclado ou enviando informação para o monitor), seja para outro programa (por exemplo, num ambiente cliente/servidor onde ocorrem trocas de dados pela Internet) ou para um dispositivo de memória secundária (por exemplo, um ficheiro de dados a ser lido de um disco rígido).

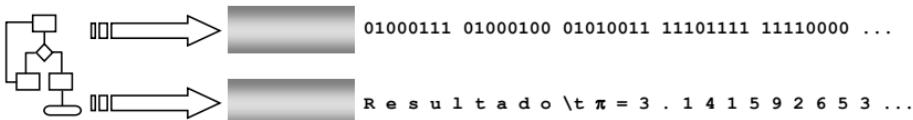
É comum separar os dados que entram ou saem do ambiente para o programa em **informação binária** e **informação textual**. A informação binária é constituída por zeros e uns que codificam qualquer tipo de dados e que têm de ser interpretados pela aplicação. A informação textual é constituída por letras, dígitos, caracteres especiais que no Java corresponde ao tipo primitivo `char`. Estes canais de informação são designados por **fluxos** (do inglês, *streams*), nomeadamente fluxos binários e fluxos de texto. Para cada tipo de dados há dois fluxos: um de entrada e um de saída. Assim, existem quatro fluxos padrão.

Um fluxo binário de entrada e um fluxo de texto de entrada:

```
... 01010010 01110000 00010011 11100011 00010011  
... n o m e : l u i s , \n i d a d e : 3 3 a n o s
```

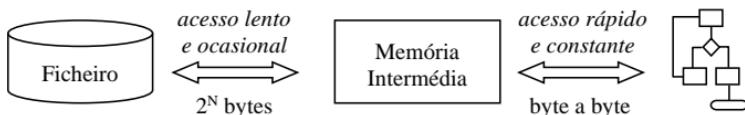


E um fluxo binário de saída e um fluxo de texto de saída:



O acesso à **memória secundária** (informação contida em ficheiros armazenados em discos rígidos, disquetes ou outros suportes) é muito mais lento – na ordem das centenas ou mesmo milhares de vezes – que o acesso à **memória primária** (como a memória RAM). A razão da diferença de desempenho resulta do facto que o acesso é realizado mecanicamente no primeiro caso e electronicamente no segundo. Para minimizar essa demora, os sistemas de acesso à memória secundária não lêem ou escrevem bit a bit: tratam conjuntos de 512, 1024 ou 2048 bytes de uma só vez. Ler um byte ou ler 512 bytes demora, em termos mecânicos, o mesmo tempo.

Aproveitando este facto criou-se o conceito de **memória intermediária** ou **memória tampão** (do inglês, *buffer*). Quando o programa requer a leitura do primeiro byte, este é lido juntamente com um conjunto de bytes sendo colocados num local apropriado na memória primária. Se porventura o programa requisitar a leitura do próximo byte, esse já se encontra na memória intermédia tornando o acesso muito mais eficiente.



A forma como a linguagem Java trata este assunto é relativamente complexa. Existem múltiplas classes com diferentes comportamentos que precisam de se integrar para obter os métodos de leitura e escrita. Não é o objectivo deste capítulo mostrar todas as possibilidades (consultar a bibliografia recomendada para esse efeito) mas apenas apresentar os conceitos essenciais e como os aplicar na linguagem.

## 7.1 As Classes Gerais de Entrada/Saída

Tudo o que diz respeito à entrada e saída de dados pertence ao pacote `java.io` (mais informações sobre pacotes no anexo A). Para os exemplos seguintes funcionarem é necessário importá-lo colocando no início de cada ficheiro, a seguinte linha:

```
import java.io.*;
```

Existem quatro classes abstractas para lidar com os referidos fluxos:

- *InputStream* – a classe que lê fluxos binários de informação.
- *OutputStream* – a classe que escreve fluxos binários de informação.
- *Reader* – a classe que lê fluxos de texto.
- *Writer* – a classe que escreve fluxos de texto.

Alguns métodos da classe *InputStream*:

- `int read()` – lê um byte (entre o valor 0 e 255) e devolve-o. Se não conseguir ler devolve -1.
- `int read(byte[] vector, int i, int n)` – lê até  $n$  bytes (se possível, pois pode chegar ao fim do fluxo) e coloca-os entre as posições `vector[i]` e `vector[i+n-1]`. Devolve o número de bytes lidos. Se não ler nenhum devolve -1.
- `long skip(long n)` – avança  $n$  bytes, se conseguir. Devolve o número de bytes avançados.
- `void close()` – fecha o fluxo de dados. Pode ser necessário para libertar recursos associados ao fluxo. Fechar um ficheiro já fechado não produz qualquer efeito.

Alguns métodos da classe *OutputStream*:

- `void write(int i)` – escreve os primeiros 8 bits do inteiro  $i$ .
- `void write(byte[] vector, int i, int n)` – escreve  $n$  bytes do vector a partir da posição `vector[i]`.
- `void flush()` – escreve todos os bytes à espera de serem escritos. Quando se efectua um `write()` não é obrigatório que o sistema o escreva nesse instante no ficheiro. Por questões de eficiência é normal esperar por um número de bytes antes de os escrever todos de uma vez (uma vez mais, a questão da memória intermédia).
- `void close()` – fecha o fluxo de dados (e executa automaticamente o `flush()`).

Alguns métodos da classe *Reader*:

- `int read()` – lê um carácter Unicode e devolve-o como um inteiro entre 0 e 65535.
- `int read(char[] vector, int i, int n)` – lê até  $n$  caracteres (se possível, pois pode chegar ao fim do fluxo) e coloca-os entre as posições `vector[i]` e `vector[i+n-1]`. Devolve o número de bytes lidos. Se não ler nenhum devolve -1.
- `long skip(long n)` – avança  $n$  caracteres se for possível. Devolve o número de bytes avançados.
- `void close()` – fecha o fluxo de dados.

Alguns métodos da classe *Writer*:

- `void write(int c)` – escreve  $c$  como um carácter. Apenas os primeiros 16 bits são escritos.
- `void write(char[] vector, int i, int n)` – escreve  $n$  caracteres do vector a partir da posição `vector[i]`.
- `void write(String frase)` – escreve a String `frase`.
- `void flush()` – escreve todos os caracteres à espera de serem escritos.
- `void close()` – fecha o fluxo de dados.

Nas classes descritas neste capítulo, se algo inesperado acontecer, os métodos lançam a excepção `IOException`. Estas classes não podem ser usadas para definir objectos mas fornecem ferramentas para as próximas classes.

## 7.2 Os Fluxos Básicos

Existem quatro classes básicas para processar informação directamente dos ficheiros. Cada uma deriva da respectiva classe da secção anterior (podendo, assim, utilizar os seus métodos).

### Fluxo Binário de Entrada

- `FileInputStream` – a classe que lê fluxos binários de informação de ficheiros.



Os construtores da classe `FileInputStream`:

- `FileInputStream(String caminho)` – abre o ficheiro na localização dada por *caminho*. Se o ficheiro não existir é levantada a excepção `FileNotFoundException`.
- `FileInputStream(File ficheiro)` – abre o ficheiro descrito pelo objecto *ficheiro*. Se o ficheiro não existir é levantada a excepção `FileNotFoundException`.

O próximo exemplo lê o número de bytes de um ficheiro cujo nome é dado pelo primeiro argumento da linha de execução do programa (i.e., `args[0]`):

```
import java.io.*;
class ContarBytes {
    public static void main(String[] args)
        throws IOException {
        int nBytes = 0;
        FileInputStream f =
            new FileInputStream(args[0]);
        while (f.read() != -1)
            nBytes++;
        System.out.println("Número = " + nBytes);
        f.close();
    }
}
```

Se este exemplo (depois de compilado) for executado com o comando:

java ContarBytes teste

E na mesma directória se encontrar um ficheiro de nome teste constituído por 1200 bytes, o programa produz o resultado:

Número = 1200 ↵

## Fluxo Binário de Saída

- *FileOutputStream* – a classe que escreve fluxos binários em ficheiros.



Os construtores da classe *FileOutputStream*:

- *FileOutputStream(String caminho)* – abre o ficheiro na localização dada por *caminho*. Se o ficheiro ainda não existe é criado. Se existe, o conteúdo é apagado.
- *FileOutputStream(String caminho, boolean anexar)* – abre o ficheiro na localização dada por *caminho*. Se o argumento *anexar* for verdadeiro e o ficheiro já existir, a informação prévia é mantida. O processo de escrita anexa a nova informação a partir do fim do ficheiro.
- *FileOutputStream(File ficheiro)* – abre o ficheiro descrito pelo objecto *ficheiro*.
- *FileOutputStream(File ficheiro, boolean anexar)* – abre o ficheiro descrito pelo objecto *ficheiro*. Se o argumento *anexar* for verdadeiro e o ficheiro já existir, a informação prévia é mantida. O processo de escrita anexa a nova informação a partir do fim do ficheiro.

O próximo exemplo escreve alguns bytes num ficheiro cujo nome é por `args[0]`:

```
import java.io.*;
class EscreverBytes {
    public static void main(String[] args)
        throws IOException {
        FileOutputStream f =
            new FileOutputStream(args[0]);
        f.write(65); f.write(66); f.write(67);
        f.close();
    }
}
```

Se este exemplo foi compilado e posteriormente executado com o comando:

✉ java EscreverBytes teste

O programa cria um ficheiro de nome `teste` constituído por 3 bytes de valores 65, 66 e 67. Se o ficheiro for lido por um editor de texto ver-se-ia o seguinte conteúdo (porque 65, 66 e 67 são os valores Unicode das letras A, B e C):

📁 [teste] ABC

## Fluxo de Texto de Entrada

- `FileReader` – a classe que lê fluxos de texto de ficheiros.



Os construtores da classe `FileReader`:

- `FileReader(String caminho)` – abre o ficheiro na localização dada por *caminho*. Se o ficheiro não existir levanta a excepção `FileNotFoundException`.
- `FileReader(File ficheiro)` – abre o ficheiro descrito pelo objecto *ficheiro*. Se o ficheiro não existir levanta a excepção `FileNotFoundException`.

O próximo exemplo lê o ficheiro cujo nome é novamente dado por `args[0]` e calcula o número de vezes que ocorre o dígito 1:

```
import java.io.*;
class ContarUns {
    public static void main(String[] args)
        throws IOException {
        int c, uns=0;
        FileReader f = new FileReader(args[0]);
        while ((c = f.read()) != -1)
            if ((char)c=='1')
                uns++;
        System.out.print("Ocorrem " + uns + " uns!");
        f.close();
    }
}
```

Com esta classe compilada veja-se o resultado do exemplo:

📁 [numeros] 12345432122111
✉ java ContarUns numeros
□ Ocorrem 5 uns!

## Fluxo de Texto de Saída

- *FileWriter* – a classe que escreve fluxos de texto em ficheiros.



Os construtores da classe *FileWriter*:

- *FileWriter(String caminho)* – abre o ficheiro na localização dada por *caminho*. Se o ficheiro ainda não existe é criado. Se existe, todo o conteúdo é apagado.
- *FileWriter(String caminho, boolean anexar)* – abre o ficheiro na localização dada por *caminho*. Se o argumento *anexar* for verdadeiro e o ficheiro já existir, a informação prévia é mantida. O processo de escrita anexa a nova informação a partir do fim do ficheiro.
- *FileWriter(File ficheiro)* – abre o ficheiro descrito pelo objecto *ficheiro*.
- *FileWriter(File ficheiro, boolean anexar)* – abre o ficheiro descrito pelo objecto *ficheiro*. Se o argumento *anexar* for verdadeiro e o ficheiro já existir, a informação prévia é mantida. O processo de escrita anexa a nova informação a partir do fim do ficheiro.

O próximo exemplo escreve alguns bytes num ficheiro:

```
import java.io.*;
class EscreverFrase {
    public static void main(String[] args)
        throws IOException {
        String s = "olá mundo!!!";
        FileWriter f = new FileWriter(args[0]);
        f.write(s);
        f.close();
    }
}
```

Observe o exemplo depois de compilar `escreverFrase.java`:

```
java EscreverFrase resultado.txt
[resultado.txt] olá mundo!!!
```

## Um Exemplo com Fluxos de Texto

Vamos inserir na classe *Carro* (apresentada no capítulo 4) métodos que permitem ler/gravar o estado do objecto de/para um ficheiro de texto.

```
import java.io.*;
public class Carro {
    ...
}
```

A forma mais prática de gravar os atributos é colocar cada valor numa linha diferente. Assim, define-se um método que converte o estado do objecto para esse formato:

```
public String convertFileFormat() {
    return matricula + "\n" +
           modelo      + "\n" +
           ano         + "\n" +
           bonsTravoes + "\n" +
           velocidade  + "\n";
}
...
}
```

Define-se agora o método `write()` que dado um nome, escreve a informação num ficheiro com esse nome:

```
public void write(String filename)
                  throws IOException {
    FileWriter f = new FileWriter(filename);
    f.write(convertFileFormat());
    f.close();
}
...
}
```

Para ler a informação da memória secundária, optámos por um novo construtor que recebe o nome do ficheiro e lê a informação para os respectivos atributos:

```
public Carro(String filename) throws IOException {
    FileReader f = new FileReader(filename);
    matricula   = readLine(f);
    modelo     = readLine(f);
    ano        = Integer.parseInt(readLine(f));
    bonsTravoes = readLine(f).equals("true");
    velocidade = Double.parseDouble(readLine(f));
}
...
}
```

O método auxiliar `readLine()` recebe a referência para um objecto do tipo `FileReader` e lê a próxima linha (eliminando o carácter `\n`).

```

public String readLine(FileReader f)
    throws IOException {
    String s = "";
    int c;
    while ((c = f.read()) != '\n')
        s += (char)c;
    return s;
}

} //end Class Carro

```

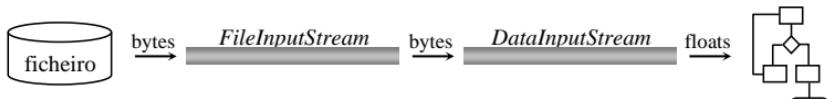
Este método porém, não se preocupa se o ficheiro chegar ao fim prematuramente, i.e., antes de uma mudança de linha. Como exercício, modifique o método de modo a devolver os caracteres lidos se chegar prematuramente ao fim do ficheiro, ou `null` se não ocorrer a leitura de pelo menos um carácter.

Porém, para cada classe que tivesse de ler/escrever a informação seria necessário repetir o método `readLine()`. Veremos em seguida classes mais elaboradas que já incluem este e outros métodos úteis.

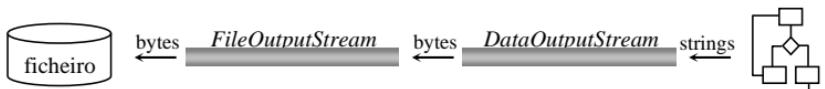
### 7.3 Os Fluxos Avançados

As classes referidas atrás manipulam os ficheiros num nível demasiado baixo. Na maioria dos casos, os ficheiros que interessam são constituídos por inteiros, reais ou outros tipos de dados mais complexos.

Apresentamos algumas classes avançadas que definem fluxos que não acedem directamente aos ficheiros mas acedem a um dos fluxos básicos. Os fluxos podem e devem ser utilizados em sequência. Por exemplo, a informação de um fluxo `FileInputStream` pode ser recebido por um outro fluxo para transformar a informação binária num formato adequado. Para esta função de conversão de bytes em formatos complexos (e vice-versa) existem as classes `DataInputStream` e `DataOutputStream`.



No diagrama anterior observamos um fluxo `DataInputStream` receber bytes de um outro fluxo ligado a um ficheiro binário e traduzi-los (por exemplo) em números do tipo `float`. Já no diagrama seguinte, o fluxo `DataOutputStream` transforma objectos do tipo `string` em bytes e transfere-os para um fluxo básico que os escreverá num ficheiro binário.



## Os Fluxos de Conversão de Dados

Um fluxo do tipo *DataInputStream* lê informação binária de um fluxo básico de leitura e converte esses bytes em qualquer tipo primitivo, *strings* ou vectores de caracteres. Já um fluxo *DataOutputStream* faz o inverso: converte tipos de dados em sequências de bytes que “alimentam” um fluxo básico de escrita.

Alguns métodos da classe *DataInputStream*:

- `boolean readBoolean()` – lê um valor booleano.
- `byte readByte()` – lê um valor do tipo `byte`.
- `char readChar()` – lê um carácter.
- `int readInt()` – lê um valor inteiro.
- `double readDouble()` – lê um valor real do tipo `double`.
- `void readFully(byte[] vector)` – preenche o vector com uma sequência de bytes lida.
- `void readFully(byte[] vector, int i, int n)` – preenche entre as posições `vector[i]` e `vector[i+n-1]`.
- `void skipBytes(int n)` – avança `n` bytes sem os armazenar.

Alguns dos métodos usados na classe *DataOutputStream*:

- `void writeBoolean(boolean b)` – escreve o booleano `b`. O valor verdadeiro é escrito como o byte 1 e falso como o byte 0.
- `void writeByte(int b)` – escreve os primeiros 8 bits de `b`.
- `void writeChar(int c)` – escreve os primeiros 16 bits de `c`.
- `void writeInt(int i)` – escreve o valor do inteiro `i`.
- `void writeDouble(double d)` – escreve o valor do `double` `d`.
- `void writeBytes(String s)` – escreve `s` como uma sequência de bytes.
- `void writeChars(String s)` – escreve `s` como uma sequência de caracteres.
- `void write(byte[] vector, int i, int n)` – escreve as posições entre `vector[i]` e `vector[i+n-1]`.
- `int size()` – devolve o número de bytes que já escreveu até agora.

No exemplo seguinte é aberto inicialmente um fluxo binário de escrita e associado a um arquivo de nome igual ao primeiro argumento de execução do programa. Este fluxo binário é então canalizado para um fluxo de dados, no qual escrevemos um real do tipo `double`.

```
import java.io.*;
class EscreverReais {
    public static void main(String[] args)
        throws IOException {
        DataOutputStream dados =
            new DataOutputStream(
                new FileOutputStream(args[0])
            );
        dados.writeDouble(1.23e-4);
        dados.close();
    }
}
```

O real `1.23e-4` é armazenado no ficheiro em formato binário (se o ficheiro for aberto com um editor de texto, o conteúdo parecerá estranho). Dado que só contém um `double`, a dimensão do ficheiro é igual a 8 bytes.

```
java EscreverReais resultado
[resultado] ? 1ôñÒF
```

Um programa para ler o real deste ficheiro:

```
import java.io.*;
class LerReais {
    public static void main(String[] args)
        throws IOException {
        DataInputStream f =
            new DataInputStream(
                new FileInputStream(args[0])
            );
        System.out.print("Real: " + f.readDouble());
    }
}
[resultado] ? 1ôñÒF
java LerReais resultado
Real: 1.23E-4
```

## Os Fluxos de Memória Intermédia

Estes fluxos mantêm um vector de caracteres no qual simulam a existência de memória intermédia. O valor padrão para a dimensão do vector é 2048 bytes. A primeira vez que é realizada uma leitura, o vector é totalmente preenchido (se o ficheiro for suficientemente grande) e cada leitura esvazia parte do vector. Quando este está vazio são lidos outros 2048

bytes. Este processo é muito eficaz quando ler um byte custa tanto como ler um certo conjunto de bytes.

Existem quatro classes, uma para cada tipo de fluxo:

- *BufferedInputStream* – a classe que lê fluxos binários de informação.
- *BufferedOutputStream* – a classe que escreve fluxos binários de informação.
- *BufferedReader* – a classe que lê fluxos de texto.
- *BufferedWriter* – a classe que escreve fluxos de texto.

Os construtores da classe *BufferedInputStream*:

- *BufferedInputStream(InputStream origem)* – liga este fluxo ao fluxo *origem*, criando uma memória intermédia de 2048 bytes.
- *BufferedInputStream(InputStream origem, int n)* – liga este fluxo ao fluxo *origem*, com uma memória intermédia de *n* bytes.

```
import java.io.*;
class ContarBytes {
    public static void main(String[] args)
                    throws IOException {
        int nBytes = 0;
        BufferedInputStream f =
            new BufferedInputStream(
                new FileInputStream(args[0]))
        ;
        while (f.read() != -1)
            nBytes++;
        System.out.println("Número = " + nBytes);
        f.close();
    }
}
```

Em relação ao exemplo do fluxo *FileInputStream*, a única diferença são as instruções destacadas. Em vez de aceder directamente ao ficheiro, foram lidos até 2048 bytes para a memória intermédia de uma só vez, de modo a tornar o acesso mais eficiente.

Os construtores da classe *BufferedOutputStream*:

- *BufferedOutputStream(OutputStream destino)* – liga este fluxo ao fluxo *destino*, criando uma memória intermédia de 512 bytes.
- *BufferedOutputStream(OutputStream destino, int n)* – liga este fluxo ao fluxo *destino*, com uma memória intermédia de *n* bytes.

Os construtores da classe *BufferedReader*:

- *BufferedReader(Reader origem)* – liga este fluxo ao fluxo *origem*, criando uma memória intermédia de 8192 caracteres.
- *BufferedReader(Reader origem, int n)* – liga este fluxo ao fluxo *origem*, com uma memória intermédia de *n* caracteres.

Esta classe já possui o método *readLine()* que lê uma linha de texto e devolve uma referência para a *string* contendo essa linha (devolve *null* se chegar ao fim do ficheiro).

```
import java.io.*;
public class LerLinhas {
    public static void main(String[] args)
                    throws IOException {
        String s;
        BufferedReader f =
            new BufferedReader( new FileReader("teste.txt") );
        while ((s=f.readLine()) != null)
            System.out.println(s);
        f.close();
    }
}
```

Os construtores da classe *BufferedWriter*:

- *BufferedWriter(Writer destino)* – liga este fluxo ao fluxo *destino*, criando uma memória intermédia de 256 caracteres.
- *BufferedWriter(Writer destino, int n)* – liga este fluxo ao fluxo *destino*, com uma memória intermédia de *n* caracteres.

```
import java.io.*;
public class EscreverLinhas {
    public static void main(String[] args)
                    throws IOException {
        BufferedWriter f =
            new BufferedWriter( new FileWriter("teste.txt") );
        f.write("uma linha escrita\n");
        f.write("e outra\n");
        f.close();
    }
}
```

Na maioria das aplicações que lidam com ficheiros deve-se utilizar estas classes de fluxo de memória intermédia por motivos de eficiência.

## 7.4 Origem e Destino da Informação

### Os Fluxos Predefinidos

Existem alguns fluxos predefinidos: `System.in`, `System.out` e `System.err`. Estes fluxos são considerados fluxos binários apesar de transferirem caracteres (foram criados numa versão do Java onde não existiam ainda os fluxos de texto). O `System.in` lê informação proveniente do utilizador (normalmente do teclado). O `System.out` escreve informação directamente para o utilizador (normalmente para o monitor). O `System.err` é um canal especial para transmissão de mensagens de erro.

Um exemplo:

```
import java.io.*;
class LerTeclado {
    public static void main(String[] args)
        throws IOException {
        BufferedReader teclado =
            new BufferedReader(
                new InputStreamReader(System.in));
        String s = teclado.readLine();
        System.out.print("Frase lida: \"'" + s + "'\"");
    }
}
$ java LerTeclado
$ OLÁ MUNDO
$ Frase lida: "OLÁ MUNDO"←
```

Do mesmo modo é possível associar um fluxo de saída para o monitor:

```
import java.io.*;
class EscreverMonitor {
    public static void main(String[] args)
        throws IOException {
        BufferedWriter monitor =
            new BufferedWriter(
                new OutputStreamWriter(System.out));
        monitor.write("olá mundo");
        monitor.flush();
    }
}
```

```
java EscreverMonitor
olá mundo
```

Neste último exemplo é mais fácil utilizar o método `print()` do fluxo `System.out` (neste exemplo, invocar directamente `System.out.print("olá mundo");`).

## A Classe Ficheiro

Os ficheiros armazenam informação de forma persistente, i.e., informação que permanece armazenada mesmo se o computador for desligado (ao contrário da memória primária que também se designa memória volátil). O custo temporal do acesso justifica-se pela propriedade de permanência.

Existem duas formas padrão de acesso (seja de escrita ou de leitura) a um dado ficheiro: o **acesso sequencial** e o **acesso dinâmico** (do inglês, *random access*). O acesso sequencial é limitado por uma ordem sequencial de acesso. A informação seguinte é colocada à frente da informação anterior – não é possível voltar atrás num ficheiro sequencial (a não ser fechá-lo e abri-lo de novo). O acesso dinâmico permite uma maior liberdade, pois é possível avançar e recuar de forma arbitrária. Existe uma classe para tratar de cada caso, nomeadamente `File` e `RandomAccessFile`.

A classe `File` define e manipula ficheiros e directorias: inclui a abertura e fecho de ficheiros, lista os ficheiros de uma directoria, apaga ficheiros e directorias, etc. Para uma explicação completa das numerosas funcionalidades desta classe consultar a bibliografia.

A classe tem os seguintes construtores:

- `File(String nome)` – associa o objecto ao ficheiro dado por *nome*.
- `File(String caminho, String nome)` – associa o objecto ao ficheiro dado por *nome* na directoria *caminho*.

O objecto devolvido – da classe `File` – deve ser associado a um fluxo de dados apropriado. Veja-se o exemplo seguinte:

```
import java.io.*;
class Sequencial {
    public static void main(String[] args)
        throws IOException {
        int c;
        File dados = new File("../","info");
        FileInputStream f = new FileInputStream(dados);
        while ((c=f.read()) != -1)
            System.out.print((char)c + ":");
    }
}
```

```
📁 [..\info] ABCDEF
📝 java Sequencial
☐ A:B:C:D:E:F:
```

A classe `RandomAccessFile` permite o acesso dinâmico a ficheiros binários. O construtor tem o seguinte formato:

- `RandomAccessFile(File ficheiro, String modo)` – associa o objecto ficheiro um acesso dinâmico de leitura (`modo` igual a “r”), de escrita (`modo` igual a “w”) ou de leitura/escrita (`modo` igual a “rw”).

Alguns dos métodos usados:

- `void seek(long n)` – prepara a leitura e/ou escrita do  $(n+1)$ -ésimo byte (por convenção, o primeiro byte encontra-se na posição zero). O valor `n` deve ser maior ou igual a zero.
- `int read()` – lê um valor inteiro.
- `int read(byte[] vector)` – preenche o vector com uma sequência de bytes lida. Se não existirem bytes suficientes, o método bloqueia.
- `int read(byte[] vector, int i, int n)` – preenche entre as posições `vector[i]` e `vector[i+n-1]`. Se não existirem bytes suficientes, o método bloqueia.
- `void write(int i)` – escreve o inteiro `i`.
- `void write(byte[] vector)` – escreve `vector`.
- `void write(byte[] vector, int i, int n)` – escreve as posições entre `vector[i]` e `vector[i+n-1]`.

Existem ainda métodos para ler e escrever nos diversos tipos de informação. Estes são idênticos aos métodos das classes `DataInputStream` e `DataOutputStream`.

No exemplo é aberto um ficheiro de acesso dinâmico para leitura e escrita simultânea:

```
import java.io.*;
class Dinamico {
    public static void main(String[] args)
        throws IOException {
        RandomAccessFile f =
            new RandomAccessFile(new File("info"), "rw");
        for(int i=65;i<91;i++)
            f.write(i);           // escreve o alfabeto
        byte[] ler = new byte[10];
        f.seek(4);             // salta para o 5º byte
        f.read(ler, 0, 5);     // lê as próximas 5 posições
```

```
    for(int i=0;i<5;i++)
        System.out.print((char)ler[i] + ":");

        f.seek(3);           // salta para o 4º byte
        System.out.print((char)f.read());
    }
}


```

## 7.5 Usando a classe Scanner

Dado um ficheiro de texto com diversos tipos de informação (eventualmente numa mesma linha), como aceder a esses diferentes valores? É possível utilizar a classe `Scanner` do pacote `java.util` para este efeito. Esta classe permite ler sequências de caracteres (em inglês, *tokens*) delimitados por um conjunto de separadores.

Alguns métodos:

- `int nextInt()` – transforma o próximo *token* num número inteiro.
- `boolean hasNextInt()` – verifica se o próximo *token* é um inteiro.
- `String next()` – devolve o próximo *token* como *string*.
- `Scanner useDelimiter(...)` – define quais os separadores entre *tokens*.

Um exemplo do uso destes métodos:

```
String s = "Pensar globalmente, agir
localmente. Quando? Sempre!";

Scanner sc =
new Scanner(s).useDelimiter("\\s*[ ,.]\\s*");
```

Os delimitadores associados à *string* `s` (neste caso, o espaço, a vírgula e o ponto) são incluídos através da invocação do método `useDelimiter()`.

```
while (sc.hasNext())
    System.out.print(sc.next() + "/");
```

Enquanto existirem *tokens* na frase, o programa imprime-os.

 Pensar/globalmente/agir/localmente/Quando?/Sempre!/

Vejamos um exemplo completo de como aceder à informação de um ficheiro de texto. O método lê cada linha do ficheiro para uma *string* e aplica o mesmo esquema de forma a recolher os vários *tokens*.

Considere o ficheiro de texto `dados.txt` com a seguinte estrutura: numa primeira linha possui o número de registo que se seguem e cada uma das linhas seguintes contém um registo de uma pessoa (número, nome, sexo e se é casada – tudo separado por espaços). Um conteúdo possível seria:

```
dados.txt
4 ↴
102  joão  M s ↴
302  maria F n ↴
1021 ana   F s ↴
4302 paulo M n ↴
```

Considere a classe `Pessoa` que armazena a informação de um registo.

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

public class Pessoa {
    long id;
    String nome;
    char sexo;
    boolean éCasado;

    public Pessoa(long id, String nome,
                  char sexo, boolean éCasado) {
        this.id = id;
        this.nome = nome;
        this.sexo = sexo;
        this.éCasado = éCasado;
    }

    public String toString() {
        return "[id:" + id + ", nome:" + nome + ", "+
               (sexo=='M'?"masculino":"feminino") + " e "+
               (éCasado?"casado":"solteiro") + "]";
    }
}
```

O método `toString` determina como a informação é apresentada no monitor.

```
public static void main(String[] args)
                        throws IOException {
    Pessoa[] p;
    int nPessoas;
    Scanner sc =
        new Scanner( new FileReader("dados.txt") );
```

```
try {
    nPessoas = sc.nextInt();
    p = new Pessoa[nPessoas];
    for (int i=0; i<nPessoas; i++)
        p[i] = new Pessoa(
            sc.nextLong(),
            sc.next(),
            sc.next().charAt(0),
            sc.next().charAt(0) == 's' );
}
```

Para cada frase lida do ficheiro são retiradas as quatro informações distintas (número, nome, sexo e se é casada) e invocado o construtor para criar um novo objecto.

```
} //try
finally {
    sc.close();
```

Aconteça o que acontecer, o ficheiro de dados deve ser fechado.

```
}
```

```
for (int i=0; i<nPessoas; i++)
    System.out.println(p[i]);
```

Neste último ciclo mostram-se os valores lidos do ficheiro.

```
} //main()
} //endClasse Pessoa
```

□ [id:102, nome:joão, masculino e casado] ↵  
[id:302, nome:maria, feminino e solteiro] ↵  
[id:1021, nome:ana, feminino e casado] ↵  
[id:4302, nome:paulo, masculino e solteiro] ↵

## Exercícios

1. Escreva um programa que leia números inteiros do teclado e os escreva num ficheiro de texto, até ser digitado o valor zero.
2. Escreva um programa que leia um ficheiro binário e determine o número de ocorrências de 0x00FF.
3. Abra um ficheiro de texto e escreva um novo ficheiro no qual cada vogal é substituída pela vogal seguinte (i.e., um ‘a’ passa a ‘e’, um ‘e’ a ‘i’, ... e um ‘u’ a ‘a’). Por exemplo:

 [in] ontem fomos ao cinema!  
 [res] untim fumus eu conime!

4. Escreva um programa que leia dois ficheiros de texto contendo números positivos ordenados de forma crescente e crie um novo ficheiro de texto com o conteúdo dos dois ficheiros iniciais e mantendo a ordem crescente. Por exemplo:

 [in1] 1 3 7 12 43 156  
 [in2] 1 2 4 6 8 88  
 [res] 1 1 2 3 4 6 7 8 12 43 88 156

5. Escreva um programa que leia um ficheiro de texto contendo números e crie dois novos ficheiros de texto, um deles com os números pares e o outro com os números ímpares. Por exemplo:

 [in] 1 4 14 15 3 12 1 23  
 [res-impar] 1 15 3 1 23  
 [res-par] 4 14 12

6. Escreva um programa que leia um ficheiro de texto contendo apenas letras que detecte sequências de dimensão dois ou maior da mesma letra e que escreva um novo ficheiro de texto em que essas sequências são substituídas pela letra e pelo número de vezes que ocorre. Por exemplo:

 [in] abcccdeaaaa  
 [res] abc3dea4

7. Abra um ficheiro de acesso aleatório e crie um novo ficheiro com o mesmo conteúdo do primeiro, mas de forma inversa (o primeiro byte do primeiro vai ficar no último byte do segundo).

 [in] abc4f  
 [res] f4cba

8. Considere a classe definida no exercício 27 do capítulo 4 (a classe que representa polinómios). (i) Acrescente os seguintes métodos:

```
/**  
 * Armazenar o polinómio em ficheiro.  
 * @param file identificador do ficheiro.  
 */  
public void save(DataOutputStream file)  
        throws IOException  
/**  
 * Ler o polinómio de ficheiro.  
 * @param file identificador do ficheiro.  
 */  
public void read(DataInputStream file)  
        throws IOException
```

Sugestão: comece por guardar o grau do polinómio. Caso ocorra algum erro na leitura do polinómio a partir do ficheiro, o polinómio deverá permanecer no mesmo estado em que se encontrava no início do método.

(ii) Acrescente ainda os seguintes métodos:

```
/**  
 * Armazenar o polinómio em ficheiro.  
 * @param file nome do ficheiro.  
 */  
public void save(String file) throws IOException  
/**  
 * Ler o polinómio de ficheiro.  
 * @param file nome do ficheiro.  
 */  
public void read(String file)  
        throws FileNotFoundException, IOException
```

Caso ocorra algum erro na leitura do polinómio a partir do ficheiro, o polinómio deverá permanecer no mesmo estado em que se encontrava no início da rotina.

(iii) Compare os métodos das alíneas (i) e (ii). Qual deles lhe parece preferível? Porquê?

9. Considere a classe Segment do exercício 21 capítulo 4. Assuma que a classe Point que usar já possui os métodos **public** void save(DataOutputStream file) **throws** IOException e **public** void read(DataInputStream file) **throws** IOException. Escreva estes métodos para a classe Segment.



# PARTE II

## Desenvolvimento de Aplicações



# **8 – Especificação**

---

A programação, enquanto actividade intelectual, é vista de diferentes formas: como uma actividade lúdica (é divertido programar), como uma ferramenta de investigação (por exemplo, o uso dos computadores é essencial para efectuar simulações de acontecimentos para os quais não há recursos para testar em ambientes físicos) ou comercial (se o contexto económico o justificar).

Como na maioria das actividades humanas, a Informática é capaz de disponibilizar um conjunto de serviços. A entidade (individual ou colectiva) que pretende um desses serviços nem sempre tem o conhecimento necessário para o realizar, tendo que o pedir a outra entidade. Neste contexto, surge uma ligação entre quem pede um serviço – o **cliente** – e quem providencia esse mesmo serviço – o **fornecedor**.

Em certas situações o acordo, ou contrato, entre as partes pode ser definido verbalmente (por exemplo, entre elementos da mesma equipa de trabalho). Mas existem situações onde a informalidade não é conveniente. Se uma instituição paga milhões de euros a uma empresa por um projecto informático é necessário indicar com toda a precisão possível quais os compromissos de ambas as partes. Usualmente, os requisitos do problema são definidos por um caderno de encargos apresentado pelo cliente que o disponibiliza num concurso público. A esse caderno de encargos, os potenciais fornecedores respondem dentro do prazo com uma proposta (que inclui orçamento, cronograma de actividades...). Escolhida a proposta (e o fornecedor) pelo cliente será especificado o contrato que os relaciona legalmente.

A necessidade de especificar o contrato tem justificações concretas:

- É importante assegurar que os engenheiros e programadores que irão cumprir o contrato saibam o suficiente para produzirem exactamente o pedido. Uma especificação errada implica num desenvolvimento errado, que custa tanto mais quanto mais tarde se descobrir o problema. É preciso muita atenção durante a fase de especificação de modo a estabelecer os fundamentos de um projecto de sucesso. Por exemplo, não se pode deixar um construtor civil começar a construir um prédio para depois se dizer que, afinal, são mais uns cinco andares – a especificação requer a planta detalhada da obra.
- O objectivo principal da especificação é conseguir realizar o projecto com o menor custo/esforço possível. Uma má especificação pode dar uma liberdade excessiva a um cliente ou a um fornecedor mal intencionado. Muitos custos são consequência de especificações incoerentes ou incompletas depois do contrato assinado. Permite que o fornecedor escolha o melhor caminho, o que nem sempre é igual (nunca é) ao caminho desejado pelo cliente. Permite ao cliente exigir do fornecedor requisitos à posteriori que não tinham sido combinados previamente na assinatura do contrato. Há quadros de empresas cuja especialização é “espremer” o contrato, i.e., descobrir falhas nas especificações fornecidas de forma a maximizar o potencial ganho sem incorrer em quebras formais do assinado.
- Com uma especificação que foque todos os aspectos relevantes, de forma clara e modular, é possível exigir um orçamento igualmente claro sendo mais fácil determinar a correcção e fiabilidade. Qualquer elemento em falta aumenta a potencial derrapagem de um orçamento sem a protecção legal que esse elemento em falta traria se mencionado no contrato. E, claro, qualquer extensão do contrato implica maiores gastos.
- Mesmo durante a fase de concurso, quanto menos ambíguo for a especificação do pedido maior é a justiça para os concorrentes. Existe uma menor dependência sobre factos privados que uns concorrentes conhecem e outros não. Evita posteriores problemas legais (leia-se, muito tempo e dinheiro) com os concorrentes que perderam. Para garantir a imparcialidade de uma especificação é preciso conhecer as restrições legais que existem sobre os contratos (por exemplo, não se pode determinar o projecto do produto, nem requerer serviços a empresas específicas para evitar preferências ou limitações técnicas sobre os recursos necessários à execução do trabalho).
- Uma especificação clara distribui responsabilidades pelos dois lados da negociação. Se algum problema grave ocorrer (como a morte de alguém) um tribunal deve ser capaz de apontar quem deve pagar as indemnizações devidas. Este é um assunto caro às seguradoras, como não é difícil de imaginar.

Assim, uma especificação é um conjunto de regras e restrições que define não só o produto desejado, mas define como e quando o projecto de desenvolvimento e manutenção deve ocorrer. Normalmente, faz-se a diferença entre o produto desenvolvido e o serviço

associado de manutenção (que inclui a formação do utilizadores bem como a assistência do produto), o que significa dois contratos separados (é boa política não misturar produtos com serviços). Quando um contrato é assinado e reconhecido, a negociação entre cliente e fornecedor termina. O fornecedor é obrigado a entregar o produto ou serviço especificado no prazo estabelecido. O cliente é obrigado a recompensá-lo quando for devido.

Outros pontos importantes:

- Existem duas formas de especificação: pelo desempenho ou pelo desenho. Na especificação por desempenho há uma preocupação pelos recursos necessários (por exemplo, quanta memória ou qual o processador mínimo para executar a aplicação) e pelos resultados garantidos (qual o tempo mínimo de resposta do sistema perante um dado conjunto de problemas pelo sistema). Na especificação por desenho o foco está centrado na estrutura e configuração geral do produto. As especificações devem ser centradas no desempenho e não no desenho. A forma como o fornecedor desenha e implementa o produto especificado é da sua responsabilidade. Se um contrato se baseia simplesmente no desenho, as questões de desempenho são problema do cliente e não do fornecedor. Implica, muito provavelmente, uma posterior reformulação contratual que custará dinheiro.
- A especificação de cada tarefa deve ser o mais modular possível e deve ser acompanhada por mecanismos de verificação de qualidade. Assim, não só se descreve o pretendido, como se define a correção da tarefa (onde se inclui o exigido e garantido pela tarefa). Esta definição da esfera de responsabilidades ajuda a eliminar ambiguidades e a decidir disputas sobre quem tem o encargo dos eventuais problemas.
- A especificação estabelece a tolerância do produto: qual a robustez da aplicação perante situações anómalias. É essencial definir e contratualizar a segurança do produto e as garantias oferecidas pelo fornecedor em casos de construção incorrecta. Do ponto de vista do fornecedor, as garantias são associadas a um contexto limitado de aplicabilidade (como a empresa de electrodomésticos processada por uma senhora que colocou o gato molhado dentro do microondas para o secar...).
- Especificações incoerentes contêm partes que quando vistas em conjunto são impossíveis de solucionar. Este tipo de problemas ocorre mais facilmente quando a especificação é criada por grupos relativamente independentes. Pode significar problemas de difícil solução se não houver boa vontade das duas partes. Este é, principalmente, um problema do fornecedor (a não ser que consiga convencer a Justiça da impossibilidade técnica do contrato libertando-se das suas obrigações).

Ao receber os requisitos do cliente, o fornecedor pode iniciar a fase de **análise**, onde se definem os componentes necessários e as relações entre os mesmos. A identificação dos componentes “atómicos” da solução é um processo de modularização que controla a complexidade do sistema dividindo-o em partes menores. A identificação das relações

entre esses componentes mostra quais as funcionalidades e dependências. É no processo de interacção destes elementos que emerge o comportamento global do sistema. A optimização deste processo não é uma tarefa trivial.

## Que Linguagem?

Num mundo perfeito, o contrato ideal (apoiado pela Justiça ideal) seria a melhor forma de resolver qualquer problema inesperado. No entanto, é presumido que o contrato é uma expressão de um compromisso não entendido exactamente da mesma forma pelo cliente e pelo fornecedor. Porquê? Primeiro, por vezes não se sabe tudo o necessário antes o projecto começar (dado um problema inesperado, ou faltaram estudos para o detectar ou simplesmente não era possível prever o que aconteceu). E segundo, porque a interpretação do contrato depende de dois pontos de vista antagónicos (o cliente quer pagar o menos possível e retirar o máximo benefício, o fornecedor quer o maior lucro possível com o mínimo esforço). Esta interpretação é diferente porque a linguagem onde o contrato está descrito é ambígua. Todas as linguagens naturais (português, inglês, chinês) possuem uma ambiguidade inerente. É necessário para o nosso dia-a-dia<sup>12</sup> ou para a literatura mas pode ser prejudicial para a investigação científica e para o mundo de negócios.

Por muito que se mantenham as frases simples (especifique, não explique), se reduza o vocabulário a um mínimo técnico aceitável (nada de sinónimos ou figuras de estilo, nunca usar reticências ou etc.) e se limite ao uso de um pequeno conjunto de construções gramaticais consideradas seguras é quase impossível não deixar alguma ambiguidade entrar na descrição do contrato. Algumas frases não afectam o significado do que se quer transmitir (as frases correctas), outras tornam o texto quase ininteligível (as incorrecções menos perigosas) enquanto outras alteram subtilmente o significado do que se queria dizer (as mais perigosas).

As ambiguidades possuem diferentes dimensões. Desde palavras e frases ambíguas (ao dizer “*pretende-se métodos de leitura e escrita seguros sobre ficheiros*” queremos dizer métodos de leitura normais e métodos de escrita seguros – a interpretação mais barata – ou métodos de leitura seguros e métodos de escrita seguros – a interpretação mais cara?) até à ambiguidade contextual (“*a documentação sobre os métodos deverá ser escrita em HTML*” significa que a documentação é para ser escrita pelo fornecedor? Pelo cliente? Faz parte do contrato ou é para alguém fazer depois?). Neste contexto, nenhuma ambiguidade é bem vindas.

---

<sup>12</sup> Existe alguma vantagem numa língua ser ambígua? Uma vantagem é a redução do léxico. Devido à enorme complexidade da realidade seria impossível possuirmos um termo para cada acontecimento diferente. Ao agregar num único termo uma multiplicidade de conceitos (que do nosso ponto de vista cultural são mais ou menos similares) ganhamos a capacidade de expressar (e, eventualmente, entender) a complexidade que nos rodeia.

Felizmente, os processos computacionais podem ser formalmente expressos sem o uso do português. A Matemática possui um conceito apropriado para descrever computações: a função. Por exemplo, para especificar um método que calcula o sucessor de um número real:

```
sucessor : real → real
  ∀R ∈ real : sucessor(R) = R + 1.0
```

Significado: é descrita uma função *sucessor* com uma assinatura cujo conjunto de partida e de chegada é o tipo *real*. Se aplicarmos à função *sucessor* um qualquer valor *R* que pertença ao tipo *real*, a função calcula *R+1*. Não há ambiguidade e existe ainda outra vantagem: uma função não possui efeitos secundários – uma propriedade útil que promove o aspecto modular de um sistema<sup>13</sup>.

Vejamos um exemplo de um tipo *ponto2D*. A configuração dos elementos deste tipo é o produto cartesiano de dois reais (as coordenadas de um ponto a duas dimensões). Considere que um ponto nas coordenadas (*x,y*) é representado por  $\langle x, y \rangle$ . Segue a especificação de um conjunto de funções relacionadas:

```
novo : real real → ponto2D
  ∀x,y ∈ real : novo(x,y) = ⟨x,y⟩
éOrigem : ponto2D → boolean
  ∀⟨x,y⟩ ∈ Ponto2D : éOrigem(⟨x,y⟩) = x=0 ∧ y=0
getX : ponto2D → real
  ∀⟨x,y⟩ ∈ Ponto2D : getX(⟨x,y⟩) = x
distOrigem : ponto2D → real
  ∀⟨x,y⟩ ∈ Ponto2D : distOrigem(⟨x,y⟩) = √x²+y²
setY : ponto2D real → ponto2D
  ∀⟨x,y⟩ ∈ Ponto2D,
  ∀r ∈ real : setY(⟨x,y⟩,r) = ⟨x,r⟩
multiplicar : ponto2D real → ponto2D
  ∀⟨x,y⟩ ∈ Ponto2D,
  ∀r ∈ real : multiplicar(⟨x,y⟩,r) = ⟨x*r,y*r⟩
```

<sup>13</sup> De facto, esta forma de olhar para as computações é responsável pelo desenvolvimento da programação funcional – cuja base teórica é o modelo de computação denominado cálculo  $\lambda$  – do qual se originaram várias linguagens de programação como o LISP, o ML ou o Haskell.

## 8.1 Tipo de Dados Abstracto

Vamos definir novos tipos de dados, descrevendo os seus componentes da forma mais abstracta possível (sem qualquer compromisso de implementação). A esta descrição abstracta (desenvolvida por C. Hoare nos anos 70) denominaremos por **tipo de dados abstracto**, ou simplesmente, TDA. Usaremos os TDA na 3<sup>a</sup> parte para especificar estruturas de dados.

Um TDA é uma descrição formal composta por cinco partes:

- Cabeçalho – onde se define o nome da especificação e o conjunto de especificações necessárias para defini-la.
- Géneros – onde se enumera os zero ou mais conjuntos de elementos que se pretendem especificar.
- Operações – onde se descreve as assinaturas das funções dos tipos em questão.
- Axiomas – onde se descreve a semântica das funções referidas na parte anterior. Cada axioma refere uma propriedade a respeitar pelas operações relacionadas.
- Pré-condições – onde são expressas as restrições que existem nas funções, definindo os respectivos domínios de aplicação (já que estabelecem quais as configurações válidas e inválidas).

A estrutura será:

```
especificação <nome da especificação> =
    importa <especificações usadas>
    géneros ...
    operações ...
    axiomas ...
    pré-condições ...
fim-especificação
```

A sintaxe e a gramática usadas foram convencionadas. O importante é ser num formato comum entendido por quem escreve a especificação e por quem a interpreta<sup>14</sup>. Enquanto as três primeiras partes descrevem o tipo e as funções associadas, os axiomas e as pré-condições definem a semântica associada ao TDA.

Reutilizando o exemplo do ponto a duas dimensões, a especificação seria a seguinte:

```
especificação Ponto2D =
    importa Real, Boolean
    géneros
        Ponto2D
```

---

<sup>14</sup> Como garantir isso? Especificando a especificação? Este problema só tem solução se existir um nível básico de interpretação partilhado por ambas as partes. Entre duas entidades com contextos culturais totalmente disjuntos, a comunicação não seria possível.

**operações**

novo	:	Real	Real	→	Ponto2D
getX	:	Ponto2D	→	Real	
getY	:	Ponto2D	→	Real	
setX	:	Ponto2D	Real	→	Ponto2D
setY	:	Ponto2D	Real	→	Ponto2D
éOrigem	:	Ponto2D	→	Boolean	
distOrigem	:	Ponto2D	→	Real	
multiplicar	:	Ponto2D	Real	→	Ponto2D
dividir	:	Ponto2D	Real	→	Ponto2D
éIgual	:	Ponto2D	Ponto2D	→	Boolean

**axiomas**

getX(novo(x,y))	=	x
getY(novo(x,y))	=	y
setX(P,r)	=	novo(r,getY(P))
setY(P,r)	=	novo(getX(P),r)
éOrigem(P)	=	getX(P)=0.0 and getY(P)=0.0
distOrigem(P)	=	$\sqrt{getX(P)^2+getY(P)^2}$
multiplicar(P,r)	=	novo(getX(P)*r,getY(P)*r)
dividir(P,r)	=	novo(getX(P)/r, getY(P)/r)
éIgual(P1,P2)	=	getX(P1)=getX(P2) and getY(P1)=getY(P2)

**pré-condições**

dividir(P,r) **requer** r≠0

**fim-especificação**

Alguns pontos sobre este exemplo:

- No cabeçalho, assumiu-se a existência de duas especificações, uma que especifica o tipo real e outra que especifica o tipo booleano.
- A função novo(x,y) tem um comportamento especial: devolve elementos do tipo ponto a partir de argumentos reais. É através dela que podemos definir qualquer ponto bidimensional. Este género de função denomina-se por **construtor**.
- A descrição <x,y> da secção anterior foi redefinida pelo uso do construtor novo(x,y). Assim, um ponto nas coordenadas (1.5,2.0) é descrito pela expressão novo(1.5,2.0). Neste caso, não é necessário adicionar axiomas para definir a operação construtor. Ela define-se implicitamente nos axiomas das outras operações.
- Seja uma especificação com dois ou mais construtores. Para definir recursivamente o conjunto de elementos associado ao tipo, pode ser necessário o uso de axiomas para os construtores. Quando? Quando o conjunto de elementos definido pela

aplicação indutiva desses construtores inclui elementos que não pertencem ao tipo. Considere o exemplo seguinte de especificação booleana em que os construtores são as operações FALSE e not:

```
especificação Boolean =  
géneros  
    Boolean  
operações  
    FALSE : → Boolean  
    not   : Boolean → Boolean  
    TRUE  : → Boolean  
    and   : Boolean → Boolean  
    or    : Boolean → Boolean  
    equals: Boolean Boolean → Boolean  
axiomas  
    not(not(FALSE)) = FALSE
```

Encontramos um axioma para um construtor. Porquê? Porque a aplicação recursiva dos construtores sem restrições definiria o conjunto infinito {FALSE, not(FALSE), not(not(FALSE)), not(not(not(FALSE)))), ... }, quando o conjunto pretendido é {FALSE, TRUE}. Isto é conseguido através do axioma anterior, que reduz o conjunto infinito ao conjunto {FALSE, not(FALSE)}.

```
TRUE = not(FALSE)  
and(FALSE,A) = FALSE  
and(TRUE,A) = A  
or(TRUE,A) = TRUE  
or(FALSE,A) = A  
equals(TRUE,TRUE) = TRUE  
equals(TRUE,FALSE) = FALSE  
equals(FALSE,TRUE) = FALSE  
equals(FALSE,FALSE) = TRUE
```

**pré-condições**  
**fim-especificação**

Resta dizer que, por motivos de clareza, optou-se por representar as operações lógicas de forma infixa, i.e., usaremos  $x$  and  $y$  em vez de  $\text{and}(x, y)$ . Também usaremos  $A = B$  em vez de  $\text{equals}(A, B)$ .

- Esta forma de **descrição implícita** dos elementos pelos construtores evita o uso de representações específicas com detalhes inúteis e mesmo prejudiciais para o nível

de abstracção desejado. Uma descrição explícita seria, por exemplo, descrito pelo axioma  $\text{setX}(P) = P[0]$ , indicando que o tipo Ponto2D seria obrigatoriamente representado por um vector.

- No caso da operação `dividir`, a existência de uma pré-condição (a divisão por zero não está definida) indica que o funcionamento está restrito a um determinado subconjunto dos reais, tornando-a uma **função parcial**<sup>15</sup>. É indicado nas assinaturas pelo símbolo  $\nrightarrow$ .
- Se for necessário criar alguma função auxiliar para definir uma das funções do TDA, inclui-se a subsecção auxiliares na secção operações onde se inserem as funções auxiliares. Por convenção prefixamos estas funções com um sublinhado “\_” para melhor distinguir das funções principais.

Um **tipo genérico** T é um tipo que contém na sua configuração elementos de outros tipos mas que a semântica das operações associadas a T são independentes das especificações desses tipos (por exemplo, as funções de união ou intersecção de conjuntos são idênticas, caso se trate de conjuntos de inteiros, de reais ou de outro tipo qualquer). Os tipos genéricos são representados da seguinte forma<sup>16</sup>:

**especificação** Conjunto<Element> = ...

Uma descrição deste género, mais do que uma especificação, é um **padrão de especificações** dado que a descrição final é parametrizada por *Element*. Na 3<sup>a</sup> parte, todas as estruturas de dados são definidas de forma genérica, já que o seu funcionamento é independente da informação armazenada.

Um exemplo de um tipo de dados abstracto é a noção de variável. A variável é uma abstracção com atributos como o nome, o endereço, o valor, o âmbito, o tipo e a sua dimensão. A declaração, a consulta e a atribuição são operações associadas a variáveis, a primeira sendo um construtor, a segunda uma interrogação, a terceira um comando. De facto, não é necessário conhecer os detalhes de implementação para utilizar variáveis na construção de programas, apenas é preciso conhecer a sua configuração e as operações disponibilizadas.

---

<sup>15</sup> Uma função parcial é uma função cujo domínio não é igual ao conjunto de partida definido na sua assinatura. Por exemplo, sejam as assinaturas  $f:\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$  e  $g:\mathbb{R} \times \mathbb{R} \setminus \{0\} \rightarrow \mathbb{R}$ . A função  $f(x,y)=x/y$  é parcial (o conjunto de partida inclui o zero que não pertence ao domínio) e a função  $g(x,y)=x/y$  é total (o conjunto de partida e o domínio são iguais).

<sup>16</sup> Eventualmente, esta noção pode ser usada recursivamente. Para o caso de implementar um conjunto de conjuntos, o elemento *Element* seria o próprio tipo Conjunto.

## Funcionalidade vs. Estrutura

Das secções anteriores podemos afirmar:

*A especificação é um método formal que descreve abstractamente o que o programa deve fazer para ser considerado correcto e não como deve ser feito.*

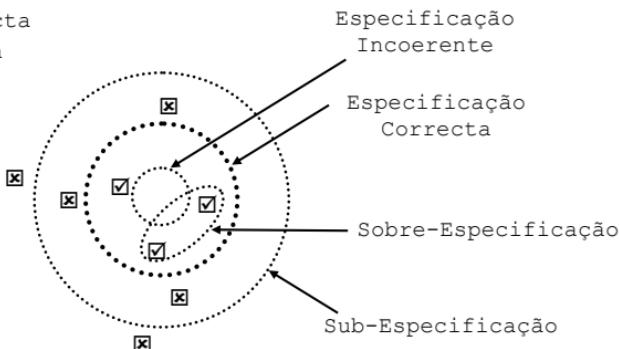
Do ponto de vista do programador, a especificação representa um conjunto de implementações possíveis (i.e., as funções descritas nos axiomas podem ser executadas por diferentes algoritmos). Ao separar o “o quê” do “como” focam-se os aspectos relevantes da especificação, mesmo sem qualquer informação sobre a linguagem a ser utilizada no desenvolvimento. Quem cria a especificação define implicitamente a dimensão desse conjunto de implementações, ficando ao cargo daquele que implementa escolher a mais adequada.

A noção de correcção de um programa não é absoluta, i.e., um programa não está simplesmente correcto ou errado. A correcção de um programa mede-se relativamente à especificação que este pretende seguir. Estabelece-se uma definição informal de programa correcto: um programa correcto descreve uma computação cuja semântica coincide à da especificação.

Construir uma especificação completa e correcta é algo bastante difícil. Entre os possíveis problemas encontramos:

- Uma **sobre-especificação** – uma especificação que se compromete com detalhes irrelevantes de implementação.
- Uma **especificação incoerente** representa um conjunto vazio de implementações. Estas especificações podem ser incorrectas (permitem e proíbem ao mesmo tempo um limitado conjunto de soluções) ou ambíguas (permitem mais do que um resultado correcto quando se pretende apenas um, ou seja, certas operações especificadas não são funções).
- Uma **sub-especificação** produz um conjunto demasiado extenso de implementações possíveis (onde se incluem algumas soluções erradas). Corre-se o risco de ser escolhida uma implementação inadequada.

- Implementação Correcta
- Implementação Errada



Considerando que a especificação E define um conjunto de implementações, uma implementação I satisfaz E se  $I \in E$ .

## 8.2 Desenho

Entre a especificação/análise e a implementação é comum existir um passo intermédio que ajuda a transformar o conjunto dos componentes especificados numa possível implementação. Um plano que mostra como o sistema proposto vai satisfazer o especificado. A esse processo denomina-se **desenho** (ou **projeto**). O desenho não é independente da especificação, o primeiro depende do segundo. Mas o desenho é independente da implementação. Um desenho correcto identifica um determinado conjunto de implementações da especificação com a vantagem de usar uma linguagem próxima das ferramentas de implementação. O compromisso não se situa ainda na forma como se deve fazer (continua a ser uma liberdade do processo de implementação) mas já existe uma especialização da linguagem utilizada. Deixamos a abstracção das funções matemáticas (ou qualquer outra abstracção usada na especificação) e passamos a comunicar numa linguagem mais próxima de quem implementa.

Como foi referido, uma especificação define formalmente o que está correcto. Um desenho D está correcto, em relação a uma especificação E, se para qualquer implementação I construída de acordo com D, então  $I \in E$ . Esta definição de desenho tem uma consequência: ao definir uma receita para a construção de implementações, diminui-se o número de implementações possíveis (só restam aquelas que respeitam a receita). Uma implementação pode estar dentro da especificação mas fora do desenho proposto (o contrário, porém, não pode acontecer). Nesse sentido é apropriado dizer que o desenho refina a especificação.

Usaremos a noção de **interface** que define um conjunto de operações para especificar um dado comportamento. Veremos como as interfaces funcionam no capítulo 9. O desenho será complementado por uma outra ferramenta de engenharia de software denominada

**desenho por contrato** descrita no capítulo 11. Serão apresentados exemplos completos na 3<sup>a</sup> parte, nomeadamente, várias estruturas de dados e algoritmos associados.

Para a implementação da receita proposta pelo desenho, usaremos conceitos das linguagens centradas em objectos (na verdade, o próprio desenho já é construído nessa direcção). A concepção de um sistema centrado em objectos foca os seguintes pontos:

- A classe é um tipo de dados com uma eventual implementação (se possuir uma implementação diz-se uma **classe concreta**, senão é uma **classe abstracta**). A noção matemática, útil num contexto totalmente abstracto, transforma-se progressivamente numa noção de *software*.
- O sistema é uma colecção de classes onde nenhuma é considerada especial.
- O sistema é estruturado através de dois tipos de relações entre classes: (i) a **relação de herança** – uma classe X é derivada de outra classe Y (em inglês, *is-a relation*); e (ii) a **relação de cliente** – uma classe X é cliente de outra classe Y (em inglês, *has-a relation*). A relação de cliente pode ser interpretada como uma **relação de composição** (quatro rodas são parte de um carro) ou como uma **relação de agregação** (um estudante aprende numa escola). A relação de composição é mais forte que a de agregação (ao eliminar o objecto carro eliminam-se os seus quatro objectos roda, mas eliminar o objecto escola não implica necessariamente eliminar o objecto estudante).

É a descrição da estrutura modular (constituída pela classes) e das inter-relações entre esses módulos que define totalmente o sistema. Que conceitos são estes e como usá-los será a temática dos próximos capítulos.

---

## Exercícios

1. Especificar o TDA Fraction com as seguintes operações: (i) `set`, que dado dois números inteiros devolve a fracção constituída por esses valores; (ii) `num`, que dado uma fracção devolve o numerador; (iii) `den`, que dado uma fracção devolve o denominador; (iv) `equals`, que verifica se duas fracções representam o mesmo valor; (v) `mult`, que devolve a multiplicação de duas fracções; e (vi) `div`, que devolve a divisão de duas fracções.

2. Especificar o TDA Natural com as seguintes operações: (i) `zero`, que devolve o número zero; (ii) `suc`, que dado um natural devolve o seu sucessor; (iii) `pred`, que dado um natural devolve o seu predecessor (o predecessor do zero é ele próprio); (iv) `add`, que devolve a soma de dois naturais; (v) `subtract`, que devolve a subtração de dois naturais; (vi) `product`, que devolve o produto de dois naturais; (vii) `div`, que devolve o quociente da divisão inteira de dois naturais; e (viii) `mod`, que devolve o resto da divisão inteira de dois naturais.

3. Estenda a especificação do TDA Boolean para incluir as operações: (i) `xor`, que calcula o “ou exclusivo” sobre dois booleanos; (ii) `impl`, que dados dois booleanos devolve a implicação do primeiro com o segundo; (iii) `eval`, que dada uma expressão booleana devolve o seu valor.
4. Especificar o TDA `Complex` com as seguintes operações: (i) `newC`, que dados dois números  $a$  e  $b$  devolve o complexo  $a+bi$ ; (ii) `real`, que dado um complexo devolve a sua parte real; (iii) `imag`, que dado um complexo devolve a sua parte imaginária; (iv) `add`, que devolve a soma de dois complexos; (v) `subtract`, que devolve a subtração de dois complexos; (vi) `product`, que devolve o produto de dois complexos; (vii) `div`, que devolve a divisão de dois complexos; e (viii) `modulus`, que devolve o valor absoluto de um complexo.
5. Estenda a especificação do TDA Natural para incluir as operações: (i) `equals`, que verifica se dois naturais são iguais; (ii) `lt`, que dados dois naturais verifica se o primeiro natural é menor que o segundo; (iii) `power`, que dados dois naturais  $a$  e  $b$ , devolve a potência  $a^b$ .



# **9 – Abstracção**

---

No fim do capítulo 8 definiu-se classe no contexto da especificação:

*Uma classe é um tipo com uma eventual implementação.*

Uma **classe abstracta** é uma classe que anuncia métodos para os quais não apresenta implementação. A consequência imediata é não ser possível criar objectos de uma classe deste género.

Uma classe abstracta é um passo entre um TDA 100% abstracto e uma implementação 100% concreta. A utilidade é listar o conjunto de métodos que definem o comportamento de modo a orientar uma futura implementação. A classe abstracta é uma descrição, uma intenção sobre o pretendido. É possível não efectuar um passo directo da classe abstracta para a classe concreta. Pode haver uma sequência de classes progressivamente menos abstractas até se atingir a implementação final.

Por exemplo, para representar objectos geométricos cria-se uma classe inicial *Figura* com as características consideradas necessárias (como *área* e *perímetro*). Esta classe não tem uma implementação associada, mas indica o necessário para um objecto ser uma figura. Para concretizar um pouco mais define-se a classe abstracta *Polígono* onde se define o perímetro mas não a área. A partir de *Polígono* cria-se uma classe concreta *Quadrado* com um conjunto de métodos que fornecem todos os serviços prometidos pela sequência de classes abstractas derivadas (esta derivação de classes é possível pelo mecanismo de **herança** – conferir capítulo 10).

## 9.1 Classes Abstractas

Para definir uma classe abstracta prefixa-se a definição da classe com a palavra `abstract` bem como os métodos que não possuem implementação. No exemplo seguinte, a classe abstracta define um tipo de pontos 2D. Esta classe tem quatro métodos abstractos que permitem consultar e alterar as coordenadas dos pontos e um método concreto `distOrigem()` para calcular a distância à origem. É possível concretizar desde já este último método porque o funcionamento é independente da implementação:

```
abstract public class Ponto2D {  
  
    abstract public double getX();  
    abstract public double getY();  
    abstract public void setX(double a);  
    abstract public void setY(double a);  
    public double distOrigem() {  
        return Math.sqrt(getX()*getX() +  
                        getY()*getY());  
    }  
}
```

Não é possível instanciar classes abstractas porque alguns dos métodos não possuem implementação. Nenhum método de uma classe abstracta pode ser prefixado com `final`, `static` ou `private`. É possível implementar todos os métodos de uma classe e ainda designá-la abstracta, informando desta forma que a implementação actual (do ponto de vista global do sistema onde a classe se insere) ainda está incompleta e deve ser completada por outras classes.

O programador, desde a definição das classes abstractas até às classes concretas, precisa analisar três pontos:

- A especificação do TDA (os serviços disponibilizados têm de ser implementados da forma especificada).
- A representação explícita dos dados.
- A implementação dos métodos especificados (segundo a representação escolhida).

Destes três pontos, apenas os serviços referidos no TDA devem ser públicos. Os outros dois, a representação e a implementação devem ser privados. O cliente não os deve conhecer nem ter acesso. Porquê? Porque o contrato apenas garante o serviço, não se compromete com detalhes de implementação. O fornecedor tem a liberdade de modificar a implementação desde que não altere os serviços prometidos. Se o cliente tivesse acesso à parte interna da implementação, qualquer modificação pelo fornecedor poderia ter consequências inesperadas na parte do cliente.

Tome-se o exemplo da especificação do ponto 2D. Se a representação das coordenadas fosse constituída por dois números `double` (`x` e `y`) e fosse pública, nada impedia o cliente de aceder directamente a esses valores (`objecto.x` e `object.y`). Se posteriormente, o fornecedor alterasse a representação para um vector `coord[]` (onde `coord[0]` armazena `x` e `coord[1]` armazena `y`), o acesso `objecto.x` algures no código fonte do cliente passaria a estar errado. Porém, os serviços especificados (`getX()`, `distOrigem()`, `setY()`, etc.) continuam a funcionar. Só foi alterada a representação não a funcionalidade especificada.

## 9.2 Encapsulamento

O acesso aos atributos e métodos de uma classe pode e deve ser controlado pelo programador. Certos componentes de um objecto não devem ser usados por qualquer outro objecto – o que justifica a existência de diferentes **níveis de acesso**, i.e., a capacidade de **encapsular informação** (do inglês, *encapsulation, information hiding*).

O programador pode escolher se cada componente deve ser visível por todas as outras classes, em nenhuma classe ou só por algumas classes. A encapsulação é conseguida através do uso de modificadores de acesso, nomeadamente: `public`, `private` e `protected`. Os níveis de acesso, do mais restrito para o mais geral, são:

- *Privado* – um atributo ou método privado só é acessível dentro da definição da própria classe. Usar a palavra `private`.
- *Pacote* – são visíveis dentro da própria classe e a todas as classes que pertençam ao mesmo pacote (ver anexo A). Este é o acesso por omissão.
- *Protegido* – são visíveis dentro da classe, nas classes do mesmo pacote e ainda nas subclasses desta classe (informação sobre subclasses no capítulo 10). Usar a palavra `protected`.
- *Público* – são visíveis em todos os locais onde a classe é visível. Usar a palavra `public`. O modificador `public` pode ser usado na classe, senão a classe só é visível dentro do pacote.

Qualquer componente público ou protegido disponível para outros programadores deve-se considerar parte do serviço disponibilizado pela classe em questão. Como consequência poderá ser impossível modificar esses componentes uma vez disponibilizados e postos em uso. Já os elementos privados e de pacote são considerados detalhes de implementação, invisíveis para todas as classes fora do contexto do pacote. Os atributos – ou seja, a representação explícita do estado do objecto – devem ser considerados componentes privados. Os atributos devem ser acedidos por métodos de leitura e escrita fornecendo ao cliente um controle indirecto sobre essa informação. Uma convenção bastante utilizada prefixa os métodos de consulta com a palavra `get` e os de alteração com a palavra `set`.

Voltando ao exemplo dos pontos. A classe seguinte implementa os serviços anunciados pela classe abstracta Ponto2D (a palavra `extends` indica que esta classe é uma derivação de Ponto2D):

```
public class Ponto2D_1 extends Ponto2D {  
    private double x, y;  
    public double getX() { return x; }  
    public double getY() { return y; }  
    public void setX(double a) { x=a; }  
    public void setY(double a) { y=a; }  
}
```

Como os atributos `x` e `y` estão escondidos dos clientes da classe, o programador pode alterar a representação e a implementação dos métodos públicos. Já as assinaturas definidas na especificação ficam na mesma.

Uma solução vectorial alternativa:

```
public class Ponto2D_2 extends Ponto2D {  
    private double[] coord = {0.0, 0.0};  
    public void setX(double a) { coord[0]=a; }  
    public void setY(double a) { coord[1]=a; }  
    public double getX() { return coord[0]; }  
    public double getY() { return coord[1]; }  
}
```

Se existirem métodos auxiliares à execução dos serviços disponibilizados (que não pertencem ao contrato) são geralmente considerados como privados.

### 9.3 Tipos e Interfaces

A classe é o conceito principal das linguagens centradas em objectos. Quando se diz que uma classe é uma (eventual) implementação de um TDA assumimos que uma classe serve para definir um certo tipo. Veremos nesta secção situações onde é útil o contrário: definir um tipo sem definir uma classe. Isto consegue-se em Java através na noção de **interface**.

*Uma interface é um tipo sem implementação associada.*

Enquanto uma classe pode ser vista como eventual mistura de desenho e implementação, uma interface é puro desenho. A interface descreve o cabeçalho dos métodos que as classes concretas têm de executar (é também possível incluir constantes numa interface). A interface é somente forma, não função.

Por exemplo, uma interface para definir o tipo “Círculo”:

```
public interface Circulo {
    double PI = 3.14159;
    double area();
    double perimetro();
}
```

O uso dos modificadores `public` e `abstract` nos métodos são opcionais porque por definição todos os componentes são públicos e abstractos. Já os atributos são considerados `final` e `static`. Não significa isto que as constantes tenham de ser inicializadas com valores constantes. Por exemplo:

```
interface X {
    int RAND_BIT = (int)Math.random()%2;
}
```

Esta inicialização ocorre antes da criação do primeiro objecto de cada classe que implemente este tipo (sendo `static` estas constantes fazem parte dos componentes estáticos da classe). Assim, duas classes que implementem a interface `X` poderão ter um valor de `RAND_BIT` diferente.

A interface é uma noção abstracta. Não podem existir objectos instanciados directamente de uma interface. Qualquer instânciação como a seguinte é inválida:

~~☒ Ponto2D obj = new Ponto2D();~~

Mas se não é possível instanciar objectos destes tipos, qual a utilidade? As classes Java podem implementar uma ou mais interfaces.

```
public class UmPonto implements Ponto2D {
    ...
}
```

A expressão sintáctica descrita acima representa um compromisso da classe. A classe `UmPonto` compromete-se a implementar o tipo definido pela interface `Ponto2D`, ou seja, implementar todos os métodos descritos pela interface. Esta abordagem permite estruturar e relacionar os principais tipos de um sistema abstraindo os detalhes de implementação.

Como as classes podem implementar uma ou mais interfaces, o programador pode conjugar tipos diferentes numa mesma classe. Continuando o exemplo anterior, considere-se uma 2<sup>a</sup> interface que define o tipo de objectos descritos por *strings*.

```
public interface Imprimivel {
    String emString();
}
```

Se a implementação da classe `UmPonto` fosse a intersecção dos objectos do tipo `Ponto2D` e do tipo `Imprimivel`:

```
public class UmPonto implements Ponto2D, Imprimivel {  
    ...  
}
```

Um objecto desta classe é igualmente visto como um objecto do tipo da interface. No exemplo seguinte, o método `imprimir()` espera um objecto de tipo `Imprimivel`. Invocar o método com um objecto da classe `UmPonto` não é errado, dado que esta classe implementou o tipo `Imprimivel`.

```
public void imprimir(Imprimivel o) {  
    System.out.println(o.emString());  
}  
UmPonto obj = new UmPonto(1.0, 0.0);  
 imprimir(obj);
```

Dentro da definição da classe é possível aceder às constantes de uma interface directamente pelo nome, ou prefixando-o com o nome da interface:

```
public class UmCirculo implements Circulo {  
    ...  
    setX(getX() * PI);  
    setX(getX() * Circulo.PI);  
    ...  
}
```

Existe uma desvantagem no uso das interfaces. Se adicionarmos um novo método à interface temos de encontrar todas as implementações relacionadas de forma a actualizá-las (o que é lógico mas pode resultar em demasiado trabalho). Já no caso de uma classe abstracta, desde que se inclua uma implementação padrão desse novo método, nada mais precisa ser alterado.

## Interfaces como Capacidades

A noção do tipo disponibilizado pela interface é visto por vezes como uma capacidade que a classe tem. No exemplo anterior, a classe `UmPonto` possui a capacidade de ser imprimível. Esta é uma outra forma de usar a noção de tipos puramente abstractos. O pacote de classes que vem com a linguagem disponibiliza um conjunto destas capacidades que estabelecem certos comportamentos entre classes de outra forma não relacionadas. As mais importantes são:

- `Cloneable` – uma classe que implemente esta interface suporta o mecanismo de clonagem, i.e., os objectos dessa classe conseguem criar uma cópia de si próprios.
- `Comparable` – objectos de uma classe com esta capacidade podem comparar-se entre si. A classe que implemente esta interface deve codificar o método

`compareTo()` que recebe um argumento com um objecto do mesmo tipo e devolve um valor negativo se o objecto for menor que o argumento, positivo se for maior, ou zero se forem iguais.

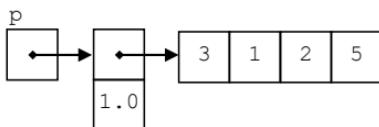
- `Runnable` – objectos deste tipo podem ser executados em concorrência.
- `Serializable` – objectos deste tipo podem ser lidos e escritos de e para fluxos de dados binários.

Existem interfaces que não possuem nenhum método, são **interfaces de sinalização** (do inglês, *marker interfaces*). Indicam que as classes que as implementam possuem alguma característica assumida (à responsabilidade de quem as programa). As interfaces `Cloneable` e `Serializable` são deste tipo. O resto deste capítulo explica o uso destas duas interfaces. Veremos ainda exemplos de clonagem e da interface `Comparable` na 3<sup>a</sup> parte. A interface `Runnable` e os mecanismos de concorrência relacionados estão fora do âmbito deste texto.

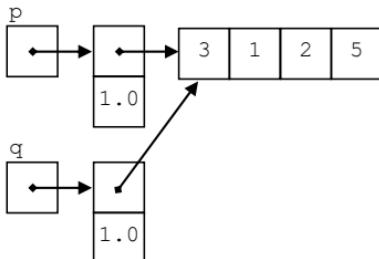
## Clonagem

A interface `Cloneable` define o tipo dos objectos que podem ser copiados. A clonagem é um mecanismo de cópia que um objecto pode oferecer se houver necessidade (e for possível). Existem dois tipos de clonagem, a **clonagem superficial** (do inglês, *shallow cloning*) e a **clonagem profunda** (do inglês, *deep cloning*). A clonagem superficial cria de um novo objecto com uma configuração idêntica ao original. A clonagem profunda cria um novo objecto com a configuração original mas também cria cópias de todos os objectos referenciados nessa configuração (e assim recursivamente).

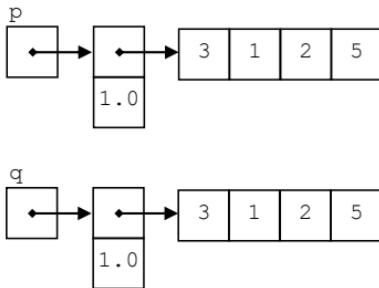
Considere o seguinte objecto com uma configuração constituída por um real e por um vector de inteiros, ou seja, um tipo primitivo e um tipo de referência.



Uma clonagem superficial para um objecto *q* resultaria em:



Na clonagem profunda:



Uma classe com este tipo de serviço deve implementar a interface `Cloneable`. O tipo de clonagem efectuado (superficial ou profundo) depende do contexto de uso da classe. Por vezes, a clonagem superficial é suficiente, noutras situações a clonagem profunda deve ser escolhida.

Se a clonagem superficial for suficiente, é possível usar o método `clone()` já existente na classe `Object`. Senão é necessário rescrever o método. Este método lança uma exceção do tipo `CloneNotSupportedException` quando, numa clonagem profunda, um dos objectos referenciados não implementa a interface `Cloneable`.

Considere a classe *C* dos objectos *p* e *q* dos diagramas anteriores com uma clonagem superficial:

```

public class C implements Cloneable {
    public double real;
    public int[] vector = {0,0,0,0};
    public Object clone() {
        try {
            return super.clone();
        }
    }
}
  
```

Neste caso, o processo de cópia utiliza a clonagem disponibilizada pela classe `Object`. No capítulo 10 será explicado o uso da palavra `super`.

```

    }
    catch (CloneNotSupportedException e) {
        return null;
    }
}
```

Se não for possível duplicar, a referência da cópia é nula.

```

    }
}
}
```

Um exemplo:

```

C obj1 = new C(), obj2;
obj1.real = 1.0;
obj1.vector[0] = 1;
obj2 = (C) obj1.clone();
System.out.println(obj2.real + ":" + obj2.vector[0]);

obj1.real = 2.0;
obj1.vector[0] = 2;
System.out.println(obj2.real + ":" + obj2.vector[0]);
□ 1.0:1←
1.0:2←
```

O atributo `real` alterado em `obj1` não se reflectiu em `obj2` já que este foi um dos componentes duplicados durante a clonagem. O vector não foi copiado (dado ser uma clonagem superficial) e é partilhado pelos dois objectos verificando-se uma alteração em `vector[0]`. Na linha carregada onde se executa o método `clone()`, é realizada uma conversão pois o `clone()` devolve um objecto da classe `Object` mas está a ser armazenado numa referência da classe `C`.

A mesma classe com clonagem profunda:

```

public class C implements Cloneable {
    public double real;
    public int[] vector = {0,0,0,0};
    public Object clone() {
        try {
            C novoObjecto = (C) super.clone();
            novoObjecto.vector = (int []) vector.clone();
            return novoObjecto;
        }
```

```
    }
    catch (CloneNotSupportedException e) {
        return null;
    }
}
```

Antes de devolver o objecto é obrigatório executar a clonagem dos atributos de tipo referência. Para cada tipo de referência precisa existir o método `clone()`. No exemplo, os vectores já possuem o método `clone()` implementado. As mesmas instruções de atribuição do exemplo anterior produzem o resultado:

```
□ 1.0:1←
  1.0:1←
```

## Serialização \*

A **serialização** (do inglês, *serialization*) é a capacidade de uma classe transformar uma sua instância numa sequência de bytes. Pode ser útil, por exemplo, para armazenar em memória secundária um objecto para posterior leitura ou para enviar um objecto pela Internet. Uma classe que implemente a interface `Serializable` pode ler uma sequência de bytes (que representa um estado) e criar um objecto com esse estado (em inglês, *deserialization*).

A serialização é efectuada pelo método `writeObject()` da classe `ObjectOutputStream` e a ‘deserialização’ pelo método `readObject()` da classe `ObjectInputStream` derivadas respectivamente das classes `OutputStream` e `InputStream` explicadas no capítulo 7.

O processo para tornar uma classe serializável segue os seguintes passos:

- Explicitar no cabeçalho que a classe implementa a interface `Serializable`.
- Rescrever, se necessário, os métodos `writeObject()` e `readObject()`.
- Para deserializar é preciso que a superclasse seja serializável ou que tenha um construtor sem argumentos para o processo de deserialização o invocar (já existe na classe `Object`).
- Todos os atributos de tipo referência têm de referenciar objectos serializáveis.

Em muitas classes é suficiente marcar a classe como serializável para que o processo funcione sem problemas.

O processo de serialização só processa atributos que não sejam de classe nem atributos **temporários** (do inglês, *transient*). Estes atributos temporários correspondem a atributos que o programador não considera relevantes para serem serializados, devendo ser prefixados com a palavra `transient`.

```
private transient int temp;
```

No seguinte exemplo, a classe `Ponto2D` possui um atributo que representa a norma do ponto (i.e., a distância à origem), que não se considera relevante, mas que precisa ser recalculado quando o objecto é deserializado. Marca-se o atributo como temporário e rescreve-se a deserialização:

```
import java.io.*;
public class Ponto2D implements Serializable {
    private double x, y;
    private transient double norma;
```

Esta é a configuração. O atributo `norma`, como referido, é temporário.

```
public Ponto2D() {
    x = y = norma = 0.0;
}
private double calcularNorma() {
    return Math.sqrt(x*x+y*y);
}
public void setX(double newx) {
    x = newx;
    norm = calcularNorma();
}
public void setY(double newy) {
    y = newy;
    norm = calcularNorma();
}
public String toString() {
    return "["+x+":"+y+"], norma = " + norma;
}
```

Para além do construtor, implementou-se os dois métodos de acesso que modificam os atributos (e actualizam a norma) e o método que descreve numa *string* a configuração do objecto.

```
private void readObject(ObjectInputStream in)
    throws IOException, ClassNotFoundException {
    in.defaultReadObject();
    norma = calcularNorma();
}
```

A rescrita do processo de deserialização. O método `readObject()` invoca um dos múltiplos métodos disponibilizados pela classe `ObjectInputStream` (para maiores detalhes sobre esta classe consultar as bibliotecas Java). Neste caso invoca o método `defaultReadObject()` que executa o processo padrão de leitura para depois realizar a tarefa extra: actualizar a norma do novo objecto (reparar que essa actualização não ocorre no construtor apenas nos métodos de acesso). As excepções `IOException` e `ClassNotFoundException` lançadas pelo método podem ocorrer em `defaultReadObject()`.

```
public static void main(String[] args)
    throws IOException, ClassNotFoundException {
    Ponto2D p = new Ponto2D(1.6, 6.5);
    System.out.println(p);
    File f = new File("teste.dat");
}
```

Vamos escrever e ler num ficheiro denominado “teste.dat”.

```
ObjectOutputStream out = new
    ObjectOutputStream(new FileOutputStream(f));
out.writeObject(p);
out.close();
Ponto2D q;
ObjectInputStream in = new
    ObjectInputStream(new FileInputStream(f));
q = (Ponto2D)in.readObject();
in.close();
System.out.println(q);
}
}
□ [3.0:4.0], norm = 5.0 ↵
[3.0:4.0], norm = 5.0 ↵
```

Alguns pontos:

- A serialização cria uma cópia profunda de todos os objectos referenciados pelo objecto serializado, sendo serializados conjuntamente com o objecto.
- Ao gravar-se no mesmo fluxo dois objectos que referem um terceiro objecto, os três são gravados de forma coerente, i.e., quando deserializados, os dois novos objectos continuam a partilhar a referência do terceiro.
- A serialização de vários objectos deve ser uma operação atómica, i.e., não se deve alterar o estado do programa antes do processo de tradução terminar. Caso

contrário, corre-se o risco de se estar a armazenar um conjunto de objectos com configurações incoerentes.

- Estes fluxos de leitura e escrita de objectos são similares aos fluxos binários de dados já que ambos partilham a mesma camada de comunicação entre o emissor e o receptor. Porém, existe uma diferença fundamental: enquanto nos fluxos binários de dados são transferidos blocos informação (inteiros, reais, caracteres, etc.), nestes fluxos são transferidos objectos concretos, ou seja, o estado de cada objecto mais o comportamento geral da classe a que pertencem.
- A serialização oferece um mecanismo de transferência de informação complexa entre qualquer ambiente. Um objecto pode ser codificado numa sequência de bits no sistema operativo X, ser transferido pela Internet e finalmente descodificado no sistema operativo Y, desde que existam máquinas virtuais para X e Y e que ambas tenham acesso ao ficheiro `.class` da classe do objecto serializado.
- A serialização fornece aos objectos uma forma de persistência. Por exemplo, o estado de um objecto pode sobreviver entre diferentes execuções do programa.
- A serialização converte em bits todos os atributos não temporários, onde se incluem os atributos privados ou protegidos. Se existirem preocupações sobre a segurança dos dados, tornam-se os atributos “secretos” em atributos temporários e procuram-se alternativas para inicializar essa informação (através de encriptação ou de outras formas de transmissão).

Ao processo de transformar uma descrição binária num objecto designa-se por **reificação do objecto** (do inglês, *object reification*). A reificação, em termos gerais, significa a tradução de um conceito ou objecto de modo a torná-lo perceptível numa representação diferente<sup>17</sup>.

A reificação de um objecto de uma dada classe é semelhante à criação de uma nova instância dessa classe. Ambas precisam da descrição da classe e ambas introduzem um novo objecto no estado do programa. Porém, enquanto a instanciação reserva necessariamente nova memória, a reificação pode utilizar se possível a memória inicial onde a representação do objecto se encontra (não no caso da leitura/escrita de ficheiros do exemplo anterior). A inicialização é diferente. Enquanto a instanciação utiliza um dos construtores para atribuir um estado válido inicial ao objecto, a reificação define o estado do objecto em termos da informação traduzida que não tem necessariamente de ser um estado válido. Por exemplo, os atributos temporários não são inicializados (o que resulta na rescrita dos métodos `readObject()` e `writeObject()` quando necessário).

Os mecanismos de reificação são igualmente usados para conseguir o que se designa por **reflexão** (do inglês, *reflection, introspection*), a capacidade para modificar características estáticas do objecto que não são normalmente visíveis em tempo de execução, como a

---

<sup>17</sup> Seja essa representação mais abstracta ou de mais baixo nível (como uma sequência de bits).

estrutura da configuração, as relações de herança ou a criação e manipulação dinâmica de classes. Se a serialização representa um processo de reificação do objecto para uma representação mais concreta, a reflexão cria uma representação mais abstracta do objecto reificado. Existe um conjunto de ferramentas para utilizar a reflexão na manipulação de tipos (classes) de forma a aumentar o controle e a versatilidade da própria linguagem Java. Porém, uma discussão mais aprofundada deste assunto está fora do âmbito deste texto.

# **10 – Herança**

---

Não vivemos num mundo aleatório mas sim num mundo onde é possível encontrar relações duradouras entre objectos. Quando certos objectos partilham um determinado conjunto de características e comportamentos é normal considerá-los semelhantes e agrupamo-los em classes. Essa classificação dos objectos que nos rodeiam, à medida que se elabora, possui uma estrutura hierárquica encabeçada por conceitos unificadores que se vão especializando em conjuntos mais específicos. A construção desta estrutura hierárquica é um acto de taxionomia.

Um exemplo é a classificação na qual dividimos tudo o que nos rodeia em dois conjuntos: os seres vivos e os objectos inanimados. Dentro dos seres vivos existe a divisão entre reinos: animal, vegetal, fungos, eucariotas unicelulares e procariotas. Cada um desses reinos divide-se e subdivide em diferentes grupos (classe, ordem, família, género) até chegar ao conceito fundamental de espécie (definido aproximadamente como: o conjunto dos seres vivos com genomas suficientemente semelhantes capazes de se reproduzir e criar descendência fértil). Assim, um homem ou uma mulher pertencem à espécie *Homo Sapiens*, mas pertencem igualmente à ordem dos Primatas, à classe dos Mamíferos e ao reino dos Animais. No Homem, esta classificação biológica pode continuar por critérios sociais (de uma língua, de uma nação, de um clube de futebol) até ao detalhe considerado conveniente.

Qualquer classificação é arbitrária. As culturas produzem classificações muito próprias do mundo que observam. Quando uma classificação reflecte um modelo apropriado da realidade (o que de certa forma separa as classificações culturais das classificações científicas) torna-se útil na separação dos problemas em categorias mais facilmente

analisáveis. É este conceito de sistematização e modularização que levou à noção informática de classes e objectos.

Para além desta divisão em grupos, a estrutura hierárquica subjacente estabelece um mecanismo de relação – de herança – entre grupos. Qualquer mamífero herda as propriedades que definem esse conjunto. O mesmo se pode dizer dos Primatas e de qualquer outro grupo genérico. Por exemplo, o facto de bebermos leite materno durante o período inicial da nossa vida não é um comportamento específico do Homem, é algo que define os Mamíferos. Assim, todas as características que partilhamos com os Mamíferos devem ser descritas no grupo Mamíferos e não no grupo *Homo Sapiens*<sup>18</sup>. Este é uma das motivações do mecanismo de herança das linguagens centradas em objectos.

## 10.1 Relações de Herança

Para além da relação de cliente e fornecedor que pode existir entre duas classes, a outra forma de relação entre classes é a **herança**. O conceito de herança procura responder a duas características importantes na construção de sistemas informáticos complexos:

- A **Extensibilidade** – corresponde à facilidade de alteração de um determinado sistema para aumento da capacidade ou da funcionalidade.
- A **Reutilização** – é o grau no qual um determinado sistema pode ser usado em diferentes contextos ou problemas.

A vida útil de um sistema deve estender-se para lá da especificação inicial. O mundo modifica-se, as necessidades do presente não são as mesmas do futuro. É esperado que dentro do processo de manutenção ocorram situações onde é necessário alterar partes do sistema para este continuar a cumprir a tarefa para a qual for desenhado. Se um sistema foi extensível e reutilizável – numa palavra, **modular** – os custos associados a mudanças (sejam conjecturais ou estruturais) são reduzidos. Um sistema é modular se alterações localizadas à funcionalidade implicarem alterações localizadas na estrutura. Se a correcção e a robustez caracterizam a fiabilidade de um produto a curto e médio prazo, a extensibilidade e a reutilização asseguram a viabilidade do mesmo a longo prazo.

### Um primeiro exemplo

Vejamos os conceitos através de um exemplo geométrico. Em primeiro lugar, considere uma classe abstracta e uma classe concreta de pontos a duas dimensões (semelhante ao capítulo 8):

---

<sup>18</sup> Em termos gerais, pelo menos. A Biologia não é Matemática, está repleta de excepções e de excepções às excepções. Facto que nos faz lembrar como a arbitrariedade sempre acompanha as classificações humanas.

```
public abstract class Ponto2D {  
    abstract public double getX();  
    abstract public double getY();  
    abstract public void setX(double a);  
    abstract public void setY(double a);  
  
    public double distOrigem() {  
        return Math.sqrt(getX()*getX() +  
                         getY()*getY());  
    }  
  
    public double distancia(Ponto2D p) {  
        return Math.sqrt((getX()-p.getX())*  
                         (getX()-p.getX()) +  
                         (getY()-p.getY())*  
                         (getY()-p.getY()));  
    }  
  
    public String toString() {  
        return "[" + getX() + "," + getY() + "]";  
    }  
}  
  
public class Ponto2D_1 extends Ponto2D {  
    private double x, y;  
  
    public Ponto2D_1(double x, double y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public void setX(double a) { x=a; }  
    public void setY(double a) { y=a; }  
    public double getX() { return x; }  
    public double getY() { return y; }  
}
```

A classe `Ponto2D_1` estende a classe abstracta `Ponto2D`. Em primeiro lugar (explicado no capítulo 9) ocorreu um processo de concretização da classe abstracta: `Ponto2D_1` implementa os métodos abstractos `getX()`, `getY()`, `setX()` e `setY()` da classe abstracta. Em segundo lugar, a classe `Ponto2D_1` herdou as funcionalidades já existentes na classe `Ponto2D` dado que esta não era uma classe 100% abstracta. Qualquer objecto da classe `Ponto2D_1` pode utilizar os métodos `distancia()`, `distOrigem()` e `toString()`. Existe uma relação de herança entre as duas classes: a classe `Ponto2D_1` deriva de `Ponto2D`, i.e., `Ponto2D_1` é uma especialização de `Ponto2D`.

Vejamos agora a classe Poligono:

```
public class Poligono {  
    protected Ponto2D[] vertices;  
    public Poligono(int cap) {  
        vertices = new Ponto2D[cap];  
    }  
    public void insert(int i, Ponto2D p) {  
        vertices[i] = p;  
    }  
    public double perimetro() {  
        double result = 0.0;  
        for (int i=0;i<vertices.length;i++)  
            result += vertices[i].distancia(  
                vertices[(i+1)%vertices.length]);  
        return result;  
    }  
    public String toString() {  
        String s = "";  
        for (int i=0;i<vertices.length;i++)  
            s += vertices[i] + " ";  
        return s;  
    }  
}
```

Esta classe disponibiliza um construtor que recebe um argumento com o número de pontos do polígono, criando um vector com a dimensão adequada. Possui um método `insert()` para inserir um novo ponto na posição *i* do vector: um método para calcular o perímetro e o método `toString()` para imprimir os valores dos vértices.

O método `insert()` recebe um valor de tipo `Ponto2D`, não `Ponto2D_1`. Porquê? Porque o método necessita apenas do comportamento descrito pela classe abstracta. Ou seja, permite que a classe `Poligono` utilize qualquer implementação de `Ponto2D`. Um ponto a favor da modularidade.

Observe o seguinte exemplo e resultado:

```
Poligono pl = new Poligono(3);  
pl.insert(0,new Ponto2D_1(0.0,2.0));  
pl.insert(1,new Ponto2D_1(2.0,2.0));  
pl.insert(2,new Ponto2D_1(1.0,1.0));  
System.out.println(pl);
```

```
System.out.println("perimetro: " + pl.perimetro());  
□ [0.0,2.0] [2.0,2.0] [1.0,1.0] ←  
perimetro: 4.82842712474619 ←
```

Considere a necessidade de criar uma classe Rectangulo. Ao derivar esta nova classe da classe Poligono obtém-se automaticamente o conjunto das funcionalidades já existentes.

```
public class Rectangulo extends Poligono {  
    public Rectangulo() {  
        super(4);  
    }  
    public double area () {  
        return vertices[0].distancia(vertices[1]) *  
               vertices[1].distancia(vertices[2]);  
    }  
}
```

Qualquer objecto da classe Rectangulo é um objecto da classe Poligono. Diz-se que Rectangulo é uma **subclasse** de Poligono e que Poligono é a **superclasse** de Rectangulo. Um objecto Rectangulo pode usar directamente os métodos de ambas as classes. A subclasse recebe os componentes da superclasse pelo mecanismo de herança. De facto, recebe apenas os componentes públicos e protegidos da superclasse, não os privados.

A subclasse define um subtipo da superclasse. Mas esta relação não é similar à relação de inclusão da teoria dos conjuntos. Por um lado, os estados possíveis dos objectos da subclasse são um subconjunto dos estados válidos da superclasse. Mas a subclasse por ser mais específica que a superclasse, possui um comportamento mais extenso e elaborado. Este argumento ajuda a não confundir tipos com conjuntos. Um conjunto é uma coleção de elementos. Um tipo é um conjunto de valores associados por um conjunto de operações.

A construção de um objecto Rectangulo é idêntica à construção de um polígono com quatro vértices. Assim, para evitar a duplicação de código fonte (um código duplicado contém duas vezes mais erros e a manutenção é duas vezes mais difícil) é invocado o construtor da superclasse. Isto foi realizado através do comando `super`.

Segue um exemplo:

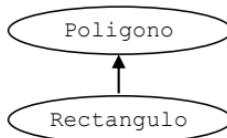
```
Rectangulo r = new Rectangulo();  
r.insert(0,new Ponto2D_1(0.0,2.0));  
r.insert(1,new Ponto2D_1(2.0,2.0));  
r.insert(2,new Ponto2D_1(2.0,0.0));  
r.insert(3,new Ponto2D_1(0.0,0.0));  
  
System.out.println(r);
```

```

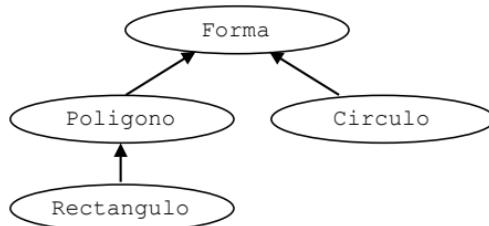
System.out.println("perimetro: " + r.perimetro());
System.out.println("area: " + r.area());
□ [0.0,2.0] [2.0,2.0] [2.0,0.0] [0.0,0.0] ←
    perimetro: 8.0 ←
    area: 4.0 ←

```

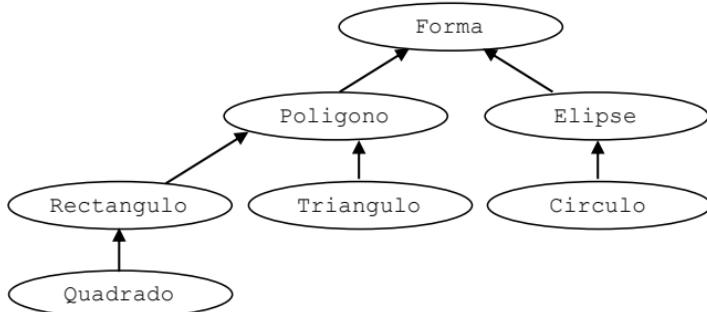
É possível representar num diagrama a estrutura de heranças de um dado sistema. Cada elipse é uma classe, cada seta representa uma relação de herança (da subclasse para a superclasse). Neste exemplo:



O exemplo poderia ser aumentado com a inserção de novas classes relacionadas. Se incluirmos a classe **Circulo** (com o cálculo específico do perímetro e da área) e a classe abstracta **Forma** de onde todas as outras derivam:



Se inserirmos mais três classes: **Elipse**, **Triangulo** e **Quadrado**:



Dependendo do problema, a estrutura de heranças deve representar um modelo adequado desse mesmo problema. A pertinência da classificação escolhida tem consequências práticas na correcção e no desempenho da aplicação final.

O mecanismo de herança promove uma modularização e um desenvolvimento gradual da implementação. Uma classe estendida não insere novos erros na classe já testada. Qualquer erro que surja é da responsabilidade da nova classe (considerando que a classe original está correcta, claro).

## 10.2 Herança no Java

### Subclasses e Classes Finais

Uma classe, abstracta ou concreta, pode ser estendida por outra classe. Como já reparámos, em Java assinala-se esse facto com a palavra `extends`.

```
public class Rectangulo extends Poligono {  
    ...  
}
```

A classe que estende é uma **subclasse** (ou descendente) da classe estendida. A classe estendida é a **superclasse** (ou ascendente) da classe que estende. A noção de superclasse e subclasse aplica-se a ascendentes ou descendentes não directos (no exemplo anterior, `Forma` é a superclasse de todas as outras classes). A classe explicitamente estendida é a **superclasse directa** da classe que estende.

Em Java é possível definir **classes finais**, i.e., classes que não permitem subclasses. Adiciona-se ao cabeçalho da classe a palavra `final`.

```
public final class Quadrado extends Rectangulo {  
    ...           // não permite derivações  
}
```

### Construtores

A palavra `this` refere o objecto actual. A palavra `super` interpreta o objecto actual como se fosse uma instância da superclasse directa. Cada classe pode aceder ao construtor da superclasse denominado `super()` que inicializa parte da configuração necessária para o correcto funcionamento da classe em questão. Esta invocação tem de ser a primeira instrução dos métodos construtores que a utilizam. Não confundir `super` (o objecto como instância da superclasse directa) e `super()` (o construtor dessa superclasse).

```
public Rectangulo() {  
    super(4);      // invoca construtor da classe Poligono  
}
```

Se numa classe não for declarado um construtor, é definido por defeito um construtor sem parâmetros<sup>19</sup>. O construtor da superclasse não é herdado pela herança, porém cada objecto da subclasse é também um objecto da superclasse. Assim, cada construtor deve invocar o construtor `super()` da superclasse directa. Se o programador não o explicita, a invocação do construtor por defeito (sem argumentos) é automática. Ocorre um erro de compilação se a superclasse não possuir esse construtor (o construtor por omissão deixa de estar disponível quando o programador declara um construtor seu).

A criação de um novo objecto significa não só a execução do construtor, mas a execução de todos os construtores das superclasses da classe desse objecto. Resumindo, a inicialização de um objecto tem as seguintes fases: (i) inicialização da superclasse, (ii) inicialização dos atributos e dos blocos de inicialização e (iii) execução do próprio construtor. Um exemplo:

```
class X {  
    private int x=1;  
    { System.out.print(x); }  
    public X() {  
        System.out.print(" X ");  
    }  
}  
  
public class Y extends X {  
    private int y=2;  
    { System.out.print(y); }  
    public Y() { // o construtor da superclasse é  
                // invocado implicitamente  
        System.out.print(" Y ");  
    }  
}  
...  
Y obj = new Y();  
□ 1 X 2 Y
```

Ao invocar o construtor da superclasse antes de qualquer outra instrução previne-se situações em que excepções ocorridas na superclasse tivessem de ser tratadas pela classe derivada.

---

<sup>19</sup> Para classes que é suposto não existirem objectos (como a classe `java.lang.Math`) e para evitar a criação de um construtor público inútil, deve-se definir um construtor privado.

A tarefa dos construtores deve ser, simplesmente, inicializar o objecto numa configuração inicial válida, evitando a execução de outros métodos que produzam alterações não triviais na configuração do mesmo.

## Destrutores

Contrariamente ao referido nos construtores, os métodos `finalize()` devem-se iniciar com as instruções relativas à classe e só depois com a invocação do correspondente método da superclasse. Não se pretende destruir a informação relativa à configuração da superclasse antes de se resolver os problemas pendentes da própria classe do objecto.

## Conversões entre Classes Derivadas

Os objectos instâncias da subclasse pertencem igualmente ao tipo definido pela superclasse. Um subtipo implementa um conjunto maior de funcionalidades mas não deixa de cumprir as restrições do supertipo. Desta forma é válido usar explicitamente um elemento da subclasse como se fosse um elemento da superclasse (em inglês, *upcasting*). O exemplo da próxima secção exemplifica esta capacidade.

Já a possibilidade inversa (em inglês, *downcasting*), i.e., usar um objecto de um supertipo como se fosse um subtipo pode ser possível, porém é perigosa. O subtipo é algo mais específico e pode conter configurações ou comportamentos que não existem no supertipo. No exemplo das figuras geométricas o seguinte poderia ser inválido:

```
Poligono p;  
...  
double total = ((Rectangulo)p).area();
```

Porquê? Se o verdadeiro tipo de `p` for um rectângulo ou uma subclasse de rectângulo, não haverá qualquer problema. Porém, se `p` for um objecto da classe `Poligono`, o método `area()` simplesmente não existe e a excepção `ClassCastException` é lançada.

A palavra conversão pode induzir em erro. O objecto sobre o qual é realizada a “conversão” não é modificado. Apenas é efectuado um teste que garante se o objecto pode ser visto como se fosse de uma dada classe. Este mecanismo é útil quando se pretende implementar métodos genéricos que manipulam objectos de um dado supertipo. Um exemplo:

```
class ConjPolig {  
    public Poligono[] v;  
    public ConjPolig(int cap) {  
        v = new Poligono[cap];  
    }
```

Esta classe armazena polígonos num vector. O atributo privado da classe é um vector para objectos do tipo `Poligono`. Podem-se igualmente armazenar objectos de subclasses, como por exemplo, objectos da classe `Rectangulo`.

```
public void insert(int pos, Poligono item) {  
    v[pos] = item;  
}  
}
```

Um exemplo:

```
ConjPolig obj = new ConjPolig(5);  
Poligono p = new Poligono(4);  
Rectangulo r = new Rectangulo();  
r.insert(0, new Ponto2D_1(0.0,2.0));  
r.insert(1, new Ponto2D_1(2.0,2.0));  
r.insert(2, new Ponto2D_1(2.0,0.0));  
r.insert(3, new Ponto2D_1(0.0,0.0));  
obj.insert(0,p);  
obj.insert(1,r);
```

Inserimos dois polígonos (um deles um rectângulo) na referência `obj`. Se tentarmos imprimir o valor das respectivas áreas:

```
System.out.println(((Rectangulo)obj.v[1]).area());  
try {  
    System.out.println(((Rectangulo)obj.v[0]).area());  
}  
catch (ClassCastException e) {  
    System.out.println("Downcast inválido!");  
}  
□ 4.0 ←  
    Downcast inválido! ←
```

O programador deve garantir que durante a execução, a conversão dos objectos seja correcta (ou apanhar a referida exceção se for incorrecta).

## O Operador de Pertença

O operador `instanceof` determina se um objecto é instância de uma classe:. O operador devolve `true` se o objecto à esquerda for uma instância da classe à direita, caso contrário devolve `false`. Este operador também funciona para vectores. Se a referência for igual a `null`, o resultado é sempre falso. Alguns exemplos:

```
Poligono p = new Poligono();  
int[] i = new int[10];  
boolean a = i instanceof Poligono;      a igual a false  
boolean b = p instanceof Poligono;      b igual a true
```

```
boolean c = i instanceof int[];      c igual a true  
boolean d = null instanceof int[];    d igual a false
```

Este operador também funciona nas relações de herança:

```
Rectangulo r = new Rectangulo();  
boolean e = r instanceof Poligono;    e igual a true
```

## Ocultar Atributos das Superclasses

Uma classe pode esconder atributos da superclasse. No exemplo seguinte, o atributo *x* da classe Avo e Pai é sucessivamente ocultado pela subclasse seguinte. No entanto é possível aceder a esses atributos através do uso das conversões, juntamente com o *this* e com o *super*.

```
class Avo {  
    protected int x;  
}  
  
class Pai extends Avo {  
    protected int x;  
}  
  
class Filho extends Pai {  
    protected int x;  
  
    void m() {  
        x = 1;  
        ((Pai)this).x = 2;  
        ((Avo)this).x = 3;  
        System.out.print("x do Filho = "+x);  
        System.out.print(", x do Pai = "+((Pai)this).x);  
        System.out.print(", x do Avo = "+((Avo)this).x);  
        super.x = 4;  
        System.out.print(", x do Pai = "+((Pai)this).x);  
    }  
}  
  
...  
(new Filho()).m();  
  
□ x do Filho = 1, x do Pai = 2, x do Avo = 3, x do Pai = 4
```

## Rescrever Métodos das Superclasses

Uma classe que declare um método com o mesmo nome e assinatura que um método de uma superclasse diz-se que rescreve (ou redefine) esse método (em inglês, *method overriding*). Os métodos da superclasse são acedidos dentro da classe derivada usando a

palavra `super`. Porém, não é possível usar em Java `super.super.metodo()`. Uma classe não pode aceder directamente aos métodos das superclasses da superclasse directa.

Existe uma diferença fundamental entre ocultar atributos e rescrever métodos. Um atributo ocultado pode ser acedido directamente por uma conversão. Isso não ocorre com os métodos. Enquanto um atributo de uma superclasse pode fazer ou não fazer sentido para a classe, um método foi escrito por alguma razão. É possível que o método da superclasse nem sequer seja compatível com o comportamento dos objectos da classe derivada.

Uma variável de um tipo primitivo (um inteiro, por exemplo) não pode armazenar informação de outro tipo (um inteiro não pode guardar um booleano). Tipos de dados com esta característica designam-se **monomórficos**. Quando um objecto é criado tem a configuração e o comportamento da classe. No entanto, a referência do objecto não tem de ser necessariamente dessa classe, pode ser de uma superclasse. À propriedade de uma referência poder referenciar objectos de diversos tipos de dados dá-se o nome de **polimorfismo** (do inglês, *polymorphism*). Mas há limitações, uma referência não pode referenciar um objecto de uma superclasse ou de uma classe não relacionada.

- Avo a = **new** Avo();
- Avo b = **new** Pai();
- Pai c = **new** Avo(); // Avo é superclasse!
- Pai c = **new** String(); // Pai e String não se relacionam

Devido ao polimorfismo, o compilador não consegue determinar, dado uma referência e um diagrama de herança, qual o método a executar porque depende do objecto armazenado num determinado instante da execução. É durante a execução do programa que o sistema reconhece o tipo de objecto e qual o método correcto a invocar. A esta capacidade dá-se o nome de **ligação dinâmica** (do inglês, *dynamic binding*). Esta faculdade contrasta com os restantes casos, onde a **ligação estática** (do inglês, *binding* ou *static binding*) entre a invocação do método e a execução do método é realizada no momento da compilação (o único mecanismo disponível em linguagens como o C ou o PASCAL).

Considere o exemplo com as mesmas três classes:

```
class Avo {  
    public void m() {  
        System.out.println("metodo do Avo");  
    }  
}  
  
class Pai extends Avo {  
    public void m() {  
        System.out.println("metodo do Pai");  
    }  
}
```

```
class Filho extends Pai {  
    public void m() {  
        System.out.println("metodo do Filho");  
    }  
}  
...  
Avo a;  
Filho f = new Filho();  
Pai p = new Pai();  
a = p;  
a.m();           // invoca m() da classe Pai  
a = f;  
a.m();           // invoca m() da classe Filho
```

- metodo do Pai ↵
- metodo do Filho ↵

O objecto a declarado como uma referência de tipo Avo, ‘soube’ em tempo de execução de que tipo eram os objectos referenciados para invocar correctamente o método `m()`. É este tipo de funcionalidade que a ligação dinâmica assegura. A regra utilizada é invocar o método mais ‘próximo’ da classe do objecto (procura na superclasse directa, na superclasse desta e assim sucessivamente).

Temos assim duas regras: (a) a **regra da atribuição**: uma atribuição `x = E` só é válida se a expressão `E` for do tipo da referência `x` ou de um subtipo de `x`; e (b) a **regra da invocação**: a invocação do método `x.m()` só é válida se `m()` for acessível ao tipo do objecto `x` ou de um supertipo de `x`.

O polimorfismo é útil mesmo sem o mecanismo de ligação dinâmica. Armazenar numa referência elementos de tipos derivados permite, por exemplo, uma maior flexibilidade no uso de estruturas de dados. Combinado com o operador `instanceof`, permite simular o mecanismo da ligação dinâmica (aqui implementado explicitamente pelo programador):

```
Avo[] v = new Avo[] { new Avo(),  
                      new Pai(),  
                      new Filho(),  
                      new Avo() };  
  
for(int i=0; i<v.length; i++)  
    if (v[i] instanceof Filho)  
        ((Filho)v[i]).m();  
    else if (v[i] instanceof Pai)  
        ((Pai)v[i]).m();  
    else if (v[i] instanceof Avo)  
        ((Avo)v[i]).m();
```

- metodo do Avo ↵
- metodo do Pai ↵
- metodo do Filho ↵
- metodo do Avo ↵

O vector v[] armazena elementos de diversos tipos (associados por herança). Este tipo de estruturas denominam-se **estruturas de dados polimórficas**.

Os métodos de classe têm um comportamento semelhante ao dos atributos. Podem ser ocultados mas não rescritos. A tentativa de ocultar um método da superclasse por um método de classe provoca um erro de compilação.

```
class Avo {  
    protected int x;  
    public static void a(Avo obj) {obj.x=3;}  
}  
  
class Pai extends Avo {  
    protected int x;  
    public static void a(Pai obj) {obj.x=2;}  
}  
  
class Filho extends Pai {  
    protected int x;  
    public static void a(Filho obj) {obj.x=1;}  
  
    void m() {  
        a(this);  
        Pai.a(this);  
        Avo.a(this);  
        System.out.print("x do Filho = "+x);  
        System.out.print(", x do Pai = "+((Pai)this).x);  
        System.out.print(", x do Avo = "+((Avo)this).x);  
    }  
}  
...  
(new Filho()).m();  
□ x do Filho = 1, x do Pai = 2, x do Avo = 3
```

## Parâmetros, Métodos e Classes Finais

Os parâmetros de um método podem ser declarados finais, i.e., não podem ser alterados dentro do próprio método. Parâmetros finais não são muito relevantes, pois todos os argumentos são passados por valor. As alterações efectuadas sobre os parâmetros são sempre sobre cópias, não produzindo efeitos secundários nos argumentos da invocação. Alterações aos objectos referenciados não podem ser impedidas pelo uso do final.

```
public int m(final int i, final C obj) {  
    ☑ i++;  
    ☑ obj = new C();  
    ☑ obj.atributo += i;  
    ☑ return i + obj.atributo;  
}
```

Já os métodos finais possuem duas motivações importantes:

- Eficiência – um método final permite que o compilador substitua cada invocação do método pelo bloco de instruções do método. Permite um melhor desempenho, principalmente se for aplicado em métodos simples (em que o tempo de invocação é uma percentagem relevante no tempo total da execução do método) invocados repetidamente.
- Herança – um método final não pode ser substituído numa classe derivada. Garante um determinado comportamento para todos os objectos do tipo definido pela classe, o que inclui todas as eventuais subclasses. Por definição, os métodos privados são declarados como finais, porque os métodos privados sendo inacessíveis às subclasses, não podem ser estendidos por estas.

Uma classe final não pode ser derivada. É perfeitamente válido criar objectos de uma classe final, somente não é possível criar subclasses dela. Por definição, todos os métodos de uma classe final são finais. Já sobre os atributos não é feita nenhuma assunção.

```
final class F { ... }  
☒ class G extends F { ... }  
☑ F obj = new F();
```

Definir uma classe final pode ser uma decisão demasiado rígida em determinadas situações. É difícil adivinhar quais serão as necessidades futuras que recairão sobre o sistema. Deve-se usar este conceito com alguma cautela para evitar restrições desnecessárias entre as relações das classes da aplicação.

## Classes Estáticas

Uma classe final não permite subclasses. Uma classe estática não permite a criação de objectos. Qual a utilidade? Pode servir como recipiente de serviços, como a classe `Math` do capítulo 4. No entanto, não existe em Java um modificador com esta função sendo necessário usar um subterfúgio: definir um construtor privado para a classe.

```
class S {  
    static double N_OURO = 0.6180339887;  
    static double mult(double x) { return x*N_OURO; }  
    private S() { }  
}
```

Ao definir um construtor privado não é possível construir objectos desta classe. Assim, a classe deve conter somente componentes de classe, sejam estes atributos ou métodos.

- S obj = new S();
- double x = S.mult(0.1);

É possível criar classes com um número fixo de objectos. No exemplo seguinte, a classe Naipe possui apenas quatro objectos (os quatro naipes) sendo impossível criar objectos novos.

```
final class Naipe {  
    private String nome;  
    private Naipe(String s) { nome = s; }  
    public final static Naipe PAUS = new Naipe("paus");  
    public final static Naipe ESPADS= new Naipe("espadas");  
    public final static Naipe COPAS= new Naipe("copas");  
    public final static Naipe OUROS= new Naipe("ouros");  
    public String toString() { return nome; }  
}  
...  
System.out.print(Naipe.PAUS);  
 paus
```

O construtor privado impede a sua utilização pública, mas dentro do âmbito da classe é possível criar objectos. Estes objectos foram considerados como de classe e finais para impedir qualquer modificação. A própria classe é considerada como final para impedir a criação de subclasses.

## Herança vs. Composição

Referimos no fim do capítulo 8 que existem duas formas de relacionar classes, a relação de composição (uma classe é cliente de outra) e a relação de herança (uma classe deriva de outra).

A composição é usada quando uma classe necessita de outra para a implementação mas não para a sua interface. Os atributos de uma classe devem ser (quase) sempre privados podendo-se disponibilizar métodos de acesso (normalmente prefixados por *get* e *set*) tanto a potenciais clientes (onde os métodos devem ser públicos) ou somente a potenciais subclasses (onde os métodos devem ser protegidos). A composição é apropriada para identificar o estado do objecto, ou seja, exprimir diferenças de configuração.

A herança é usada quando uma classe é vista como uma especialização de outra, para estender o comportamento, concretizando ou adicionando funcionalidades. A herança é apropriada para identificar e exprimir diferenças de comportamento. A herança entre classes possui um aspecto duplo: (a) a herança do tipo, i.e., a subclasse é também do tipo

da superclasse, os requisitos e promessas que os objectos da superclasse possuem, têm de ser respeitados pelos objectos da subclasse; e (b) a herança da implementação, i.e., a subclasse pode reaproveitar os atributos e a implementação dos métodos acessíveis da superclasse.

## 10.3 A Classe Object

Quando se define uma classe sem indicar uma extensão, o compilador implicitamente assume que a nova classe é derivada da classe `Object`. A classe `Object` é a superclasse de todas as classes Java. Assim, em qualquer definição de classe existe sempre uma relação de herança. A classe `Object` define um conjunto de métodos que formam parte do comportamento de todos os objectos. Entre os mais usados:

- `boolean equals(Object obj)` – Devolve verdadeiro se as referências do `this` e o `obj` são idênticas. É conveniente rescrevê-lo consoante o contexto (provavelmente para comparar os estados dos objectos, em vez de verificar simplesmente se referenciam o mesmo objecto).
- `Object clone()` throws `CloneNotSupportedException` – Para as classes que implementam a interface `Cloneable` faz um clone do próprio objecto e devolve essa referência. A clonagem é superficial ou profunda dependendo da implementação deste método.
- `String toString()` – Devolve uma representação textual do objecto. Deve ser rescrita por todas as classes que tenham representação textual. Por omissão, esta função mostra o nome da classe e a posição relativa na memória associada ao programa.
- `void finalize()` throws `Throwable` – O método executado antes da recolha de lixo quando o objecto já não possui referências associadas. Deve ser rescrito quando houver necessidade de apagar algo que a recolha de lixo automática não resolve.
- `Class getClass()` – Devolve o objecto `Class` relacionado com o objecto em questão. Cada classe instanciada num programa Java é associada com um objecto desta classe especial. Para mais informações, consultar a bibliografia.

## 10.4 As Excepções Revisitadas

No capítulo 6 foi apresentado o mecanismo das excepções e explicado como se criam, lançam e apanham. Porém, ficou em aberto a questão de como criar novos tipos de excepções.

### Criar Excepções

O programador não está restrinido às excepções disponibilizadas. Pode-se criar novas excepções consoante as necessidades. É necessário derivar a nova excepção de uma

excepção existente. Usualmente a derivação ocorre a partir da classe `Exception`. Por exemplo:

```
public class DivisaoPorZero extends Exception {  
    public DivisaoPorZero() {  
        super();  
    }  
    public DivisaoPorZero(String msg) {  
        super(msg);  
    }  
}
```

## Apanhar Vários Tipos de Excepções

Quando um `catch` apanha objectos de uma dada classe, automaticamente apanha todos os objectos que pertencem a subclasses desta. No próximo exemplo, o segundo `catch` nunca é activado dado que `DivisaoPorZero` é uma subclasse de `Exception`.

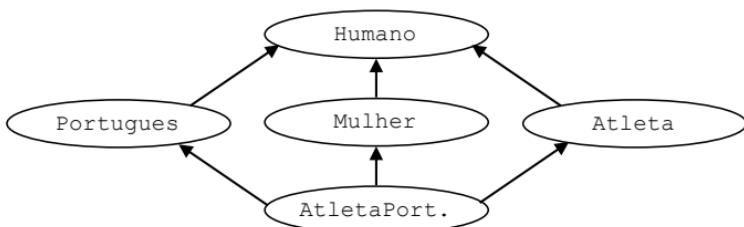
```
try {  
    a = sc.nextDouble();  
    divide(a);  
}  
catch (Exception excp) {  
    ...          // apanha todas as excepções!!!  
}  
catch (DivisaoPorZero excp) {  
    ...          // nunca chega aqui  
}
```

Por esta razão, a ordem como são colocados os vários blocos `catch` é relevante. O compilador detecta este tipo de situações.

## 10.5 Herança Múltipla

O mecanismo de herança em Java possui uma forte restrição: cada classe só pode estender uma outra classe. Em Java não existe o que se denomina por **herança múltipla**.

Em linguagens como o Eiffel ou o C++, cada classe pode estender uma ou mais classes pertencendo a um ou mais tipos. Esta capacidade corresponde à nossa visão do mundo: muitos objectos reais pertencem a mais do que um tipo. Um carro anfíbio pertence ao conjunto dos automóveis e ao conjunto dos barcos. Uma atleta portuguesa pertence ao conjunto das mulheres, ao conjunto dos atletas e ao conjunto dos portugueses:



Cada tipo é representado por uma classe que determina a configuração e o comportamento. Por vezes é útil a uma nova classe recolher esses componentes de mais de uma superclasse. Mas na linguagem Java não existe herança múltipla, promovendo-se uma filosofia de **herança simples**. A motivação desta decisão é que certas funcionalidades da herança múltipla são difíceis de gerir, complicando a estrutura interna do compilador de forma desnecessária (segundo os arquitectos do Java).

O argumento em defesa deste ponto de vista baseia-se no facto de existirem problemas quando se herda atributos ou implementações de métodos com o mesmo nome de mais de uma superclasse. Por exemplo, no diagrama acima, se a classe Humano tiver um método abstracto `correr()` e este for implementado por Mulher e Atleta, a que método de que superclasse nos referimos quando usamos `correr()` na classe AtletaPortuguesa?

A abordagem usada na linguagem Java é admitir a herança de vários desenhos e somente uma implementação. Assim, é válido a uma classe implementar uma ou mais interfaces e estender uma só classe. Com uma só implementação herdada, os problemas referidos da herança múltipla são eliminados.

Uma solução do problema anterior seria definir os tipos Humano, Mulher, Portugues e Atleta como interfaces implementando-as, quando for necessário, em classes concretas. Para seguir o diagrama, a classe AtletaPortuguesa teria de implementar as três interfaces herdando assim os seus desenhos. Algo como:

```
interface Humano { ... }
interface Portugues extends Humano { ... }
interface Mulher extends Humano { ... }
interface Atleta extends Humano { ... }
class AtletaPortugues implements Portugues,
                           Mulher,
                           Atleta { ... }
```

Este exemplo aponta igualmente para uma outra possibilidade das interfaces: cada nova interface pode estender uma ou mais interfaces, como nas classes. O problema das heranças múltiplas não ocorre porque não existe qualquer implementação associada aos diagramas de derivação de interfaces. Se a hierarquia de classes em Java é uma árvore, a

hierarquia de todas derivações incluindo os tipos definidos pelas interfaces é um grafo acíclico.

---

## Exercícios

1. Qual a razão para não existirem classes privadas?
2. Se é possível adicionar atributos e métodos às classes por herança, porque não é possível remover?
3. Defina uma classe final e tente criar uma sua subclasse. O que acontece?
4. Defina a classe Fruta e as subclasses Laranja, Banana, Morango, Maçã. Inclua atributos e métodos nos locais apropriados que permitam saber a cor da fruta, a forma, o peso médio e o preço.
5. Relacione as seguintes classes: Animal, Servivo, Vegetal, Homem, Árvore, Mamífero, Fruta e Gato.
6. Relacione as seguintes classes: Escola, Estudante, Professor, Funcionário, Turma, Sala, Pessoa e Aula.
7. Defina duas classes A e B com construtores que se anunciem (por exemplo, incluindo a instrução `System.out.println("criada instância da classe X")`). Defina uma classe C subclasse de A que crie, no construtor, um objecto da classe B. O que ocorre se for criado um objecto da classe C?
8. Qual a diferença nas funcionalidades das classes seguintes?

```
class Point {  
    protected double x,y;  
    public void Point() {  
        x = y = 0;  
    }  
}  
  
class Pixel extends Point {  
    public char color;  
}
```

9. Como se pode confirmar que os construtores por defeito são criados pelo compilador?
10. Como se pode confirmar que o construtor da superclasse é executado antes do construtor da subclasse?
11. Implemente um método que receba um valor inteiro como parâmetro e lance uma excepção `ValorNegativo` se esse valor não for positivo.

12. Altere o método anterior para gerar uma outra excepção `ValorNulo` se o parâmetro for zero, mas essa excepção é tratada internamente.

13. Qual o tipo de excepção apanhado por este bloco?

```
try {  
    ...  
}  
catch (Exception e) {  
    ...  
}
```

14. Quais os tipos de excepção apanhados? O exemplo está correcto?

```
try {  
    ...  
}  
catch (Exception e) {  
    ...  
}  
catch (ClassCastException e) {  
    ...  
}
```

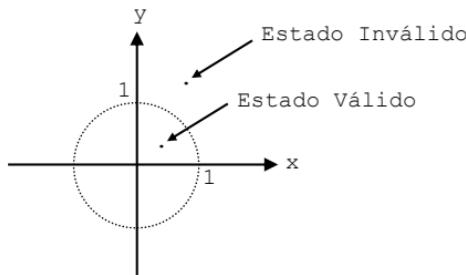


# 11 – Desenho por Contrato

---

Uma classe representa um tipo de objectos que partilham um conjunto de métodos (i.e., um comportamento) e um conjunto de atributos (i.e., uma configuração). Os atributos definem um **espaço de estados** para os objectos desse tipo. Porém, o estado de um objecto é restringido por um subconjunto de estados considerados válidos, por um número de **restrições** que definem o **domínio de aplicação** desses objectos. As restrições são limitações aos intervalos de valores dos atributos e/ou dependências entre dois ou mais atributos. As restrições diminuem a incerteza sobre o estado do objecto e facilitam – dado que estabelecem os critérios para – a manutenção num estado correcto.

Considere-se o seguinte exemplo onde se define um tipo constituído por pontos pertencentes ao círculo de raio 1 centrado na origem.



O espaço de estados é  $\mathbb{R}^2$ . O estado válido de um objecto deste tipo é dado pelo par ordenado  $(x,y)$  com a restrição R descrita por:

$$\sqrt{x^2 + y^2} \leq 1$$

Esta restrição torna interdependentes os dois atributos da classe. Um comando que altere a coordenada  $x$  está limitado pela restrição R.

O conjunto de restrições de uma classe cria um mecanismo de resistência a processos de mudança que produziriam estados incoerentes<sup>20</sup>. Se isso ocorresse, o objecto deixaria de pertencer à classe que o define. Na execução de uma aplicação este factor leva à incorrecção do estado do programa tornando-o inútil.

## 11.1 Correcção

A correcção de um programa não acontece por acaso. Todo o processo de especificação, o refinamento promovido pelo desenho, as técnicas que a programação centrada em objectos promove, são ferramentas de planificação que minimizam a probabilidade de incorrecções. Em certos casos é possível provar que um programa está correcto, ou seja, que o algoritmo descrito calcula exactamente o pretendido. Porém, este processo pode tornar-se muito complexo. É útil em casos simples onde a análise e consequente demonstração consegue ser realizada.

Neste contexto, cada instrução é vista como um acontecimento que modifica o estado do programa (o estado de um ou mais objectos numa perspectiva centrada em objectos). A especificação de uma instrução é uma asserção descrita da seguinte forma (esta notação foi introduzida por Charles Hoare em 1969 a partir de trabalho de Robert Floyd sobre o uso de asserções para cálculo de propriedades de programas):

```
{x==1} y=x+1; {y==2}
```

A expressão lógica que se situa antes da instrução (rodeado por chavetas) é a **pré-condição** e a que se situa depois é a **pós-condição**. No exemplo, o que se pretende dizer é o seguinte: assumindo que o valor da variável  $x$  é igual a 1, se a instrução terminar, então o estado do programa inclui a variável  $y$  com valor 2. Por motivos de simplificação (porque o estado do programa pode ser muito extenso) em cada condição é apenas indicado o conjunto de valores considerado relevante.

Outros dois exemplos:

```
{a==10 ∧ b==1}
while (a>0) {
    a--;
    b++;
```

---

<sup>20</sup> Designa-se por homeostase (do inglês, *homeostasis*) à capacidade dos sistemas complexos serem capazes de resistir a mudanças, mantendo o seu estado válido por adaptação ao ambiente a partir de um conjunto de restrições sobre os seus atributos.

```
}
```

```
{a==0 ∧ b==11}
```

```
{n≥0 ∧ j==1}
```

```
i=1;
```

```
while (i<=n) {
```

```
    j*=i;
```

```
    i++;
```

```
}
```

```
{j==n!}
```

Ao assumir uma dada pré-condição (se a pré-condição for sempre verdadeira designa-se **condição universal**) é necessário provar que a pós-condição se verifica. É neste ponto que reside a dificuldade, pois nem sempre é fácil provar que a expressão lógica manifestada na pós-condição é igual ao estado final do programa. Quando se prova que a pós-condição é verificada *se o programa terminar*, obtemos a **correcção parcial** do programa. Para obter a **correcção total** é preciso garantir que o programa efectivamente termina.

Se provarmos que o estado final do programa satisfaz o resultado da pós-condição então *o programa está correcto do ponto de vista dessa especificação*. A correcção é sempre relativa a uma dada especificação. Este facto imprime uma enorme importância na adequação da especificação do problema. O correcto deve ser equivalente ao considerado útil. Observe os seguintes exemplos:

```
{falso} programa {...}
```

```
{...}   programa {verdadeiro}
```

```
{verdadeiro} programa {...}
```

```
{...}           programa {falso}
```

- Na primeira linha temos a pré-condição mais forte<sup>21</sup> possível: a expressão falsa. Indiferentemente do trabalho de programação o programa está sempre correcto (aliás, o programa nem sequer precisa terminar). Quanto mais forte a pré-condição (i.e., quanto mais restrito é o conjunto de estados admissíveis) menos trabalho é necessário para atingir a correcção do programa.
- Na segunda linha temos a pós-condição mais fraca possível (é sempre verdadeira), qualquer programa *que termine* está correcto.
- Já no terceiro caso encontramos a pré-condição universal. Nada é garantido, o programa tem de fazer tudo para satisfazer a pós-condição.
- A pós-condição da quarta linha informa que qualquer programa será, por definição, incorrecto.

---

<sup>21</sup> Uma expressão lógica X é mais forte que uma expressão lógica Y se e só se  $X \neq Y \wedge X \Rightarrow Y$ .

Para analisar a correcção deve-se decompor o programa em subprogramas e verificar a correcção destes (a técnica de “dividir para conquistar” em acção). Seja um programa S dividido em dois subprogramas S1 e S2, i.e.,  $S = S1; S2$ . Dada a especificação  $\{a\} S \{c\}$ , a regra de composição diz que se provarmos  $\{a\} S1 \{b\}$  e  $\{b\} S2 \{c\}$  verificamos a especificação inicial. O processo de decomposição repete-se recursivamente até se chegar a comandos suficientemente simples com regras apropriadas. Veremos em seguida um exemplo para a instrução while.

## Metodologia de Dijkstra

O procedimento seguinte, desenvolvido por Edsger Dijkstra, prova a correcção total do comando de ciclo while (as outras instruções de ciclo podem ser traduzidas nesta). A estrutura geral de um ciclo while é a seguinte:

```
{PRE} ← a eventual pré-condição do problema  
inicialização  $I_0$   
while (guarda G) {  
    {G}  
    passo P ← o corpo do ciclo  
}  
{¬G}  
{POS} ← a pós-condição que se quer garantir
```

Se o ciclo terminar então a guarda G é falsa (logo a negação de G é verdadeira). Enquanto a guarda não for falsa, o corpo do ciclo P vai ser executado. É a iteração de P que produz o resultado desejado, a repetida execução de P tem de convergir para o estado final.

A primeira tarefa é observar a pós-condição para determinar o conteúdo de P. Os ciclos são necessários quando não conseguimos calcular directamente os valores desejados. Um ciclo é uma aproximação sucessiva da solução final: cada nova iteração aproxima o estado do programa do objectivo. A **invariante do ciclo** é a expressão dessa aproximação. É necessário encontrar a invariante apropriada para o corpo do ciclo P. A invariante, neste contexto, é a expressão lógica I que verifica a asserção:

$$\{I \wedge G\} \quad P \quad \{I\}$$

Considerando a invariante I e a guarda G como verdadeiras, se o passo P terminar então a invariante I é verdadeira. A execução do bloco da iteração não modifica o valor lógico de I (daí a designação de invariante).

Voltando ao exemplo, rescrevemos:

```
{PRE}
inicialização I0
{I}
while (guarda G) {
    {I ∧ G}
    passo P
    {I}
}
{I ∧ ¬G}
{POS}
```

Como escolher I? O critério é escolher I de modo que  $I \wedge \neg G \Rightarrow \text{POS}$ . Uma vez determinada a invariante I, a tarefa da inicialização  $I_0$  é validar a invariante com valores iniciais adequados. Ao encontrar  $I_0$  e I garantimos a correcção parcial.

É conveniente separar P em duas partes. Primeiro, há que determinar quais os comandos que realizam as acções necessárias para a progressiva concretização do objectivo (descrito na asserção POS) – designam-se estes comandos por **acção** do ciclo. Esta acção vai inevitavelmente modificar o estado do programa sendo quase certo que invalida a invariante. Deste modo, o segundo componente do ciclo designado **progresso**, tem como tarefa revalidar a invariante sem desfazer o trabalho levado a cabo pela acção:

```
{PRE}
inicialização I0
{I}
while (guarda G) {
    {I ∧ G}
    acção A
    progresso Pr
    {I}
}
{I ∧ ¬G ⇒ POS}
```

Outra tarefa do progresso é mostrar que para qualquer estado válido, o ciclo corresponde a um processo finito (existe um número limitado de estados que levam o estado inicial ao estado final). Com esse resultado garante-se a correcção total do ciclo.

Segue um exemplo completo para o ciclo que calcula o factorial do valor da variável n e coloca o resultado na variável j:

```

{n≥0}
i=0; j=1;           ← inicialização
{j==i!}            ← invariante I: j==i!
while (i!=n) {     ← guarda G: i≠n
    {j==i! ∧ i≠n}
    j*=i+1; ← a acção invalida a invariante, agora j é igual a (i+1) !
    i++;    ← o progresso sem desfazer a acção, revalida a invariante j==i!
}
{j==i! ∧ ¬(i≠n) ⇒ j==n!}

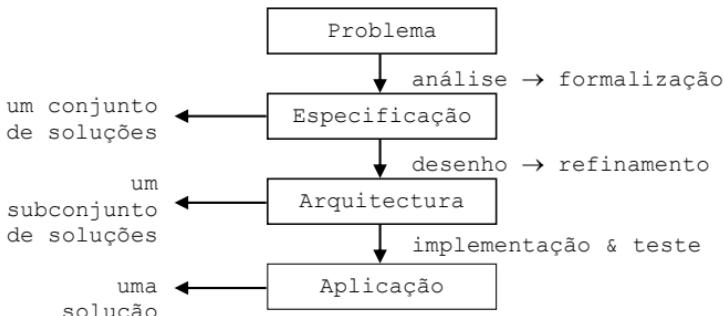
```

Com esta parte garantimos a correcção parcial (i.e., se terminar, termina correctamente). Basta verificar no progresso que a variável  $i$  começa no valor 1 e a cada iteração aumenta uma unidade. Para qualquer valor de  $n$  (maior que zero) a variável  $i$  acaba inevitavelmente por ser igual a  $n$ . Garantimos assim a terminação do ciclo. Com a terminação mais a correcção parcial garantimos a correcção total.

Para os casos de programação de larga escala, os métodos de correcção não são uma ferramenta viável (as demonstrações tornam-se inexequíveis). Porém, estes métodos fornecem uma bagagem conceptual extremamente útil na seguinte perspectiva: em vez de provar as asserções, descrevem-se numa linguagem apropriada (de modo a que possam ser testadas pelo computador). Se durante a execução do programa alguma asserção for violada é possível enviar um aviso e parar o programa. Não se garante que violações deixem de ocorrer mas garante-se que se ocorrerem não passam despercebidas. É nesta ideia que se desenvolve a programação por contratos.

## 11.2 Desenho por Contratos

Ao longo desta 2<sup>a</sup> parte observamos uma progressão do mais abstrato para o mais concreto que decorre da forma proposta para resolver problemas. Esquematicamente:



No contexto da programação, o **desenho por contrato** promove a concretização do acordo entre o cliente (quem requer um conjunto de serviços) e o fornecedor (quem disponibiliza esses serviços) de forma a assegurar determinados níveis de qualidade dos serviços em questão. Esse acordo contém um conjunto de restrições descritas pelo fornecedor sobre os serviços que proporciona, nomeadamente o uso de cada um dos atributos e métodos acessíveis pelo cliente. Estas restrições são de dois tipos:

- **Pré-condição** – uma asserção que descreve as restrições associadas a um dado método público do objecto.
- **Invariante da Classe** – uma asserção que descreve restrições gerais (seja de atributos individuais, seja de atributos interrelacionados) a que todos os objectos da classe estão sujeitos. A invariante caracteriza o conjunto dos estados válidos.

O fornecedor deve descrever, com o maior pormenor possível, o comportamento dos métodos no caso das restrições terem sido satisfeitas. Assim, o cliente conhece o que é garantido pelos serviços disponibilizados. Este tipo de asserção denomina-se por **pós-condição**. A invariante define o domínio, as pré e pós-condições descrevem as transições possíveis entre os estados válidos desse domínio. Em certos casos, alguns estados considerados válidos pela invariante podem não ser atingidos devido às restrições impostas aos métodos públicos, os únicos que manipulam os estados dos objectos. Um **contrato** é um grupo de asserções destes três tipos sobre uma classe e sobre os métodos públicos dessa classe. A invariante diz respeito somente à classe. As pré-condições descrevem o que cada método requer para o seu correcto funcionamento. As pós-condições asseguram um determinado conjunto de resultados do método em questão. Tanto as pré-condições como as pós-condições devem levar em conta a invariante da classe: as pré-condições não podem exigir um estado inválido do objecto, as pós-condições não podem forçar um estado inválido.

Não se pode discutir uma parte do contrato sem conhecer o contexto onde essa parte se insere. Desse ponto de vista, o contrato é uma unidade atómica formal referente à classe em questão. Um contrato descreve um conjunto de direitos e obrigações entre o fornecedor e o cliente. O cliente é obrigado a respeitar as restrições impostas pelas pré-condições. O fornecedor é obrigado a respeitar as pós-condições e as invariantes (o estado do objecto nunca pode ser inválido). Qualquer desrespeito por uma qualquer asserção implica uma quebra de contrato. Uma quebra de contrato nas pré-condições é um erro do cliente, enquanto uma quebra de contrato nas invariantes e nas pós-condições é um erro do fornecedor. Temos assim uma definição exacta das respectivas áreas de responsabilidade.

Um contrato descreve um conjunto de direitos. O cliente tem o direito de esperar os resultados descritos nas pós-condições, se respeitar as restrições do contrato. O fornecedor tem o direito de esperar o respeito pelas restrições, caso contrário não é obrigado a cumprir as pós-condições.

A metodologia do desenho por contrato não pretende resolver todas as ambiguidades na comunicação entre cliente e fornecedor. Propõe somente uma estrutura formal que

soluçona uma série de questões, minimizando os problemas contratuais que existem na diferença entre o pedido e o interpretado. Os contratos devem ser usados tanto na fase de desenho como na fase de implementação. Entre as vantagens:

- O desenho por contrato dá-nos uma definição formal de correcção. Uma classe está correcta em relação ao contrato, se cada uma das operações respeitar a invariante e as pós-condições quando receber qualquer conjunto de dados que respeite as pré-condições.
- A verificação se as propriedades especificadas estão a ser correctamente implementadas torna-se uma tarefa modular. Para saber se uma operação está correcta não necessitamos saber quais foram as operações executadas anteriormente. Sabendo que todas as operações do tipo respeitam a invariante, garante-se por indução que o estado do objecto é sempre válido (desde que correctamente inicializado).
- O programador ao pensar sobre os contratos apropriados para os métodos é levado a um raciocínio mental mais estruturado, reflectindo-se na simplicidade e na eficiência do desenho final do sistema.
- Aproxima a fase de especificação da fase de implementação ao disponibilizar uma linguagem com uma semântica clara e uma sintaxe semelhante à linguagem de programação usada pelo programador (quando existe uma ferramenta de contratos apropriada).
- Os contratos ajudam o cliente a entender o exigido e o assegurado pelas classes que pretende utilizar. Têm assim um papel importante na documentação das restrições e funcionalidades da aplicação.
- Servem para detectar problemas automaticamente se um dos objectos entrar num estado inválido. Controlam parte do mecanismo de lançamento de excepções, sendo uma ferramenta útil na fase de verificação. Esta relação entre verificação e documentação está profundamente interligada, já que ambos os processos usam o mesmo conjunto de asserções que definem o contrato. O documentado relaciona-se com o verificado.
- Simplificação da implementação dos métodos. Como as asserções tratam um conjunto de situações anómalas torna-se desnecessário incluir esses testes no método. Representa um maior desempenho na execução final do programa (menos testes) e uma maior facilidade de manutenção (menos código fonte).

Como em todas as opções, nem tudo são vantagens no uso dos contratos:

- Escrever bons contratos não é trivial. Escrever maus contratos é aumentar o número de erros sobre a aplicação que se deseja correcta.
- O uso de contratos pode induzir uma falsa sensação de segurança. Como foi dito não é possível exprimir tudo (e nem sempre é fácil lembrar-se de tudo) produzindo assim, uma especificação incompleta. Outra questão é que mesmo certos erros considerados nos contratos podem não surgir durante a validação da aplicação, por

serem muito raros (e, segundo a Lei de Murphy, deixam de ser raros quando executados pelo cliente).

- A qualidade de *software* custa tempo, o lançamento de uma aplicação com menos erros é mais lenta do que uma com mais erros (não é realmente um defeito dos contratos, mas mais uma inevitável consequência das constantes pressões do mercado).
- Os contratos foram inicialmente definidos para computação sequencial. Um estado válido é definido apenas para antes e depois do método terminar. Nada é garantido, principalmente sobre a invariante da classe, quando os métodos funcionam sobre um ambiente concorrente.
- Nem todas as linguagens possuem suporte para contratos. O Eiffel tem os contratos incluídos na estrutura base, mas o Java e muitas outras linguagens, não. Ou se usam ferramentas dedicadas (como veremos) ou não se utilizam de todo.
- Por vezes há propriedades que só são descritas por asserções que modificam o objecto para obter a resposta desejada. Como uma asserção nunca pode produzir efeitos secundários (*a computação especificada pelo programa deve ser independente das suas asserções*) esse tipo de asserções não pode ser executado. Designam-se por asserções **não verificáveis**.

No exemplo dos pontos referido no início do capítulo, a asserção que descreve o comportamento do método *setX(double x)* poderia ser assim:

*Se o argumento x respeitar a restrição R, então o novo valor deste objecto para o atributo que armazena a coordenada no eixo dos x é igual a x.*

O problema deste exemplo é estarmos a usar o português como linguagem para definir contratos (um problema semelhante à especificação dos TDA no capítulo 8). Todas as linguagens naturais são extremamente flexíveis e parte dessa flexibilidade reside na sua ambiguidade. Por este mesmo motivo são inadequadas para descrever situações formais. Se escolhermos uma linguagem com uma sintaxe e semântica bem definida, esta ambiguidade desaparece. Podemos não ser capazes de descrever tudo mas o que é expresso não sofrerá de problemas de interpretação.

### 11.3 A Linguagem de Contratos

Há numerosas abordagens para descrever uma linguagem de contratos. Nesta secção é utilizada uma ferramenta inspirada no desenho por contratos criado e desenvolvido por Bertrand Meyer nos fins dos anos oitenta para a linguagem de programação Eiffel. Esta ferramenta denominada JML<sup>22</sup> é um projecto cooperativo iniciado por Gary Leavens.

---

<sup>22</sup> A aplicação está disponível em <http://www.cs.iastate.edu/~leavens/JML/>

## Descrição do JML

As asserções do Java referidas na secção 6.2 são expressões lógicas que utilizam os tipos e operadores disponíveis na linguagem. No JML as asserções possuem um conjunto extra de operadores. As asserções são colocadas dentro de comentários especiais `//@ ...` ou `/*@ ... @*/`. Esta ferramenta actua como um pré-processador, adicionando ao programa original um conjunto de comandos que testam as asserções descritas nas pré-condições, pós-condições e invariantes. O programa alterado é então executado de modo a verificar se alguma das asserções é activada, sinal de que o programa possui defeitos por corrigir. Este tipo de execução é especialmente útil durante a fase de teste.

Usaremos três tipos de etiquetas:

- `public invariant` – especifica invariantes públicas de classes e interfaces.
- `requires` – especifica pré-condições de métodos.
- `ensures` – especifica pós-condições de métodos.

Cada uma destas etiquetas é incluída num comentário JML. Por exemplo:

```
//@ requires x != 0;  
public void div(int x);
```

Se existir mais de uma etiqueta do mesmo tipo na mesma classe, interface ou método, são interpretadas como uma conjunção. Os dois exemplos seguintes são equivalentes:

```
/*@ requires x > 0;  
 @ requires x < 10;  
 @*/  
  
//@ requires x > 0 && x < 10;
```

Cada uma das etiquetas é seguida por uma expressão lógica Java onde podem também ser usados os seguintes operadores:

- `\result` – o valor retornado pelo método. Somente usado nas pós-condições.  
`//@ ensures \result >= 1 && \result <= 4;`  
**public** int quadrant(Ponto2D p);
- `\old` – representa o valor de uma variável ou interrogação antes da execução do método. É somente usado nas pós-condições.  
`//@ ensures length() == 1 + \old(length());`  
**public** void insert(Object item);
- `<antecedente> ==> <consequente>` – a implicação lógica: se o antecedente for verdadeiro então é forçoso que o consequente o seja.

```
//@ ensures length() > 0 ==> !\result;
public boolean isEmpty();

• (\forall <tipo> <variável>; <expressão-1>; <expressão-2>) – o quantificador universal. A asserção é verdadeira se e só se cada <expressão-2> (para todos os valores de <variável> que satisfaçam <expressão-1>) for verdadeira.

//@ requires (\forall int i; 0<i && i<v.length();
              v[i-1]<v[i]);
public void binarySearch(int x, int[] v);

Neste exemplo, a procura binária de um elemento x num vector v[] exige como pré-condição que o vector deva estar ordenado de forma crescente.
```

- (\exists <tipo> <variável>; <expressão-1>; <expressão-2>) – o quantificador existencial. Funciona como o operador \forall mas basta existir um caso onde a asserção seja verdadeira, para a expressão total ser verdadeira.

```
//@ requires (\exists int i; 0<=i && i<v.length(); v[i]>0);
public int getMaxPositive(int[] v);
```

Neste exemplo, o método exige como pré-condição que pelo menos um dos elementos do vector v seja positivo.

- assert – o teste de uma asserção. O mecanismo é semelhante ao comando Java mas o predicado pode ser construído com os operadores do JML.

```
//@ assert i<nElems ==> v[i]==null;
```

Os atributos/métodos que podem ser usados nas asserções são os seguintes:

- Para as invariantes das interfaces/classes é possível aceder aos componentes de classe bem como aos métodos públicos dos objectos.
- As pré-condições têm o âmbito dos seus métodos.
- As pós-condições têm o mesmo âmbito das pré-condições, podendo ainda aceder ao resultado final do método (através do \result) bem como às interrogações públicas sobre a instância antes do método ter sido executado (através da função \old()).

É possível invocar métodos nas asserções. Todos os métodos utilizados em asserções têm de ser prefixados (nos respectivos cabeçalhos) com /\*@ pure @\*/.

```
/*@ pure @*/ public boolean isEmpty();
```

Antes da execução do construtor deve ser respeitada a pré-condição (se existir). Após a execução do construtor são testadas as invariantes e as pós-condições. Se o construtor

falhar por algum motivo (lançando uma exceção) a invariante e as pós-condições não são testadas (já que o objecto não foi criado correctamente).

Para os métodos públicos, todas as asserções (invariantes, pré e pós-condições) têm de ser verdadeiras. Para os métodos privados, apenas as suas pré e pós-condições têm de ser respeitadas. Os métodos privados podem violar a invariante pública enquanto o método público que os activou não terminar. Do ponto de vista externo não existe problema: o objecto, antes e depois da execução do método público, mantém a invariante.

Se um método terminar com uma exceção apenas se verifica a invariante, não a pós-condição. A lógica subjacente é que apesar de terminar de forma anormal – não havendo possibilidade de satisfazer a pós-condição – o método não pode invalidar a invariante da classe a que o objecto pertence. Em caso contrário, o comportamento do objecto não seria robusto e o programa entraria num estado inválido.

O JML possui igualmente uma forma de representar asserções não verificáveis, colocando a respectiva expressão entre (\*...\*), como no seguinte exemplo:

```
//@ requires (* x inclui uma função monótona *);  
public void m(Object x);
```

## Linhas Condutoras para o Uso de Contratos

Qualquer relação entre atributos que limita o conjunto de estados válidos do objecto deve ser descrita por uma invariante. No exemplo inicial, a restrição R (que delimita os pontos no interior da circunferência de raio 1 e centro na origem) é uma invariante natural da classe que descreve este tipo de objectos.

```
public class Ponto2D {  
    //@ public invariant distOrigem() <= 1;  
    ...  
}
```

Uma invariante pode ser vista como pré e pós-condição de todos os métodos públicos da classe, exceptuando o facto que as invariantes são responsabilidade do fornecedor enquanto as pré-condições são responsabilidade dos clientes. A invariante descreve quais são os estados válidos dos objectos antes e depois das execuções de cada comando (designados **estados estáveis**). Durante a execução de um comando, o objecto está num **estado instável** (e provavelmente inválido) mas as asserções não devem funcionar nesse intervalo de execução. Do ponto de vista das pré, das pós-condições e das invariantes, os métodos públicos são unidades atómicas de execução. Resumindo, a invariante deve ser estabelecida pelos construtores e mantida pelos métodos públicos.

Não é um erro se a invariante for redundante quando comparada com outras asserções (i.e., há duas ou mais asserções que assumem o mesmo). A redundância neste contexto não é um problema, é uma segurança. Porém, não se deve reproduzir no código de um método os

testes efectuados pelas pré-condições desse método (i.e., programação defensiva + contratos). Para além de desnecessário é um procedimento incorrecto no contexto do desenho por contrato. Um exemplo:

```
//@ requires r != 0.0;  
public void dividir(double r) {  
    if (r!=0.0)      // desnecessário e incorrecto!  
    ...  
    ...  
};
```

É desnecessário porque o teste deve ser efectuado pelo cliente (para satisfazer a pré-condição). É incorrecto porque o excesso de código redundante, por muito inofensivo que pareça de um ponto de vista local ao método, é prejudicial em relação a todo o sistema. Se esta programação defensiva for repetida por dez mil métodos, existem dez mil instruções extra para manter.

Desde o capítulo 4 que distinguimos interrogações de comandos. Uma interrogação não altera o estado do objecto, ao contrário de um comando. Entre as interrogações é preciso separar as **interrogações básicas** – perguntas directas sobre o estado do objecto – e as **interrogações derivadas** – questões mais complexas descritas pelas interrogações básicas. A definição da interrogação derivada à custa das interrogações básicas é a pós-condição da interrogação derivada. No exemplo, `getX()` e `getY()` são interrogações básicas que devolvem as coordenadas do objecto e `éOrigem()` é uma interrogação derivada pois o resultado pode ser derivado das duas interrogações anteriores. Assim:

```
//@ ensures \result == (getX() == 0.0 && getY() == 0.0);  
public boolean éOrigem();
```

Quaisquer restrições que interrogações ou comandos tenham devem ser expressas nas pré-condições. Qualquer componente que surja na pré-condição deve estar disponível ao cliente. O cliente deve poder verificar se está a violar ou não as pré-condições a que está obrigado. A versão seguinte do exemplo anterior é incorrecta, pois o cliente não tem acesso aos atributos privados `x` e `y` usados na implementação.

```
//@ ensures \result == (x == 0.0 && y == 0.0);    ERRADO!  
public boolean éOrigem();
```

Para os comandos é conveniente descrever nas pós-condições qual o estado actual dos atributos modificados pelo comando através do uso das interrogações. Segue-se o pressuposto da menor alteração, assumindo que a parte da configuração não descrita na pós-condição não sofreu alterações (pode-se provocar erros não detectados se o pressuposto da menor alteração não for respeitado).

```
/*@ requires r != 0.0;
 @ requires Math.sqrt(getX()*getX()/r*r +
 @                                     getY()*getY()/r*r) <= 1;
 @ ensures eIgual(novo(\old(getX())/r, \old(getY())/r));
 */
public void dividir(double r);
```

As pré-condições determinam o conjunto de valores dos parâmetros para os quais o método funciona correctamente. O fornecedor pode assumir duas atitudes diferentes perante as restrições que impõe ao cliente. Ou uma atitude tolerante admitindo ser capaz de resolver algumas informações erradas; ou uma atitude exigente admitindo apenas informação correcta. Se a atitude do fornecedor for exigente teremos o seguinte contrato para o método `setX()`:

```
//@ requires Math.sqrt(x*x + getY()*getY()) <= 1;
//@ ensures getX() == x;
public void setX(double x);
```

Se a atitude do fornecedor for tolerante, o método aceita uma gama maior de valores tendo de descrever o que acontece nesses casos errados (neste exemplo, afirma que só modifica o valor do atributo `xx` se a invariante for aceite com esse novo valor; senão não modifica o estado do objecto):

```
//@ requires Math.abs(x) <= 1;
//@ ensures Math.sqrt(x*x+getY()*getY())<=1 ==> getX()==x;
//@ ensures Math.sqrt(x*x+getY()*getY())>1
//                                     ==> getX() ==\old(getX());
public void setX(double x);
```

Como foi referido, não se deve repetir a pré-condição no corpo do método. Um dos objectivos do contrato é precisamente eliminar a programação defensiva associada aos testes dos argumentos. A repetição desse teste no método (já testado no lado do cliente) tem como resultado mais instruções (e mais erros potenciais), mais trabalho para as escrever (e corrigir) e menor desempenho (mais instruções executadas), i.e., mais custos.

As asserções dos contratos são mecanismos de controlo que filtram as mensagens trocadas internamente pelos módulos do sistema. A pré-condição verifica a mensagem que chegou ao método enquanto a pós-condição verifica a resposta calculada pelo método e a enviar para quem o invocou. Os contratos não servem para filtrar informação que vem do exterior (por exemplo, através do teclado), esta é uma tarefa específica da classe que faz a interface entre o sistema e o ambiente.

As pré-condições devem aparecer na documentação (afinal são as restrições impostas pelo fornecedor) e devem ser justificadas através da especificação acordada pelas partes. As interrogações incluídas na pré-condição devem ser públicas para que o cliente seja capaz de verificar se a pré-condição é ou não cumprida.

## 11.4 O Desenho Revisitado

Referimos no fim do capítulo 8 que o desenho de um TDA será realizado através do conceito de interface. Uma interface Java é 100% desenho e 0% implementação. Existem alguns motivos para introduzir os contratos do tipo especificado no desenho:

- O tipo Java que reflecte o TDA é descrito por uma interface. Assim, o melhor local para descrever os contratos descritos no TDA é na definição dessa interface.
- Sendo 100% desenho, uma interface não tem semântica. O uso de invariantes e de pré e pós-condições para os métodos traz um conjunto de restrições e significados que enriquece a própria interface.

O TDA em questão determina a estrutura da interface e dos contratos:

- Uma função do TDA onde o respectivo tipo só se encontra no domínio é uma **interrogação**.
- As restantes funções são **comandos**. Os comandos são procedimentos (i.e., métodos que devolvem o tipo `void`) onde se considera que a modificação opera-se sobre o próprio objecto (designado por `this`).
- Eventualmente, algumas destas funções podem ser vistas na interface e na implementação como interrogações (em vez de modificarem o objecto `this`, devolvem um objecto novo). Se todos os comandos forem transformados desta forma, obtemos uma classe de objectos imutáveis (como a classe `String` do Java).
- Uma pré-condição do TDA surge como pré-condição do método respectivo na interface.
- Um axioma do TDA constituído apenas por interrogações reaparece como pós-condição do método correspondente. Deve-se tentar traduzir o mais fielmente possível, os axiomas do TDA para as pós-condições dos métodos na interface. Se não for possível, deve-se procurar uma expressão alternativa que descreva o comportamento da propriedade descrita pelo respectivo axioma, ou que represente uma expressão logicamente mais fraca (i.e., que seja uma implicação do axioma) que significa só descrever parte do comportamento do axioma. Em qualquer dos casos, deve-se referir o axioma em comentário. A propriedade tem sempre de ser respeitada pelo programador, mesmo que não seja possível expressá-la nos contratos.

Como exemplo, consideremos a especificação do ponto a duas dimensões referida na página 164, na qual se adiciona a restrição R do exemplo inicial deste capítulo (qualquer ponto deve estar no interior do círculo de raio 1 com centro na origem). Para actualizar o TDA altera-se a secção de pré-condições:

```
...
pré-condições
    setX(P, r) requer
```

```
    distOrigem(novo(r,getY(P))) ≤ 1
    setY(P,r) requer
        distOrigem(novo(getX(P),r)) ≤ 1
        multiplicar(P,r) requer
            distOrigem(novo(getX(P)*r,getY(P)*r)) ≤ 1
        dividir(P,r) requer
            r≠0 and
            distOrigem(novo(getX(P)/r,getY(P)/r)) ≤ 1
```

Na passagem do TDA para a interface, as assinaturas dos comandos são alteradas. Por exemplo, a função `setX : Ponto2D Real → Ponto2D` transformar-se-á no método `public void setX(double x)`. Uma das motivações para esta alteração é usarmos notação infixa no TDA – *função(objecto)* – mas usarmos notação sufixa na invocação dos métodos – *objecto.função()*. A única excepção desta regra é aplicada ao construtor do TDA que cria novos objectos que mantém a sua assinatura (ver o exemplo seguinte).

A interface ainda tem os contratos:

```
public interface Ponto2D {
    // construtores
    Ponto2D novo(double x, double y);
    // interrogações
    double getX();           // interrogações básicas
    double getY();
    boolean eOrigem();       // interrogações derivadas
    double distOrigem();
    boolean eIgual(Ponto2D p);
    // comandos
    void setX(double x);
    void setY(double y);
    void multiplicar(double r);
    void dividir(double r);
}
```

Algumas notas:

- Assume-se com a interface pelo menos duas restrições: (i) a implementação será em Java e (ii) o tipo de dados usado para armazenar a informação numérica dos valores das coordenadas é o tipo primitivo `double`. O desenho proposto pela interface refinou o TDA abstracto (i.e., as duas restrições referidas reduziram o número de potenciais implementações).
- A interface apenas descreve limitações sintácticas. Não existe semântica explícita associada à interface, mas indirectamente sabemos que a interface é um desenho

do TDA proposto. A inserção dos contratos adiciona alguma dessa semântica retida no TDA.

A interface agora com contratos:

```
public interface Ponto2D {  
    // @ public invariant distOrigem() <= 1;  
    // @ ensures (\result.getX() == x) && (\result.getY() == y);  
    Ponto2D novo(double x, double y);  
    double getX();  
    double getY();  
    // @ requires Math.sqrt(x*x + getY()*getY()) <= 1;  
    // @ ensures getX() == x;  
    void setX(double x);  
    // @ requires Math.sqrt(getX()*getX() + y*y) <= 1  
    // @ ensures getY() == y  
    void setY(double y);  
    // @ ensures \result == (getX() == 0.0 && getY() == 0);  
    boolean eOrigem();  
    /*@ ensures \result == (getX() == p.getX() &&  
     * @                                     getY() == p.getY());  
     * /  
    boolean eIgual(Ponto2D p);  
    /*@ ensures \result == Math.sqrt(getX()*getX() +  
     * @                                     getY()*getY());  
     * /  
    double distOrigem();  
    /*@ requires Math.sqrt(getX()*getX()*r*r +  
     * @                                     getY()*getY()*r*r) <= 1;  
     * @ ensures eIgual(novo(\old(getX())*r, \old(getY())*r));  
     * /  
    void multiplicar(double r);  
    /*@ requires r != 0.0;  
     * @ requires Math.sqrt(getX()*getX()/r*r +  
     * @                                     getY()*getY()/r*r) <= 1;  
     * @ ensures eIgual(novo(\old(getX())/r, \old(getY())/r));  
     * /  
    void dividir(double r);  
}
```

## 11.5 Contratos e Herança

Os contratos são herdados juntamente com a classe/interface a que estão associados. Tanto por uma subclasse como na implementação da interface numa classe concreta. Este é um aspecto bastante importante. Não faria sentido associar um conjunto de asserções a um tipo e não o forçar aos tipos derivados. Esta atenção é descrita pelo seguinte **Princípio da Substituição**:

*Um objecto de uma classe estendida deve ser usado correctamente em todos os contextos onde a classe original estava prevista.*

Consequências deste princípio:

- Todas as invariantes têm de ser respeitadas (conjunção das invariantes).
- As pré-condições dos métodos de uma derivação são iguais ou mais fracas. A derivação só pode assumir menos nunca mais (disjunção das pré-condições).
- As pós-condições dos métodos de uma derivação são iguais ou mais fortes. A derivação só pode prometer mais nunca menos (conjunção das pós-condições).

Ou seja, a classe derivada não pode pedir mais nem oferecer menos.

No exemplo geométrico dos capítulos anteriores, um Rectângulo tem obrigatoriamente de cumprir todas as restrições (i.e., pré-condições e invariantes) de um Polígonos, já que nunca deixou de ser um Polígonos. Porém, pode possuir restrições mais fortes por ser um Rectângulo: não são dadas garantias algumas que um qualquer Polígonos possa ser usado em todos os contextos onde um Rectângulo é usado.

Tendo em conta as promessas (i.e., pós-condições) que assume, um Rectângulo nunca pode prometer menos que um Polígonos, pois violaria a relação de herança (e o princípio de substituição). Sendo um Polígonos por derivação, um Rectângulo tem que garantir, no mínimo, os mesmos serviços. No entanto nada proíbe que um Rectângulo disponibilize uma maior gama de funcionalidades que o Polígonos. Por exemplo, se o Polígonos tivesse um método para calcular o perímetro, este poderia ser usado por todos os rectângulos. Já um método específico ao Rectângulo, por exemplo calcular o comprimento da diagonal, não poderia ser usado por qualquer objecto Polígonos. Como se calcularia a diagonal de um Triângulo (que também é um Polígonos)?

Ao trabalhar com contratos, é formalizada uma das componentes implícitas no mecanismo da herança: a herança do tipo descrita no princípio da substituição. Porém, para o problema geral, com expressões lógicas arbitrárias no subtipo e supertipo, seria necessário um avaliador de teoremas demasiado complexo para ser prático. Por isso o JML realiza automaticamente a disjunção das pré-condições e a conjunção das pós-condições que garante o princípio da substituição (' $A \vee B$ ' é sempre igual ou mais fraco que  $A$ , ' $A \wedge B$ ' é sempre igual ou mais forte que  $A$ ).

Esta abordagem, para alguns é considerada insatisfatória porque restringe demasiado o programador na escrita dos contratos do subtipo (dado ter de pensar na conjunção das pré e na disjunção das pós para que estas resultem no contrato desejado do tipo em questão). Outras ferramentas de contratos (como o Jass) tem uma atitude diferente. No caso do Jass, por exemplo, o programa avalia um conjunto definido de expressões, como ‘ $\text{Pré}_{\text{supertipo}} \wedge \neg \text{Pré}_{\text{subtipo}}$ ’, que sendo verdadeiras indicam violações do princípio de substituição.

Como há formas diferentes de interpretar o princípio de substituição, é importante consultar como funciona a linguagem de asserções escolhida.

---

## Exercícios

- Verifique que o seguinte código fonte satisfaz as asserções associadas.

```
i=0; s=0;
{s == Σk=0i k} ∧ 0 ≤ i ≤ 99
while (i < 99) {
    s += (i+1);
    i++;
}
{s == Σk=099 k}
```

- Desenvolva um comando Java, usando a metodologia de Dijkstra, que dado um real  $x$  e um inteiro positivo  $n$ , calcule  $x^n$ .

- Considere a seguinte interface. (i) escreva os contratos apropriados; (ii) defina uma classe que implemente este tipo.

```
public interface Counter {

    /**
     * Valor do contador.
     */
    int value()

    /**
     * Incrementar uma unidade ao contador.
     */
    void inc()

    /**
     * Colocar o contador com um dado valor.
     * @param v novo valor do contador.
     */
    void reset(int v)
}
```

4. Considere a classe que representa um número racional definida no exercício 23 no capítulo 4. Escreva os contratos apropriados para esta classe.

5. Implemente a classe Time com as seguintes operações (inclua os contratos necessários):

```
/**
 * Construir um tempo a partir das horas e minutos.
 * @param hour as horas iniciais
 * @param min os minutos iniciais
 */
public Time(int hour, int min)

/**
 * Construir um tempo a partir de outro tempo.
 * @param t o outro tempo
 */
public Time (Time t)

/**
 * Devolver as horas.
 */
public int hour()

/**
 * Devolver os minutos.
 */
public int minutes()

/**
 * Devolver uma representação textual do objecto.
 */
public String toString()

/**
 * Avançar um minuto.
 */
public void forth()

/**
 * Retroceder um minuto.
 */
public void back()

/**
 * Avançar um determinado número de minutos.
 * @param minutes número de minutos a avançar
 */
public void add(int minutes)
```

```
/**  
 * Retroceder um determinado número de minutos  
 * @param minutes número de minutos a retroceder  
 */  
public void subtract(int minutes)  
/**  
 * Número de minutos até um determinado tempo  
 * @param t tempo em relação à qual é determinado o número  
 *         de minutos em relação ao próprio objecto.  
 */  
public int minutesTo(Time t)  
/**  
 * Número de minutos desde um determinado tempo  
 * @param t tempo a partir do qual é determinado o número  
 *         de minutos até ao próprio objecto  
 */  
public int minutesSince(Time t)  
/**  
 * Horas iguais ?  
 * @param t tempo utilizado na comparação  
 */  
public boolean equals(Object t)
```

6. Considere a classe descrita no exercício 30 do capítulo 4 (a classe que representa uma sequência de dígitos). Assuma que os cabeçalhos dos métodos da classe são os que se seguem. Escreva os contratos destes métodos.

```
/**  
 * Criar uma sequência vazia.  
 */  
public DigitSequence()  
/**  
 * Comprimento da sequência.  
 */  
public int length()  
/**  
 * I-ésimo elemento da sequência.  
 * @param i índice do elemento pretendido.  
 */  
public int ith(int i)
```

```
/**  
 * Adicionar um dígito à direita da sequência.  
 * @param digit dígito a adicionar.  
 */  
public void addRight(byte digit)  
/**  
 * Adicionar um dígito à esquerda da sequência.  
 * @param digit dígito a adicionar.  
 */  
public void addLeft(byte digit)  
/**  
 * Verificar se uma dada sequência é prefixo da sequência.  
 * @param seq sequência prefixo.  
 */  
public boolean prefix(DigitSequence seq)  
/**  
 * Verificar se uma dada sequência é sufixo da sequência.  
 * @param seq sequência sufixo.  
 */  
public boolean sufix(DigitSequence seq)  
/**  
 * Verificar se seq é subsequência desta sequência.  
 * @param seq subsequência.  
 */  
public boolean subsequence(DigitSequence seq)  
/**  
 * Concatenar uma sequência à sequência.  
 * @param seq subsequência a concatenar.  
 */  
public void concat(DigitSequence seq)  
/**  
 * Sequência igual à sequência?  
 * @param seq sequência a comparar.  
 */  
public boolean equals(Object seq)  
/**  
 * Cópia da sequência.  
 */  
public DigitSequence copy()
```

# **12 – Programação por Eventos**

---

Um **evento** (em inglês, *event*) é algo que ocorre numa determinada posição e num determinado momento. Quando se fixa uma das dimensões obtém-se **intervalos** (conjuntos de eventos que ocorrem num período fixo de tempo) e **locais** (conjuntos de eventos que ocorrem num espaço determinado). O **estado** de um sistema representa um intervalo onde não há alterações de comportamento desse sistema. Uma **mudança de estado** (do inglês, *state transition*) ocorre quando um evento modifica o comportamento do sistema, alterando, assim, o seu estado.

Na programação por eventos, um evento desencadeia reacções sobre um ou mais objectos, eventualmente modificando o estado desses objectos. Essas modificações podem produzir novos eventos, provocando uma reacção em cadeia de relações causa/efeito. Esta sequência de acções e reacções determina, em parte, a dinâmica do sistema. Dois exemplos:

- Num ambiente procedural, o utilizador inicia uma aplicação e aguarda pelo fim da sua execução. Quando há necessidade de pedir informação ao utilizador, é invocado um método apropriado. A execução do programa bloqueia enquanto não se recebe a informação desejada.
- Num ambiente gráfico, a aplicação encontra-se à “espera” de um evento válido. O sistema operativo gera todos os eventos que ocorrem no computador e envia para a aplicação os eventos que lhe dizem respeito. A aplicação tem de ser capaz de “ouvir” (i.e., receber) e processar esses eventos adequadamente.

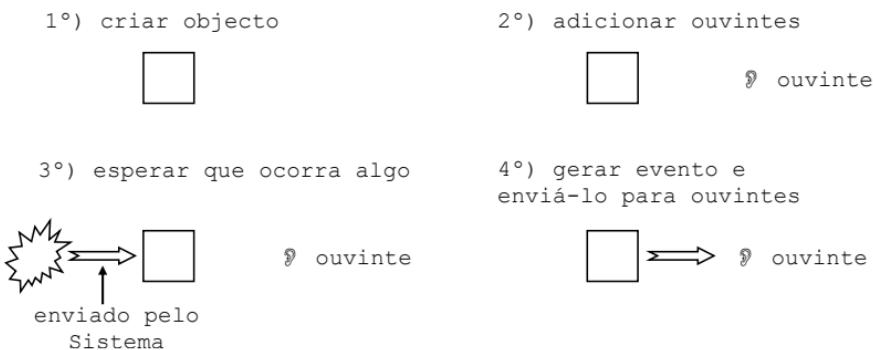
Estas situações mostram dois tipos de eventos possíveis:

- Chamadas (ou invocações) – o envio de uma mensagem **síncrona**, i.e., o objecto emissor fica à espera de uma resposta para continuar. É este o processo das invocações dos métodos Java.
- Sinais – o envio de uma mensagem **assíncrona**, i.e., o objecto emissor não espera uma resposta do objecto receptor e continua o seu processamento.

## 12.1 Eventos em Java

Numa aplicação Java, os eventos estão relacionados com certas classes. Quando um determinado acontecimento ocorre sobre um objecto X, é gerado o evento correspondente que é enviado para quem “ouve” X. Cada um destes ouvintes recebe o evento e processa-o conforme as suas necessidades.

Este mecanismo é descrito no seguinte diagrama:



O programador preocupa-se com a primeira e a segunda parte. A máquina virtual Java (conjuntamente com o sistema operativo) gera os detalhes da terceira e da quarta parte.

Por exemplo, um botão em Java (i.e., um objecto da classe `Button`) gera eventos do tipo `actionEvent` (um evento deste tipo é gerado quando alguém carrega no botão). A classe `Button` inclui o método `addActionListener()` que adiciona um objecto ouvinte (que tem de ser do tipo `ActionListener`).

Definiremos um *applet* para exemplificar os conceitos mais relevantes. Um *applet* é um objecto destinado a ser executado dentro de uma aplicação<sup>23</sup> (normalmente, num

<sup>23</sup> Existem restrições sérias ao uso de *applets*. Não é possível aceder (nem ler, nem escrever) na memória secundária local (isto evita problemas de segurança, como ler informação confidencial ou inserir um vírus no computador). Um applet de maiores dimensões pode levar muito tempo a

*web-browser* como o *Internet Explorer*). É normal usar *applets* como programas gráficos interactivos através da Internet. Para criar um *applet* define-se uma classe pública que estende a classe *Applet*:

```
public class Exemplo extends Applet { ... }
```

Alguns dos objectos capazes de gerar eventos pertencem a classes do pacote gráfico AWT (do inglês, *Abstract Windows Toolkit*) do Java<sup>24</sup> que usaremos aqui como exemplo.

```
import java.applet.Applet;
import java.awt.*;
public class Exemplo extends Applet {
```

Apresentaremos neste exemplo três tipos de objectos: (i) etiquetas (classe *Label*), (ii) caixas de texto (classe *TextField*) e (iii) botões (classe *Button*). É criada inicialmente uma referência para cada objecto:

```
Label    l1;
TextField t1;
Button   b1;
```

Num *applet* o método de classe *main()* é substituído pelo método *init()* invocado automaticamente no início da execução do *applet*. É neste método que criamos os objectos:

```
public void init() {
    l1 = new Label("Texto:");
    t1 = new TextField(10);
    b1 = new Button("Premir");
```

É ainda necessário informar que os objectos devem ser adicionados na componente visual respectiva (neste caso, no próprio *applet*):

---

carregar porque todos os recursos (as classes, os ficheiros de som, imagem, etc.) têm de ser transferidos para o computador local (mas podem ser compactados pela ferramenta JAR disponível no JDK – ver anexo A). Existem, porém, algumas vantagens. Não é necessário qualquer instalação, os *applets* podem ser executados via Internet por quase todos os *browsers*, são independentes do sistema operativo usado e não há qualquer perigo de provocar prejuízos noutros computadores.

<sup>24</sup> Este pacote disponibiliza um amplo conjunto de classes para construção de aplicações gráficas. Não entraremos em detalhes sobre este pacote que contém um elevado número de classes e funcionalidades, nem falaremos do novo pacote gráfico *Swing* incluído no Java 2 (não é essa a intenção do capítulo).

```
    add(l1);
    add(t1);
    add(b1);
}
} //endClass Exemplo
```

Neste momento já é possível compilar a classe, produzindo o ficheiro `Exemplo.class`. Este pode ser incluído num ficheiro HTML como o seguinte<sup>25</sup>:

```
<HTML>
<HEAD>
<TITLE>Teste</TITLE>
</HEAD>
<BODY>
<APPLET code="Exemplo"
         width=500
         height=200>
</APPLET>
</BODY>
</HTML>
```

Quando a página HTML é lida pelo *browser*, é criado automaticamente um objecto da classe `Exemplo`, aparecendo os três objectos criados durante a invocação do método `init()`.

Ao experimentar o *applet* verifica-se ser possível alterar o conteúdo da caixa de texto, mas quando se carrega no botão nada acontece. Porquê? Primeiro porque nada foi dito sobre o que o botão deve fazer e segundo porque não se definiu o ouvinte dos eventos gerados pelo botão. Neste caso, o ouvinte será o próprio *applet*.

Para o *applet* ser capaz de ouvir eventos gerados por botões tem de implementar a interface `ActionListener`. Existem diferentes tipos de ouvintes dependendo do tipo de objectos que geram os eventos. Os tipos `MouseListener` e `MouseMotionListener` ouvem eventos do tipo `MouseEvent` produzidos pelo rato (respectivamente, ao premir os botões do rato e ao mover o rato). O tipo `WindowListener` ouve eventos do tipo `WindowEvent` produzidos por janelas. O tipo `KeyListener` ouve eventos do tipo `KeyEvent` produzidos pelo teclado. Estas interfaces são derivadas da interface `EventListener` e incluídas no pacote `java.awt.event`.

```
import java.applet.Applet;
import java.awt.*;
```

---

<sup>25</sup> Se a classe `Exemplo` pertencer ao pacote X e o ficheiro HTML encontrar-se na directória que contém a directória X, então deve-se escrever `<APPLET code="X.Exemplo" ...>`.

```
import java.awt.event.*;
public class Exemplo extends Applet
    implements ActionListener { ... }
```

A interface `ActionListener` inclui o método `actionPerformed()`. Este método define como são tratados os eventos do tipo `ActionEvent` (gerados por objectos como botões e caixas de texto). No seguinte exemplo, quando se prima o botão é adicionado um ponto de exclamação à caixa de texto:

```
public void actionPerformed(ActionEvent event) {
    if (event.getSource() == b1) {
        t1.setText(t1.getText() + "!");
    }
}
```

O método `getSource()` devolve a referência do objecto que gerou o evento. Se for o botão referenciado por `b1`, adiciona-se o ponto de exclamação através das invocações dos métodos `setText()` e `getText()` incluídos na classe `TextField`.

Incluindo este método na classe, compilando e lendo novamente a página HTML, verificamos que o botão continua sem funcionar. Porquê? Porque apenas foi dito que o *applet* é capaz de processar eventos gerados por botões. O objecto *applet* ainda não é um ouvinte do objecto botão referenciado por `b1`. Isso é realizado através do método `addActionListener()`:

```
public void init() {
    l1 = new Label("Texto:");
    t1 = new TextField(10);
    b1 = new Button("Click");
    add(l1);
    add(t1);
    add(b1);
    b1.addActionListener(this);
}
```

A partir deste momento, o evento gerado pelo botão é ouvido pelo *applet* que executa o método `actionPerformed()`. É possível realizar a operação inversa, remover um ouvinte, através do método `removeActionListener()`.

O botão não “sabe” o que tem de ser feito quando for premido. Ocorre um processo de **delegação** (do inglês, *delegation*) dessa tarefa noutros objectos (neste exemplo, no *applet*) cujas referências são recebidas através das invocações do método `addActionListener()`. Quando um evento deste tipo relacionado com o botão é gerado, há um processo de **rechamada** (do inglês, *callback*) dos objectos delegados de

modo a executarem as respectivas acções (neste exemplo, a invocação do método `actionPerformed()` do objecto *applet*).

Segue o código fonte completo:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class Exemplo extends Applet
    implements ActionListener {
    Label l1;
    TextField t1;
    Button b1;

    public void init() {
        l1 = new Label("Texto:");
        t1 = new TextField(10);
        b1 = new Button("Click");

        add(l1);
        add(t1);
        add(b1);

        b1.addActionListener(this);
    }

    public void actionPerformed(ActionEvent event) {
        if (event.getSource() == b1) {
            t1.setText(t1.getText() + "!");
        }
    }
} //endClass Exemplo
```

## Eventos do Rato

O rato é uma ferramenta essencial para interagir graficamente com uma aplicação. No contexto dos eventos em Java, o rato produz eventos do tipo `MouseEvent`. Os objectos capazes de ouvir estes eventos têm de implementar a interface `MouseListener`. Finalmente, para o *applet* ser capaz de ouvir eventos deste tipo, tem de invocar o método `addMouseListener()`.

```
public class Exemplo2 extends Applet
    implements MouseListener { ... }
```

A interface contém os seguintes métodos (que têm de ser implementados pela classe):

- `mouseClicked()` – invocado quando um componente gráfico é “clicado”;

- `mouseEntered()` – invocado quando o rato entra num componente gráfico;
- `mouseExited()` – invocado quando o rato sai de um componente gráfico;
- `mousePressed()` – invocado quando um botão do rato é premido num componente;
- `mouseReleased()` – invocado quando um botão do rato é solto num componente;

Pretendemos que este *applet* possua uma caixa de texto e que quando um botão do rato for premido sejam aí colocadas as coordenadas do rato nesse instante.

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class Exemplo2 extends Applet
    implements MouseListener {
    TextField t1;

    public void init() {
        t1 = new TextField(10);
        add(t1);
        addMouseListener(this); // o applet passa a ser ouvinte
                               // de eventos do tipo MouseEvent
    }
}
```

Seguem os vários métodos da interface `MouseListener`. Neste caso, interessa apenas implementar o método `mousePressed()`:

```
public void mouseClicked (MouseEvent event) {};
public void mouseExited (MouseEvent event) {};
public void mouseEntered (MouseEvent event) {};
public void mouseReleased (MouseEvent event) {};

public void mousePressed (MouseEvent event) {
    t1.setText("x: " + event.getX() + " "
               + event.getY());
}
} //endClass Exemplo2
```

Para obter o pretendido, invocou-se os métodos `getX()` e `getY()` que devolvem as coordenadas do rato (relativas à janela activa da aplicação) no instante da criação do evento. Outro método, `getClickCount()`, indica o número de vezes que o botão foi premido durante o evento (simples, duplo ou triplo).

É também possível determinar se algumas teclas são premidas no momento do evento. No exemplo seguinte, o método `isShiftDown()` é verdadeiro se a tecla SHIFT estivesse activada no instante em que o botão do rato foi premido:

```
public void mousePressed(MouseEvent event) {  
    if (event.isShiftDown())  
        t1.setText("SHIFT");  
    else  
        t1.setText("x: " + event.getX() + " "  
                  + event.getY());  
}
```

## 12.2 Máquinas de Estados

O estado de um objecto é definido pelos valores dos seus atributos. A configuração da classe define os estados possíveis enquanto o contrato da classe define os estados válidos. Em muitos casos, o comportamento de um objecto depende do seu estado actual. Por exemplo, um objecto do tipo `Telemóvel` quando invocado o método `ligarNúmero()` comporta-se de forma diferente se o seu atributo `crédito` for diferente ou igual a zero. Existem numerosos exemplos: interruptores, canetas retractáveis, máquinas de bilhetes dos transportes públicos, máquinas de lavar, ... Por vezes é útil associar um conjunto de estados (válidos) com o mesmo comportamento. No exemplo anterior, os estados válidos de um telemóvel poderiam ser divididos em dois grandes conjuntos: `COMCRÉDITO` e `SEMCRÉDITO`.

Uma **máquina de estados** (em inglês, *finite state machine*) é um sistema definido por um conjunto de estados<sup>26</sup> (onde se destacam um estado inicial e um subconjunto de estados finais) e por um conjunto de **transições** (do inglês, *transitions*). Uma transição é constituída por:

- Um evento – o evento que desencadeou a transição,
- Um estado origem – o estado do sistema antes do evento,
- Um estado destino – o estado do sistema depois do evento,
- Uma guarda (opcional) – uma expressão booleana que se for falsa, impede a transição (isto significa a existência de várias transições associadas ao mesmo evento assumindo guardas mutuamente exclusivas<sup>27</sup>),

---

<sup>26</sup> Nesta secção há dois significados para a palavra “estado”: (i) o estado do objecto, i.e., o conjunto dos valores dos seus atributos e (ii) o estado do sistema modelado pela máquina de estados. A noção (ii) de estado, aplicada às classes, refere-se a um conjunto de estados válidos do tipo (i) que partilham um mesmo comportamento.

<sup>27</sup> Se a condição de exclusividade não for respeitada, é possível um conflito entre dois ou mais eventos. Neste caso, a máquina de estados necessita de um algoritmo que decida qual das transições a seguir. Se o processo de decisão não depender exclusivamente do estado actual e do(s) evento(s) em questão, a máquina de estados diz-se **não determinista**.

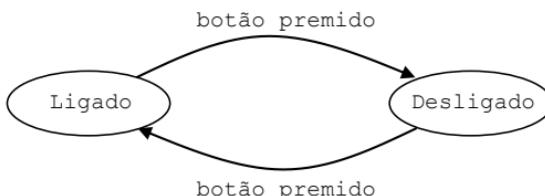
- Uma **acção** (opcional) executada pelo sistema após a saída do estado origem e antes da entrada no estado destino. Uma acção é um bloco de instruções **atómico**, i.e., a sua execução não pode ser interrompida (por exemplo, se ocorrer um evento entretanto, este terá de esperar pelo fim da acção actual).

A máquina de estados termina se atingir um estado final. Se ocorrer um evento inesperado num dado estado (i.e., não existe uma transição associada) é lançada uma exceção.

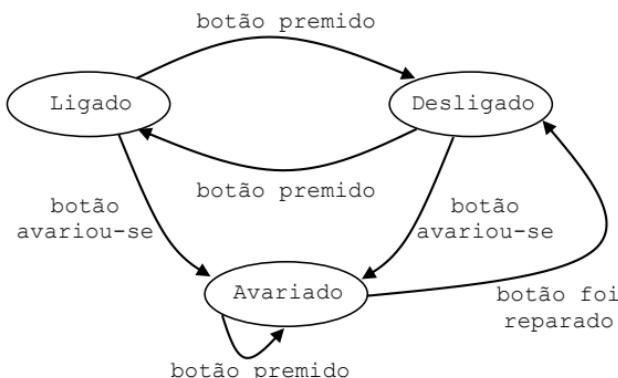
A máquina de estados serve, assim, para modelar o **ciclo de vida** de um objecto (do inglês, *object life cycle*). Para além do conjunto dos seus estados válidos, é definido o conjunto de transições possíveis desde a sua criação até à sua eventual destruição.

### Um exemplo de interruptor

Vejamos um interruptor descrito por uma máquina de estados. Neste caso simplificado, não há guardas, acções, nem estados finais. O estado inicial é DESLIGADO.



No diagrama, o sistema possui dois estados – LIGADO e DESLIGADO – e um tipo de evento – o botão foi premido (por alguém...). Consoante o estado actual, o comportamento do interruptor é diferente (por exemplo, liga ou desliga o objecto associado). A máquina de estados pode ser aperfeiçoada para modelar um sistema mais realista:



Quando se deve usar uma máquina de estados? Quando tem de se lidar com a existência de eventos e o comportamento do objecto em questão depende da ordem de chegada desses mesmos eventos. Nestes casos é interessante definir o comportamento de uma classe baseado numa máquina de estados.

Como exemplo apresentamos uma implementação do interruptor descrito pela máquina de estados anterior. Neste exemplo, um objecto interruptor executa uma dada acção consoante o seu estado.

```
abstract public class TInterruptor { // TInterruptor.java
    abstract protected int getId();
    public void executarAccao(Object ob) {
        System.out.println(ob);
    }
}
```

Esta primeira classe é abstracta mas já define a execução padrão (imprimir um dado objecto). O objectivo do método abstracto `getId()` é devolver um valor numérico correspondente ao estado (`LIGADO`, `DESLIGADO` ou `AVARIADO`) em que o objecto em questão se encontra (as classes concretas que estendam `TInterruptor` têm de implementar `getId()`).

A classe `Interruptor` estende a classe `TInterruptor` com o comportamento descrito pela máquina de estados anterior. Os diferentes estados são descritos em classes interiores. Cada classe interior possui uma implementação própria dos métodos que descrevem essas diferenças de comportamento.

```
public class Interruptor extends TInterruptor {
    class Int_DESLIGADO extends TInterruptor {
        protected int getId() { return 0; }
        public void executarAccao(Object ob) {}
    }
}
```

Quando o interruptor muda para o estado `DESLIGADO` não há acção associada. Convencionou-se que `DESLIGADO` corresponde ao número zero. Esta numeração é essencial quando se descrever a máquina de estados.

```
class Int_LIGADO extends TInterruptor {
    protected int getId() { return 1; }
}
```

Como não foi reimplementado, o interruptor no estado `LIGADO` possui o comportamento padrão `executarAccao()` descrito em `TInterruptor`.

```
class Int_AVARIADO extends TInterruptor {  
    protected int getId() { return 2; }  
    public void executarAccao(Object ob) {  
        System.out.println("#%$&%%#!$&");  
    }  
}
```

No estado AVARIADO, o interruptor imprime “lixo” independentemente do objecto dado.

```
protected final Int_LIGADO    LIGADO  
    = new Int_LIGADO();  
protected final Int_DESLIGADO DESLIGADO  
    = new Int_DESLIGADO();  
protected final Int_AVARIADO AVARIADO  
    = new Int_AVARIADO();
```

São definidos três objectos, um para cada estado. As referências para esses objectos são finais e não podem ser alteradas. Em cada instante, um e um só destes três objectos será utilizado para definir o comportamento do interruptor (consoante o seu estado).

```
/* A Maquina de Estados.  
 * Assume-se os seguintes valores para os eventos:  
 * 0 - botao premido  
 * 1 - interruptor avariou-se  
 * 2 - interruptor foi reparado  
 */  
TInterruptor FSM[][] = {  
    { LIGADO,    AVARIADO, null      },  
    { DESLIGADO, AVARIADO, null      },  
    { AVARIADO,  null,       DESLIGADO }  
};
```

A descrição da máquina de estados segundo o diagrama da página 239. O primeiro índice da matriz diz respeito ao valor numérico associado ao estado (0 para DESLIGADO, 1 para LIGADO e 2 para AVARIADO). O segundo índice da matriz diz respeito ao valor numérico associado ao evento que ocorreu (0 botão premido, 1 avaria, 2 reparação). As referências nulas dizem respeito a situações que não podem ocorrer (por exemplo, se o botão não está avariado, não é suposto receber um evento de reparação).

```
TInterruptor estadoActual = DESLIGADO;
```

Este atributo indica qual dos objectos (DESLIGADO, LIGADO ou AVARIADO) define o comportamento actual. Ao convencionar DESLIGADO como estado inicial, o atributo estadoActual é inicializado com a referência do objecto DESLIGADO.

```
public void processarEvento(int esteEvento)
    throws InvalidTransitionException {
    int esteEstado = getId();
    estadoActual = FSM[esteEstado][esteEvento];
    if (estadoActual == null)
        throw new InvalidTransitionException(
            "Evento " + esteEvento +
            " inesperado no estado " + esteEstado );
}
```

A partir da informação contida na matriz, o método `processarEvento()` dado o índice do estado actual e o valor associado ao novo evento, determina o próximo estado. Caso o evento seja inesperado (a nova posição da tabela armazena a referência nula) é lançada a excepção indicada.

```
protected int getId() {
    return estadoActual.getId();
}

public void executarAccao(Object ob) {
    estadoActual.executarAccao(ob);
}

} //endClass Interruptor
```

Um exemplo de aplicação:

```
public class Start {
    public static void main(String[] args) {
        Interruptor i = new Interruptor();
            // após a criação, o botão está desligado
        try {
            i.executarAccao(new Integer(1));
            i.processarEvento(0);           // botão premido
            i.executarAccao(new Integer(2));
            i.processarEvento(0);           // botão premido
            i.executarAccao(new Integer(3));
            i.processarEvento(0);           // botão premido
            i.executarAccao(new Integer(4));
            i.processarEvento(1);           // botão avariou-se
            i.executarAccao(new Integer(5));
```

```
i.processarEvento(0);           // botão premido
i.executarAccao(new Integer(6));
i.processarEvento(2);           // botão reparado
i.executarAccao(new Integer(7));
i.processarEvento(0);           // botão premido
i.executarAccao(new Integer(8));
i.processarEvento(2);           // botão reparado???
i.executarAccao(new Integer(9));

} catch (InvalidTransitionException e) {
    System.out.println("ERRO: " + e.getMessage());
}

} //endClass Start

□ 2 ↵
4 ↵
#%$&%%#!$& ↵
#%$&%%#!$& ↵
8 ↵
ERRO: Evento 2 inesperado no estado 1 ↵
```

---

## Exercícios

1. Um rádio possui três controles: (i) um botão para ligar/desligar, (ii) um botão para inicializar a sintonia e (iii) um botão para procurar a próxima frequência. Quando o rádio está ligado, o botão de inicializar sintoniza a banda dos 88 MHz. O botão de procurar sintoniza a próxima estação (assuma que existe uma a cada 10 MHz) ou pára se chegar aos 110 MHz. Modele uma máquina de estados e defina uma classe que a implemente.
2. Considere um soldado a guardar uma base. O seu estado inicial é de vigia. Se localizar algo estranho, passa para uma fase de ataque. Se eliminar o alvo ou este desaparecer volta ao estado inicial, mas se for atingido procura esconder-se para recuperar as feridas. Caso consiga esconder-se e recuperar, volta ao estado de vigília inicial. Se não conseguir recuperar-se, morre. Construa uma máquina de estados que descreva este problema.
3. Uma máquina de bebidas guarda três tipos de latas, cada uma ao custo de 75 céntimos. A máquina recebe moedas de 25 e 50 céntimos e de 1 ou 2 Euros. A lotação máxima da máquina é de 100 latas de cada bebida. Encontre os estados e os eventos necessários e defina a classe que implemente o controle deste tipo de máquina.



# PARTE III

## Algoritmos e Estruturas de Dados



# **13 – Análise de Algoritmos**

---

Nem todos os programas requerem o mesmo esforço computacional, seja em tempo seja em memória. Entre programas que resolvem o mesmo problema há uns mais exigentes que outros. Quais as medidas dessa exigência? Existem vários critérios para medir um programa. Por exemplo, seria possível criar uma classificação baseada no número de instruções ou nas transferências de memória secundária para memória primária. Vamos utilizar os dois conceitos considerados mais relevantes: o **tempo** que o programa demora a ser executado e o **espaço** em memória que o programa necessita para ser executado. A avaliação destes critérios dá-nos a **complexidade temporal** e a **complexidade espacial** do programa em questão. Por omissão, o termo complexidade referir-se-á à complexidade temporal.

Um exemplo: seja o problema da soma de uma sequência de  $N$  números positivos. Usando um algoritmo aprendido na escola primária, a soma de  $N$  inteiros pode ser realizada a partir da soma dos respectivos dígitos das unidades, depois das dezenas (à qual se acrescenta o excesso da primeira soma) e assim sucessivamente. Qual a memória necessária? Para além dos dados iniciais (os  $N$  números) precisamos de armazenar o excesso da soma actual bem como o resultado final. Qual o tempo necessário? Com  $P$  dígitos para somar precisamos de tempo suficiente para realizar aproximadamente  $P \times N$  somas. Determinamos assim, que o problema possui exigências proporcionais à dimensão dos dados iniciais do problema.

## **13.1 Introdução**

Existem diferentes formas de avaliar as exigências de um programa:

- Empiricamente – observando a execução efectiva do programa;

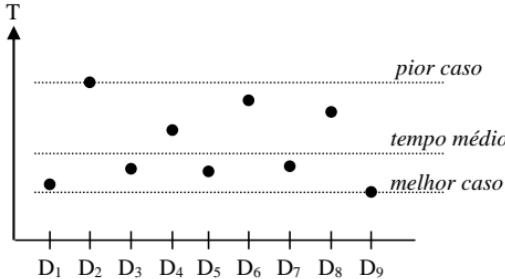
- Experimentalmente – criando um modelo simplificado do problema de forma a estimar o comportamento futuro;
- Analiticamente – demonstrando a existência de uma função matemática que calcule as exigências do programa em relação aos parâmetros do problema.

Considere um problema P com parâmetros D e um programa que resolve P. Após N experiências com a mesma dimensão de dados mas com valores diferentes é possível estimar os recursos gastos em termos do melhor caso, do pior caso ou da média dos casos observados.

Por exemplo, considere um programa para ordenar (em ordem crescente) um conjunto de elementos guardados num vector. Pode-se obter informação observando o tempo gasto pelo programa a ordenar diferentes vectores. Escolhemos um número de exemplos para utilizar na experiência:

```
int[] D1 = {1,2,3,6,4,5};
int[] D2 = {6,5,4,3,2,1};
int[] D3 = {2,3,5,6,1,4};
int[] D4 = {4,5,1,6,3,2};
int[] D5 = {1,2,5,6,3,4};
int[] D6 = {6,5,1,2,4,3};
int[] D7 = {1,2,5,4,3,6};
int[] D8 = {5,4,1,6,3,2};
int[] D9 = {1,2,3,4,5,6};
```

Todos os vectores possuem o mesmo número de elementos porque pretendemos entender como o programa se comporta com diferentes instâncias do problema da mesma dimensão. Ao executar o programa, mede-se o tempo gasto em cada experiência obtendo-se o seguinte gráfico:



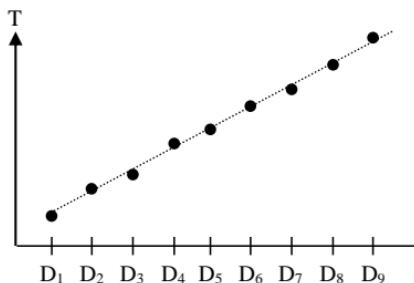
Neste conjunto de experiências, o pior tempo foi obtido na ordenação do vector D2 (o vector estava ordenado ao contrário do pretendido) e o melhor tempo no vector D9 (o vector já estava ordenado correctamente).

Mas e se variarmos a dimensão de D? Na maior parte dos casos, o tempo gasto pelo programa na resolução tende a crescer com a dimensão de D. Quanto maior for o conjunto dos dados iniciais mais complicado se torna – em princípio – encontrar a solução.

No exemplo da ordenação, escolhe-se um novo conjunto de exemplos (o pior caso para progressivas dimensões do problema) a utilizar numa segunda experiência:

```
int[] D1 = {2,1};
int[] D2 = {3,2,1};
int[] D3 = {4,3,2,1};
int[] D4 = {5,4,3,2,1};
int[] D5 = {6,5,4,3,2,1};
int[] D6 = {7,6,5,4,3,2,1};
int[] D7 = {8,7,6,5,4,3,2,1};
int[] D8 = {9,8,7,6,5,4,3,2,1};
int[] D9 = {10,9,8,7,6,5,4,3,2,1};
```

Comparando os vários tempos:



O que se pode observar? Que o tempo de execução  $T$  gasto pelo programa cresce linearmente com a dimensão de  $D$  (a tendência é representada pela recta pontilhada). A análise que se baseia neste tipo de raciocínio estuda a **complexidade assintótica**.

O tempo de execução depende de muitos factores externos. Por exemplo, a arquitectura do processador usado, a velocidade do relógio do computador, a velocidade de acesso à memória primária e secundária, o sistema operativo, a linguagem usada, o compilador, etc. No entanto, há uma particularidade extremamente relevante. Observemos novamente o gráfico do tempo. A taxa de crescimento do tempo é proporcional ao crescimento linear descrito pelo declive da recta pontilhada. Um computador mais rápido pode baixar esse declive (realiza a tarefa em menos tempo) mas não pode alterar a relação de proporcionalidade. A taxa de crescimento temporal será linear quaisquer que forem os recursos usados para a execução do programa. Este facto é uma consequência da seguinte propriedade:

*O tempo para a correcta resolução de um problema é determinado pelos recursos computacionais disponíveis, mas a taxa de crescimento temporal em relação à dimensão dos parâmetros iniciais depende, única e exclusivamente, do algoritmo utilizado.*

A existência desta característica leva-nos à avaliação analítica de algoritmos. Em muitos casos, é possível calcular uma medida exacta da taxa de crescimento temporal de um algoritmo. Não precisamos estar preocupados com os detalhes técnicos referentes aos suportes físicos e lógicos utilizados.

Consideram-se três situações distintas: (i) o melhor caso possível, (ii) o pior caso possível e (iii) o caso médio, onde se assume uma distribuição dos dados iniciais (a distribuição real pode não ser conhecida). Normalmente, opta-se pelo cálculo do pior caso pois fornece o tempo máximo gasto pelo algoritmo (uma garantia para quem utiliza o respectivo programa) e é uma situação comum (por exemplo, procurar num ficheiro ou num vector por uma informação que não existe).

## 13.2 Notação O

Vamos exprimir a complexidade assimptótica através do uso da notação O (ó grande):

### Definição:

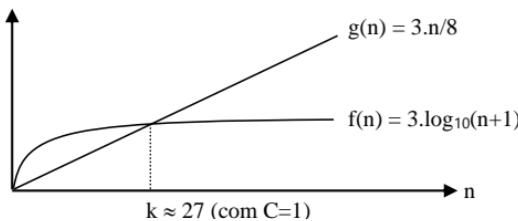
Considere  $f$  e  $g$  duas funções de domínio inteiro não negativo e contradomínio real.

Diz-se que  $g$  domina assintoticamente  $f$ , escrevendo-se  $f \in O(g)$  ou  $f$  é da ordem de  $O(g)$ , se e só se,

$$\exists_{k>0} \exists_{C>0} \forall_{n>0} : n>k \Rightarrow |f(n)| \leq C|g(n)|$$

Existem constantes  $k$  natural e  $C$  real tais que, para todo o argumento  $n>k$ , a função  $g$  multiplicada pela constante  $C$  é maior que  $f$ . Isto determina um limite superior ao crescimento da função  $f$ . De notar que  $O(g)$  representa o conjunto de todas as funções limitadas superiormente por  $g$ . Diz-se que  $O(g)$  é uma **classe de complexidade**. Sendo  $O(g)$  um conjunto de funções, pode-se escrever  $f$  é  $O(g)$  como  $f \in O(g)$  ou ainda  $O(f) \subseteq O(g)$ .

Considere o seguinte exemplo com a função  $g(n) = 3.n/8$  e a função  $f(n) = 3.\log_{10}(n+1)$ . Com  $n=1$ , a função  $f$  é maior que  $g$ . Porém, há medida que  $n$  aumenta, a função  $g$  cresce mais depressa e, inevitavelmente, chega a um valor (por volta de  $n=27$ ) a partir do qual é sempre maior. Assim, é possível definir um valor para  $k$  (um qualquer valor maior que 27) de modo a que a expressão lógica da definição anterior seja satisfeita (o valor de  $C$ , neste caso, pode ser igual a 1). Logo  $f$  é  $O(g)$ , i.e.,  $f(n) = 3.\log_{10}(n+1)$  é  $O(3.n/8)$ .



A próxima lista apresenta algumas propriedades:

- $f \in O(f)$
- $O(f) = O(c.f)$ ,  $c > 0$
- se  $f \in O(g)$  e  $g \in O(h)$  então  $f \in O(h)$
- se  $f \in O(g)$  então  $O(f+g) = O(g)$
- se  $f \in O(h_1)$  e  $g \in O(h_2)$  então  $f.g \in O(h_1.h_2)$

O produto por constantes positivas ou a adição de funções com taxas de crescimento inferior não alteram a classe de complexidade. Costuma-se identificar as classes pelas funções de maior crescimento simplificando, assim, a expressão. No exemplo anterior é válido dizer  $3.\log_{10}(n+1)$  é  $O(n)$ .

Outros exemplos:

- 1000000 é  $O(1)$
- $4.m - 1$  é  $O(m)$
- $10/n$  é  $O(1/n)$
- $5.x^2$  é  $O(x^3)$
- $5.x^2$  é  $O(x^2)$
- $5.x^2$  não é  $O(x)$
- $500.n^4 - n^2$  é  $O(n^4)$
- $500.n^4 + n^2$  é  $O(n^4)$
- $n^2.\log_{10}(n)$  é  $O(n^2.\log(n))$
- $4^5.2^n$  é  $O(e^n)$       $e \approx 2.71828\dots$
- $6^n$  não é  $O(5^n)$
- $2^n$  é  $O(n!)$
- $n!$  é  $O(n^n)$

A notação  $O$  representa um limite superior da taxa de crescimento de uma função. Apesar de ser correcto dizer que  $3.\log_{10}(n+1)$  é  $O(n^{10})$  interessa reduzir o limite superior de uma função o quanto for possível, de forma a maximizar o nosso conhecimento sobre os limites de crescimento da função. Melhor que afirmar  $3.\log_{10}(n+1)$  é  $O(n)$  é afirmar  $3.\log_{10}(n+1)$  é  $O(\log(n))$ .

Referimos que as constantes e as funções de menor crescimento podem ser removidas da classe de complexidade. Alguns comentários:

- Para comparar algoritmos da mesma classe de complexidade pode ser útil apresentar um maior detalhe da função de crescimento de modo a compará-los de forma mais precisa. Por exemplo,  $f(n) = n^2 + n.\log(n)$  cresce mais depressa que  $g(n) = n^2 + n$ , apesar de  $f$  e  $g$  serem ambas  $O(n^2)$ .
- A constante do factor de maior crescimento também é útil nessa comparação. Por exemplo, a função  $f(n) = n^2/10$  cresce mais devagar que  $g(n) = 10.n^2$ .

- A base do logaritmo em  $O(\log(n))$  é irrelevante porque basta multiplicar o valor por uma constante para mudar de base:  $\log_a n = \log_a b \cdot \log_b n$ .
- Em casos que na prática acontecem raramente, as constantes das funções podem ser relevantes na comparação de algoritmos de classes distintas. Por exemplo, a função  $f(n) = 2^{n/100000}$  é menor que a função  $g(n) = n^{100000}$  para um extenso conjunto de valores de  $n$ , apesar de  $g$  ser  $O(f)$  e  $f$  não ser  $O(g)$ .

## Classes de Complexidade Comuns

Entre as classes de complexidade existentes, a seguinte lista apresenta as mais comuns:

- Constante,  $O(n^0)$ , ou simplesmente  $O(1)$
- Logarítmica,  $O(\log(n))$
- Polinomial,  $\cup_{p \geq 1} O(n^p)$ 
  - Linear,  $O(n)$
  - Pseudo-linear,  $O(n \cdot \log(n))$
  - Quadrática,  $O(n^2)$
  - Cúbica,  $O(n^3)$
- Exponencial,  $\cup_{p \geq 1} O(p^n)$
- Factorial,  $O(n!)$  ( $n! \approx \sqrt{2\pi n} (n/e)^n$ )

As classes estão apresentadas da menor para a maior classe. Por exemplo, a classe de complexidade exponencial inclui todas as classes anteriores.

É comum considerar uma linha divisória entre as classes contidas na classe polinomial (designada simplesmente por  $P$ ) e as restantes. Qualquer algoritmo de ordem  $P$  é considerado um **algoritmo eficiente** e diz-se possuir um **crescimento polinomial**. Qualquer algoritmo que não seja de ordem  $P$  é considerado um **algoritmo ineficiente** e diz-se possuir um **crescimento superpolinomial** (onde se inclui o crescimento exponencial). As funções superpolinomiais crescem de uma forma extremamente rápida, sendo inimportável o cálculo a partir de uma dada dimensão dos dados iniciais. Diz-se que problemas com esta complexidade são problemas **intratáveis**<sup>28</sup>.

Um exemplo: num dado computador uma operação demora um milissegundo a ser executada. Considere um programa que dado um parâmetro  $n$ , o número de operações realizadas é dado pela função  $f(n)=n$ . Qual o maior valor de  $n$  para o qual o programa é capaz de terminar se o tempo disponível for (i) um minuto, (ii) um dia, (iii) um ano? E para  $f(n)=n^2$ ,  $f(n)=n^3$  e  $f(n)=2^n$ ?

---

<sup>28</sup> Dizer que um algoritmo é intratável, não significa que ele não seja útil para instâncias muito pequenas do problema em questão (principalmente se não for conhecido um algoritmo mais eficiente).

É necessário determinar quantos milissegundos tem cada uma das questões. Um minuto tem 60000 milissegundos, i.e.,  $6 \times 10^4$ . Um dia tem  $8,64 \times 10^7$  milissegundos. Um ano (não bissexto) possui 365 dias, aproximadamente  $3,15 \times 10^{10}$  milissegundos. De seguida, dada a função  $f(n)$  e o número de operações  $op$  resolve-se a equação  $f(n) = op$  em ordem a  $n$ . Por exemplo, para a função quadrática aplica-se a raiz quadrada, para a função exponencial aplica-se o logaritmo da mesma base.

As respostas do problema:

- $f(n)=n$ : (i)  $n = 6 \times 10^5$ , (ii)  $n = 8,64 \times 10^7$  e (iii)  $n = 3,15 \times 10^{10}$
- $f(n)=n^2$ : (i)  $n = 245$ , (ii)  $n = 9295$  e (iii)  $n = 177583$
- $f(n)=n^3$ : (i)  $n = 112$  (ii)  $n = 442$  e (iii)  $n = 3159$
- $f(n)=2^n$ : (i)  $n = 15$ , (ii)  $n = 26$  e (iii)  $n = 34$  (!)

Observando estes valores entende-se mais facilmente a fronteira definida entre as classes polinomial e superpolinomial.

## Outros Limites

A expressão  $f$  é  $O(g)$  indica um limite superior da função  $f$ . Porém, nada é dito sobre o limite inferior.

### Definição:

Considere  $f$  e  $g$  duas funções de domínio inteiro não negativo e contradomínio real. Diz-se que  $f$  é  $\Omega(g)$ , se e só se,

$$\exists_{k>0} \exists_{c>0} \forall_{n>0} : n>k \Rightarrow |f(n)| \geq c|g(n)|$$

Por exemplo:

- 1000000 é  $\Omega(1)$
- $4.m - 1$  é  $\Omega(m)$
- $5.x^2$  é  $\Omega(x)$
- $5.x^2$  é  $\Omega(x^2)$
- $5.x^2$  não é  $\Omega(x^3)$
- $n^2.\log(n)$  é  $\Omega(n^2)$

Se os limites superior e inferior forem iguais utiliza-se a notação seguinte.

### Definição:

Considere  $f$  e  $g$  duas funções de domínio inteiro não negativo e contradomínio real. Diz-se que  $f$  é  $\Theta(g)$ , se e só se,  $f$  é  $O(g)$  e  $f$  é  $\Omega(g)$ .

Por exemplo:

- 1000000 é  $\Theta(1)$
- $4.m - 1$  é  $\Theta(m)$

- $5 \cdot x^2$  não é  $\Theta(x)$
- $5 \cdot x^2$  é  $\Theta(x^2)$
- $5 \cdot x^2$  não é  $\Theta(x^3)$
- $n^2 \cdot \log(n)$  não é  $\Theta(n^2)$

É possível concluir que  $f$  é  $\Theta(g)$  se  $\exists_{c>0} : \lim_{n \rightarrow +\infty} f(n)/g(n) = c$

Por vezes não é fácil calcular a classe de complexidade  $O$  de uma função. Em certas situações é possível calcular uma classe de complexidade alternativa, a classe  $o$  (leia-se ó pequeno):

#### Definição:

Considere  $f$  e  $g$  duas funções de domínio inteiro não negativo e contradomínio real.

Diz-se que  $f$  é  $o(g)$ , se e só se,

$$\lim_{n \rightarrow +\infty} f(n)/g(n) = 0$$

Existe uma propriedade que afirma  $f$  é  $o(g) \Rightarrow f$  é  $O(g)$  estabelecendo-se uma relação entre as duas classes de complexidade. Porém, este cálculo não indica a classe de complexidade mais estrita. Por exemplo, para  $f(n) = 2n^2$  consideramos  $g(n) = n^3$  para calcular que  $f$  é  $o(n^3) \Rightarrow f$  é  $O(n^3)$ . Mas  $O(n^3)$  não representa o menor conjunto que limita superiormente  $f$  dado ser igualmente  $O(n^2)$ .

### 13.3 Análise de Programas

A complexidade temporal de um programa pode ser determinada analisando a sua estrutura. Como uma linguagem de programação possui uma sintaxe e uma semântica bem definidas, a tarefa de analisar a estrutura de um programa dessa linguagem é facilitada. Considera-se o pior caso, i.e., escolhe-se sempre a opção que implique o número máximo de instruções. Procede-se da mesma forma para o cálculo da complexidade espacial.

Existe um conjunto de comandos cuja complexidade temporal é constante, i.e., pertencem à classe  $O(1)$ :

- Declarações
- Atribuições
- Acessos aos métodos
- Uso das operações aritméticas, lógicas e relacionais dos tipos de dados primitivos (dado que a representação dos tipos primitivos possuem dimensão fixa)

Em relação aos restantes comandos, considere a guarda  $G \in O(f_G)$ , os comandos  $A \in O(f_A)$ ,  $B \in O(f_B)$  e  $C \in O(f_C)$  e  $N$  como o número de iterações máximo de cada ciclo:

- $A; B; \quad \in O(f_A + f_B)$
- $\text{if } (G) \text{ } A \text{ else } B; \quad \in O(f_G + f_A + f_B)$

- **while** (G) A;  $\in O(N \cdot (f_G + f_A))$
- **do** A **while** (G);  $\in O(N \cdot (f_G + f_A))$
- **for** (B;G;C) A;  $\in O(f_B + N \cdot (f_G + f_C + f_A))$

Um exemplo: seja T(n) o número de instruções executadas pelo código fonte seguinte:

```
long total = 1;
for(int i=0; i<vector.length; i++)
    total *= vector[i];
```

Seja  $n$  a dimensão do vector. Qual a classe de complexidade temporal O da função T? Existem duas instruções: uma declaração (e atribuição) e um ciclo. A primeira instrução é  $O(1)$  e a segunda é  $O(f_B + N \cdot (f_G + f_C + f_A))$ . A guarda G do ciclo é uma comparação, logo  $f_G$  é  $O(1)$ . As outras componentes do ciclo são uma declaração e uma atribuição, logo  $O(1)$ . O corpo do ciclo é uma atribuição após uma operação aritmética assim,  $f_A$  é  $O(1)$ . Como o ciclo se repete  $n$  vezes, a complexidade do ciclo é  $O(n)$ . A classe de complexidade de T é a soma de  $O(1)$  e  $O(n)$ , i.e., T(n) é  $O(n)$ .

Temos referido sempre a componente temporal. Para encontrar a complexidade espacial de um algoritmo soma-se o maior número de criações dinâmicas de memória (no Java, vectores e objectos) com o número máximo de métodos invocados num dado momento durante a execução do programa. Todas as restantes instruções possuem complexidade espacial constante.

## Algoritmos Recursivos

A análise anterior funciona para programas iterativos onde não existem invocações recursivas. Quando se utiliza recursão, a análise torna-se mais complexa. Considere o seguinte método que calcula o mesmo valor do exemplo anterior:

```
public long produto(int[] vector, int n) {
    if (n == 0)
        return 1;
    return vector[n-1] * produto(vector, n-1);
}
```

Seja T(n) o número de instruções executadas em `produto(vector, vector.length)`. Qual a complexidade temporal de T? Esta função é descrita por:

$$T(n) = \begin{cases} \Theta(1) & , n=0 \\ T(n-1)+\Theta(1) & , n>0 \end{cases}$$

Se o argumento  $n$  for zero, são realizadas duas instruções (o `if` e o `return`). Senão é realizado o `if` (cujo valor da guarda é falso), o `return` e a operação de multiplicação sobre a invocação recursiva do algoritmo com argumento  $n-1$ .

Como resolver esta função recorrente? Não há uma resolução geral para este tipo de problema. Existem métodos para encontrar soluções de certas famílias de funções. Seja o seguinte padrão (os valores  $a, c, m \geq 0$  e  $b > 0$  são constantes) onde se inclui a função anterior:

$$T(n) = \begin{cases} a & , n \leq m \\ b.T(n-1)+c & , n > m \end{cases}$$

Demonstra-se que  $T$  é  $O(n)$  se  $b=1$ , ou  $T$  é  $O(b^n)$  se  $b>1$ . Assim, a função anterior relativa ao factorial é  $O(n)$ .

As constantes possuem os seguintes significados:

- O valor  $a$  representa o tempo gasto para processar o(s) caso(s) base(s) da recursão.
- O valor  $b$  representa o número de subproblemas no qual o problema inicial é decomposto.
- O valor  $c$  representa o tempo gasto na divisão do problema inicial em subproblemas mais o tempo gasto na combinação das soluções desses subproblemas (uma vez calculados).

O número de instruções executadas pelo primeiro método (no capítulo 5) que calcula o  $n$ -ésimo valor da sequência de Fibonacci é limitada superiormente pela seguinte função:

$$T(n) = \begin{cases} \Theta(1) & , n \leq 1 \\ 2.T(n-1)+\Theta(1) & , n > 1 \end{cases}$$

Esta função está incluída na família de funções descrita acima para a qual obtemos uma resposta:  $T$  é  $O(2^n)$ .

O **método mestre** (do inglês, *master method*) calcula a complexidade para a seguinte família de funções (com  $a, m \geq 0$  e  $b > 0$  e  $d > 1$ ):

$$T(n) = \begin{cases} a & , n \leq m \\ b.T(n/d)+f(n) & , n > m \end{cases}$$

Para além do significado anterior dado às constantes  $a$  e  $b$ , o valor  $n/d$  representa a dimensão aproximada dos subproblemas. A função  $f$  representa o tempo gasto na divisão do problema inicial e na posterior recombinação das soluções.

O método mestre diz que a ordem de  $T(n)$  é:

- $\Theta(n^{\log_d b})$  se  $\exists_{\varepsilon > 0} : f(n) \in O(n^{(\log_d b) - \varepsilon})$
- $\Theta(n^{\log_d b} \cdot \log(n))$  se  $f(n) \in \Theta(n^{\log_d b})$
- $\Theta(f(n))$  se  $\exists_{\varepsilon > 0} : f(n) \in \Omega(n^{(\log_d b) + \varepsilon}) \wedge \exists_{\delta < 1} \forall_{n \geq d} : b.f(n/d) \leq \delta.f(n)$

O primeiro caso representa as situações onde  $f(n)$  é limitado superiormente (mas não inferiormente) pela função especial  $n^{\log_d b}$ . O segundo caso inclui as situações em que  $f$  e  $n^{\log_d b}$  são assintoticamente iguais. O terceiro caso trata das funções onde a

complexidade do processo de divisão e recombinação (dado por  $f$ ) é limitado inferiormente (por um factor polinomial) pela função especial  $n^{\log_d b}$ .

Primeiro exemplo:

$$T(n) = \begin{cases} 1 & , n \leq 1 \\ 4.T(n/2) + n & , n > 1 \end{cases}$$

O primeiro passo é calcular a função especial  $n^{\log_d b}$ . Com  $b=4$  e  $d=2$ , obtemos  $n^2$ . Como  $f(n)=n$ , logo  $f$  é  $O(n^{2-\varepsilon})$  para algum  $\varepsilon>0$ . Se escolhermos um valor apropriado, por exemplo  $\varepsilon=0.1$ , obtemos  $n$  é  $O(n^{1.9})$ . Estamos assim no primeiro caso e pelo método mestre  $T$  é  $\Theta(n^2)$ .

Segundo exemplo:

$$T(n) = \begin{cases} 1 & , n \leq 1 \\ T(n/4) + 5 & , n > 1 \end{cases}$$

Com  $b=1$  e  $d=4$ , a função especial  $n^{\log_d b}$  é igual a  $n^{\log_4 1} = n^0 = 1$ . Com  $f(n) = 5$  temos que  $f(n)$  é  $\Theta(1)$ . Estamos no segundo caso do método, logo  $T(n)$  é  $\Theta(\log(n))$ .

Terceiro exemplo:

$$T(n) = \begin{cases} 1 & , n \leq 1 \\ T(n/3) + n & , n > 1 \end{cases}$$

Com  $b=1$  e  $d=3$ , a função especial  $n^{\log_d b}$  é igual a  $n^{\log_3 1} = n^0 = 1$ . Como  $f(n) = n$ ,  $f(n)$  é  $\Omega(n^{0+\varepsilon})$ , por exemplo para  $\varepsilon=1>0$ . A segunda condição do terceiro caso verifica-se:  $f(n/3) = n/3 = (1/3).f(n) \leq \delta.f(n)$ , para  $\delta=1/2<1$ . Sendo esta função parte do terceiro caso do método mestre,  $T(n)$  é  $\Theta(n)$ .

Quarto exemplo:

$$T(n) = \begin{cases} 1 & , n \leq 1 \\ 2.T(n/2) + (n-1) & , n > 1 \end{cases}$$

Com  $b=2$  e  $d=2$ , a função especial  $n^{\log_d b}$  é igual a  $n^{\log_2 2} = n$ . Como  $f(n) = n-1$ , ambas as funções possuem a mesma complexidade. Sendo esta função parte do segundo caso, o método mestre diz que  $T(n)$  é  $\Theta(n \cdot \log(n))$ .

## Dimensão dos dados

É preciso ter atenção com os dados quando se calcula a complexidade. Seja a função  $f(\text{int } N)$ . Como olhar para este  $N$  para encontrar a classe de complexidade a que pertence? Devemos considerar o número de dígitos de  $N$ , o número de bits necessários para o representar ou o próprio valor? Por exemplo, o valor 50000 ocupa cinco dígitos e 16 bits. Será que podemos usar qualquer uma destas três medidas?

Vejamos dois algoritmos:

- Somar os valores  $v[1], v[2], \dots, v[N]$  percorrendo o vector  $v[]$
- Factorizar o número  $N$  calculando  $N/2, N/3, N/4, \dots, N/N-1$

O primeiro algoritmo é linear; dado um vector com  $N$  posições, o algoritmo que calcula este somatório é  $O(n)$ . Mas o segundo algoritmo corresponde a uma tarefa cujos melhores algoritmos conhecidos são exponencialmente complexos. No entanto, dir-se-ia ser igualmente  $O(n)$ . Como é isto possível?

O número de bits é a medida mais usada no cálculo da complexidade. O número de dígitos é-lhe proporcional ( $n^o$  dígitos  $\sim 3.32 n^o$  bits) o que significa que também o poderíamos usar (as constantes são irrelevantes no cálculo da complexidade). Já o valor em si não é proporcional ao número de bits. Por exemplo, com três bits podemos representar números até sete. Já com quatro bits podemos representar até quinze. Com cinco bits, números até 31... O valor  $N$  é proporcional à potência  $2^{\# \text{bits de } N}$ .

Assim, sendo  $n$  o número de bits dos dados de entrada:

- Somar  $v[1], v[2], \dots, v[N] \rightarrow n \sim N \rightarrow$  complexidade linear
- Factorizar com  $n/2, n/3, \dots, n/n-1 \rightarrow n \sim \log N \rightarrow$  complexidade exponencial

Ou seja, quando se calcula a complexidade deve-se considerar sempre o número de bits necessários. Quando nos referimos a dimensões de tipos de dados (como percorrer um vector) é a dimensão dessas estruturas que pretendemos. O mesmo já não se passa quando analisamos valores numéricos onde é preciso distinguir entre o valor em si e o número de bits para o representar (a medida que interessa).

---

## Exercícios

1. Num dado computador, uma operação demora um milisegundo a ser executada. Considere um programa que dado um parâmetro  $n$ , o número de operações realizadas é dado pela função  $f(n)=n^5$ . Qual o maior valor de  $n$  para o qual o programa é capaz de terminar se o tempo disponível for (i) um ano, (ii) uma década, (iii) mil milhões de séculos? Calcule também para  $f(n)=\log_{10}n$ ,  $f(n)=n/10$ ,  $f(n)=n/1000$ ,  $f(n)=n^{10}$  e  $f(n)=10^n$ .

2. Mostre que:

- 2004 é  $O(1)$
- $n^3.(5n+n^3) \in O(n^6)$
- $(n+3)^3 \in O(n^3)$
- $\log \log(n) \in O(\log n)$
- $2^{n+a} \in O(2^n)$
- $2^{an} \notin O(2^n), a>1$

g)  $n^{1.01} + n \cdot \log(n) \in O(n^{1.01})$

h)  $2^n \in O(n!)$

i)  $n! \in O(n^n)$

j)  $\sum_{k=0}^n k^2 \in O(n^3)$

3. Considere  $f(n) = 2^n$  e  $g(n) = n^n$ . Será que  $f \in O(g)$  ou  $g \in O(f)$ ? Justifique?

4. Considere  $f(n)$  igual ao  $n$ -ésimo número de *Fibonacci*. Encontre uma função  $g$  (diferente de  $f$ ) tal que  $f \in O(g)$ .

5. Encontre duas funções  $f_1$  e  $f_2$  que sejam  $O(g)$  mas que  $f_1 \notin O(f_2)$ .

6. Assumindo que  $f_1 \in O(g_1)$  e  $f_2 \in O(g_2)$  prove as seguintes expressões:

a)  $f_1(n)+f_2(n) \in O(\max(g_1(n), g_2(n)))$

b)  $f_1(n) \times f_2(n) \in O(g_1(n) \times g_2(n))$

c) se  $f_1 \in O(g_2)$  então  $O(f_1+f_2) = O(g_2)$

d) seja  $g \in O(h)$ . Nem sempre  $f(g(n)) \in O(f(h(n)))$

7. Considere  $T(n)$  o número de instruções executadas pelos exemplos de cada alínea. Para cada caso, qual a classe de complexidade que limita superiormente  $T$ ?

- |                                |                                   |
|--------------------------------|-----------------------------------|
| a) int m=0;                    | e) int x=n*n*n;                   |
| <b>for</b> (i=1; i<10*n; i++)  | <b>for</b> (; x>1;)               |
| <b>for</b> (j=1; j<n; j++)     | x /= 2;                           |
| m++;                           |                                   |
| b) int a=0, soma=0;            | f) int soma=n*n;                  |
| <b>do</b> {                    | <b>while</b> (soma%2==0)          |
| soma += (++a)*n;               | soma--;                           |
| } <b>while</b> (a<1000);       |                                   |
| c) int b=n*n;                  | g) <b>for</b> (int i=0; i<n; i++) |
| <b>while</b> (b>n)             | <b>for</b> (int j=0; j<n; j++) {  |
| <b>if</b> (b%2==0)             | a[i][j] = 0;                      |
| b--;                           | <b>for</b> (int k=0; k<n; k++)    |
| <b>else</b>                    | a[i][j] += b[i][k] *              |
| b-=2;                          | c[k][j];                          |
| d) int soma=0;                 | }                                 |
| <b>for</b> (int i=0; i<n; i++) | h) int c=0;                       |
| <b>for</b> (int j=i; j<n; j++) | <b>for</b> (int i=0; i<n; i++)    |
| soma += v[i];                  | <b>for</b> (int j=i; j<n; j++)    |
|                                | c++;                              |

8. Calcule pelo método mestre, a complexidade das seguintes funções:

$$a) T(n) = \begin{cases} 1000 & , n \leq 1 \\ 6.T(n/3)+50 & , n > 1 \end{cases}$$

$$b) T(n) = \begin{cases} 1 & , n \leq 1 \\ 3.T(n/8)+n & , n > 1 \end{cases}$$

$$c) T(n) = \begin{cases} 10 & , n \leq 1 \\ 4.T(n/2)+n^2 & , n > 1 \end{cases}$$

9. Procure na internet informação sobre a função de Ackermann e tente perceber como se processa o seu grande crescimento.

# 14 – Listas

---

No capítulo 1 definimos o tipo de dados lista de forma recursiva: uma lista ou é vazia ou é constituída por uma célula de informação e uma lista. Uma lista é uma sequência de elementos do mesmo tipo onde a ordem de inserção é relevante. Existe um primeiro elemento (denominado por cabeça), seguido por um segundo elemento e assim sucessivamente. Comecemos por especificar o tipo de dados abstracto `List`<sup>29</sup>. Uma vez especificado o tipo, veremos a interface Java que associa os contratos adequados. Por fim, é apresentada uma implementação da interface.

## 14.1 Especificação da Lista

A representação de uma lista é definida a partir de duas operações: `empty` que representa a lista vazia e `cons` que representa uma lista constituída por um elemento (de um qualquer tipo designado por `Element`) e uma outra lista. Estas duas operações são os construtores do tipo.

As outras operações definidas sobre este TDA são:

- `isEmpty` – verifica se a lista está vazia
- `head` – devolve o primeiro elemento da lista
- `tail` – devolve a lista sem o primeiro elemento

---

<sup>29</sup> Tanto nas especificações como no código Java, as palavras usadas para descrever tipos, variáveis, classes e métodos serão palavras inglesas. Os comentários serão em português.

- `length` – devolve o número de elementos de uma lista
- `equals` – verifica se duas listas são iguais
- `addBegin` – insere um novo elemento no início da lista
- `addEnd` – insere um novo elemento no fim da lista
- `removeBegin` – remove o primeiro elemento do início
- `removeEnd` – remove o último elemento
- `get` – devolve o  $n$ -ésimo elemento
- `add` – insere um elemento na  $n$ -ésima posição
- `contains` – verifica se o elemento existe na lista
- `indexOf` – devolve a primeira posição do elemento na lista (se não encontrar, devolve zero).
- `concat` – concatena os elementos de duas listas.
- `remFirst` – remove a primeira ocorrência do elemento
- `remAll` – remove todas as ocorrências do elemento
- `reverse` – inverte a informação da lista (o primeiro elemento passa para último, o segundo para penúltimo...)

Segue o Tipo de Dados Abstracto (os TDAs são introduzidos no capítulo 8):

```
especificação List<Element> =
importa Integer, Boolean
géneros List
operações

    empty : → List
    cons : List Element → List
    isEmpty : List → Boolean
    head : List ↗ Element
    tail : List ↗ List
    length : List → Integer
    equals : List List → Boolean
    addBegin : List Element → List
    addEnd : List Element → List
    removeBegin : List ↗ List
    removeEnd : List ↗ List
    get : List int ↗ Element
    add : List Element int ↗ List
    contains : List Element → Boolean
    indexOf : List Element → Integer
```

```
concat : List List → List
remFirst : List Element → List
remAll : List Element → List
reverse : List → List

axiomas
isEmpty(empty)      = TRUE
isEmpty(cons(L,I)) = FALSE
head(cons(L,I))   = I
tail(cons(L,I))   = L
```

Definimos cada operação para o seu domínio. Sendo `head` e `tail` parciais (a lista vazia não pertence ao domínio) não se definem esses casos. Observamos nestes axiomas como é representada uma lista não vazia: `cons(L,I)` que representa uma lista tendo o elemento `I` à cabeça e sendo `L` o resto da lista.

```
length(L) =
  if isEmpty(L)
  then 0
  else 1 + length(tail(L))
```

A dimensão de uma lista é definida recursivamente. Uma lista vazia tem dimensão zero (a base da recursão), uma lista não vazia é igual a um mais a dimensão do resto da lista (o passo de recursão).

```
equals(L1,L2) =
  if isEmpty(L1)
  then isEmpty(L2)
  else not isEmpty(L2) and
        equals(head(L1),head(L2)) and
        equals(tail(L1),tail(L2))
```

Este axioma define a igualdade entre listas. Duas listas são iguais se ambas são vazias ou ambas possuem a mesma dimensão e em cada posição das duas listas existir um elemento idêntico. A operação `equals` na expressão `equals(head(L1), head(L2))` representa uma operação relativa ao tipo `Element` e não ao tipo `List`.

```
addBegin(L,I) = cons(L,I)
```

O comportamento da operação `addBegin` é idêntica ao `cons`.

```
addEnd(L,I) =
  if isEmpty(L)
  then cons(empty,I)
  else cons(addEnd(tail(L),I),head(L))
```

Para obter a representação do último elemento da lista é necessário consultar recursivamente o resto da lista até chegar à lista vazia, a base da recursão (devolve-se uma lista unitária com o elemento pretendido). O passo da recursão indica que inserir um elemento no fim de uma lista não vazia é o mesmo que inserir esse elemento no fim do resto da lista – `addEnd(tail(L), I)` – não esquecendo de colocar a cabeça da lista original no resultado.

```
removeBegin(L) = tail(L)
removeEnd(L) =
  if isEmpty(tail(L))
    then empty
    else cons(removeEnd(tail(L)), head(L))
```

O raciocínio para definir as propriedades das operações `removeBegin` e `removeEnd` foi semelhante ao utilizado para definir as inserções. O comportamento da remoção no início tem de verificar a mesma propriedade da operação que devolve o resto da lista. Em relação à remoção no fim, é necessário consultar recursivamente o resto da lista até chegar à lista com um elemento (a base da recursão). Quando o respectivo subproblema é resolvido, volta-se a inserir a cabeça da lista em questão.

```
get(L, N) =
  if N=1
    then head(L)
    else get(tail(L), N-1)
```

Se o  $N$  é igual a 1 a resposta é imediata: devolve-se a cabeça da lista (a base da recursão). Caso contrário, devolver o  $n$ -ésimo elemento de uma lista é devolver o  $n$ -ésimo menos um do resto da lista.

```
add(L, I, N) =
  if N=1
    then cons(L, I)
    else cons(add(tail(L), I, N-1), head(L))
```

O caso base da operação `add`: inserir na primeira posição; o passo da recursão: inserir um elemento na posição  $n$  da lista é igual a inserir esse elemento na posição  $n-1$  no resto da lista (sem esquecer de inserir a cabeça da lista original nesse resultado).

```
contains(L, I) = not isEmpty(L) and
  (equals(head(L), I) or contains(tail(L), I))
```

Uma lista contém um elemento se este estiver à cabeça (a base da recursão) ou se estiver no resto da lista (o passo da recursão).

```
indexOf(L, I) =
  if not contains(L, I)
    then 0
```

```
else if equals(head(L), I)
    then 1
    else 1 + indexOf(tail(L), I)

remFirst(L, I) =
    if isEmpty(L)
        then empty
    else if equals(head(L), I)
        then tail(L)
        else cons(remFirst(tail(L), I), head(L))

remAll(L, I) =
    if isEmpty(L)
        then empty
    else if equals(head(L), I)
        then remAll(tail(L), I)
        else cons(remAll(tail(L), I), head(L))
```

A diferença entre estes dois últimos axiomas ocorre quando o elemento é encontrado: a operação `remFirst` termina a recursão enquanto `remAll` continua até ao caso base (ou seja, à lista vazia).

```
concat(L1, L2) =
    if isEmpty(L1)
        then L2
    else cons(concat(tail(L1), L2), head(L1))
```

Concatenar duas listas é o mesmo que concatenar o resto da primeira lista com a segunda lista para, no fim, adicionar a cabeça da primeira lista.

```
reverse(L) =
    if isEmpty(L)
        then empty
    else addEnd(reverse(tail(L)), head(L))
```

Para inverter uma lista recursivamente, inverte-se o resto da lista e adiciona-se no fim a cabeça da lista original.

#### **pré-condições**

```
removeBegin(L) requer not isEmpty(L)
removeEnd(L) requer not isEmpty(L)
head(L) requer not isEmpty(L)
tail(L) requer not isEmpty(L)
get(L, N) requer N <= length(L)
        and N > 0
```

```
add(L, I, N) requer N <= length(L)+1  
           and N > 0  
fim-especificação
```

As operações `get` e `add` exigem como pré-condições que o índice do elemento seja válido, não garantindo qualquer resultado se tal não ocorrer. As restantes operações parciais requerem que a lista dada não seja vazia.

## 14.2 O Desenho e os Contratos do TDA Lista

Com a especificação da secção anterior desenhemos a interface Java para descrever o que uma implementação deste TDA deve respeitar.

```
package dataStructures;
```

Esta interface e todo código fonte implementado nesta 3<sup>a</sup> parte constituem um pacote denominado `dataStructures` onde serão disponibilizadas as estruturas de dados e algoritmos apresentados ao longo dos restantes capítulos.

```
/**  
 * <p>Titolo: A interface do TDA Lista</p>  
 * <p>Descrição: O desenho da Lista + contratos</p>  
 */
```

Cada método (e a própria interface) será antecedido por comentários de documentação (ver anexo A para maiores detalhes) onde é explicado cada um dos componentes do método (nomeadamente, os parâmetros e o valor devolvido).

```
public interface List {  
    //@ public invariant !isEmpty() ==> head() != null;
```

Definiu-se uma invariante a ser satisfeita por qualquer objecto deste tipo: não é possível armazenar referências nulas.

```
/**  
 * Criar uma lista vazia  
 * @return uma lista vazia  
 */  
//@ ensures \result.isEmpty();  
/*@ pure @*/ List empty();
```

O primeiro método refere-se ao construtor lista vazia da especificação. O contrato associado refere qual o compromisso: uma vez terminado o método, é devolvida uma estrutura vazia. Existe, assim, na própria interface uma semântica associada ao método. Aqui reside uma opção no desenho, o método `empty()` devolve um novo objecto vazio em vez de reiniciar o eventual objecto de onde o método é invocado (por isso é considerado uma interrogação e não um comando). A razão desta escolha é a existência

obrigatória de um construtor explícito em cada classe Java. Se `empty()` fosse um comando, teria o mesmo comportamento do construtor da classe.

```
/**  
 * Adicionar à lista um novo objecto  
 * @param o A referência do objecto a adicionar  
 */  
//@ requires o != null;  
//@ ensures head().equals(o);  
//@ ensures tail().equals(\old(clone()));  
void cons(Object o);
```

Este método refere-se ao segundo construtor, `cons()`. Encontramos um requisito inicial: a referência do objecto a ser inserido na lista deve ser diferente de `null`. Esta pré-condição garante que a lista não armazenará referências nulas, satisfazendo a invariante da interface. Qual a necessidade de evitar referências nulas? A primeira razão encontra-se na pós-condição seguinte, ao permitir a inserção de uma referência nula à cabeça da lista, a execução de `head().equals(o)` produziria uma excepção do tipo `NullPointerException`. Existe porém outra razão mais fundamental. O TDA asserta que o domínio da operação `head` é um valor do tipo `Element` que parametriza `List`. Uma referência nula em Java não refere qualquer objecto de qualquer tipo, violando por isso essa asserção.

A primeira pós-condição garante que quando terminar a execução, a lista resultante tem o novo elemento à cabeça. A segunda pós-condição informa que o resto da lista é igual à lista inicial antes de executar o método. O facto de inserir à cabeça está definido nas propriedades especificadas no TDA. As interrogações `head()` e `tail()` serão definidas a seguir.

A segunda pós-condição só foi possível construindo uma cópia do conjunto para armazenar o primeiro elemento antes do método se iniciar. Porquê? Porque no JML é realizada uma cópia do valor ou da variável argumento do `\old()`. Se em vez de `\old(clone())` estivesse `\old(this)` apenas se criaria uma cópia da referência para o objecto. Assim, mesmo depois da execução do método, `\old(this)` refere-se ao objecto transformado e não ao original (foi criado uma cópia da referência para o objecto, não uma cópia do objecto). Para que a pós-condição funcione correctamente é necessário construir uma cópia explícita do objecto.

De notar que o método `equals()` refere-se à classe do objecto `o` e não ao `equals()` da lista (ainda por definir). Como é reconhecido o método apropriado? Pela capacidade de ligação dinâmica (descrita no capítulo 10). Como se garante que a classe do objecto inserido possui esse método? Porque `equals()` é um método da classe `Object` de onde são derivadas todas as classes. Se a classe do objecto inserido não tiver definido `equals()` será usado o método com o mesmo nome da classe `Object`.

Uma outra questão: não definimos no TDA que o tipo genérico dos dados era *Element*? Aqui está descrito como *Object*?! O desenho é um refinamento da especificação, um conjunto de compromissos. Um desses compromissos reside na escolha da classe *Object* como parâmetro para definir o tipo das referências dos objectos armazenados. Queremos que a lista seja genérica, i.e., que possa armazenar qualquer tipo de dados. Queremos também que não prejudique a definição da lista, dado que o funcionamento interno é independente da informação guardada. Se definirmos que as referências para os elementos são do tipo *Object* ficamos com a garantia que qualquer objecto de qualquer classe é aceite. Porquê? Porque se todas as classes Java, por definição, derivam de *Object*. Qualquer objecto de qualquer classe é igualmente um objecto da classe *Object*. Se for necessário aceder aos componentes específicos da classe do objecto, faz-se uma coerção de tipos.

```
/**  
 * A lista está vazia?  
 * @return TRUE se a lista está vazia, FALSE c.c.  
 */  
/*@ pure */ boolean isEmpty();
```

Esta é a interrogação básica que informa se a lista está vazia ou não. Todas as interrogações são precedidas por `/*@ pure */` para informar ao JML que não provocam efeitos secundários, podendo assim ser utilizadas nas asserções.

```
/**  
 * Devolver o elemento do início da lista  
 * @return A referência do primeiro elemento  
 */  
//@ requires !isEmpty();  
/*@ pure */ Object head();
```

O método `head()` refere-se à segunda interrogação da especificação, devolvendo a referência do objecto que está à cabeça da lista. A pré-condição afirma que o método só funcionará adequadamente se a lista não estiver vazia no momento da invocação.

```
/**  
 * Devolver o resto da lista  
 * @return A referência para o resto da lista  
 */  
//@ requires !isEmpty();  
/*@ pure */ List tail();
```

A pré-condição do método `tail()` reflecte a pré-condição da especificação. No método `tail()` não se utilizou a regra que transforma as operações do TDA que devolvem o tipo em questão em métodos que alteram o objecto (devolvendo `void`). Neste caso, optámos por definir o método como uma interrogação. A razão deveu-se ao facto de facilitar a

escrita dos contratos seguintes, já que as operações `head` e `tail` foram as preferencialmente usadas nos restantes axiomas.

Neste momento, não há qualquer indicação se a lista resultado deve ser uma cópia da lista original ou se é apenas devolvida a referência do segundo elemento. Essa decisão será tomada durante a fase de implementação. Para forçar uma das opções, poder-se-ia incluir uma pós-condição que verificasse `\result.get(i-1) == get(i)` (comparam-se referências não conteúdos) de modo que a lista não fosse duplicada ou negar a igualdade para forçar a duplicação.

```
/**  
 * Calcular o número de elementos da lista  
 * @return O número de elementos  
 */  
  
//@ ensures isEmpty() ==> \result == 0;  
//@ ensures !isEmpty() ==> \result == 1 + tail().length();  
/*@ pure @*/ int length();
```

A interrogação que devolve a dimensão da lista. As asserções reflectem o respectivo axioma: zero se a lista é vazia, caso contrário é um mais a dimensão do resto da lista.

```
/**  
 * Devolver o n-ésimo elemento da lista  
 * @param n A posição requerida  
 * @return A referência para o n-ésimo elemento  
 */  
  
//@ requires n > 0 && n <= length();  
//@ ensures n==1 ==> \result.equals(head());  
//@ ensures n!=1 ==> \result.equals(tail().get(n-1));  
/*@ pure @*/ Object get(int n);
```

A interrogação `get()` requer que o parâmetro seja um valor válido, i.e., que o valor de `n` seja positivo e que não guarde um valor maior que a dimensão total da lista – uma das pré-condições da especificação – sendo, por isso, uma pré-condição obrigatória. Se fosse incluída a pós-condição `equals(\old(clone()))` para garantir que a lista não seja modificada durante a execução do método, provocaria um ciclo infinito de invocações (`equals()` invocaria `get()` que invocaria `equals()`...).

```
/**  
 * Inserir no início  
 * @param o A referência do elemento a ser inserido  
 */  
  
//@ requires o != null;  
//@ ensures head().equals(o);  
//@ ensures tail().equals(\old(clone()));  
void addBegin(Object o);
```

O método `addBegin()` insere um elemento no início da lista (o mesmo comportamento do método `cons()` tem como consequência os mesmos contratos).

```
/**
 * Inserir no fim
 * @param o A referência do elemento a ser inserido
 */
/*@ requires o != null;
 @ ensures length() == \old(length()) + 1;
 @ ensures get(length()).equals(o);
 @ ensures (\forall int i; 1 <= i && i <= length()-1;
 @         ((List)\old(clone())).get(i).equals(get(i)));
 @*/
void addEnd(Object o);
```

Como não foi possível exprimir directamente o respectivo axioma dado este utilizar o comando `cons()` o que produziria um contrato com efeitos secundários, encontrou-se uma expressão alternativa das propriedades através das interrogações `get()` e `length()` (repetiremos este processo para outros métodos mais adiante).

Para verificar se o elemento foi introduzido no fim, a primeira pós-condição verifica `get(length()).equals(o)`. A segunda pós-condição verifica se a dimensão da lista, uma vez inserido o elemento, aumenta uma unidade (dado haver mais um elemento armazenado). A terceira pós-condição percorre todos os restantes elementos da lista (usando o quantificador universal `\forall` desde a posição 1 até à posição do penúltimo elemento) verificando a seguinte propriedade: os elementos originais mantêm as suas posições iniciais. Fez-se uma cópia da lista inicial, `\old(clone())`, para comparar com a lista final.

```
/**
 * Inserir na n-ésima posição
 * @param o A referência do elemento a ser inserido
 * @param n A posição de inserção (n=1 insere no inicio)
 */
/*@ requires o != null;
 @ requires n > 0 && n <= length()+1;
 @ ensures length() == \old(length()) + 1;
 @ ensures get(n).equals(o);
 @ ensures (\forall int i; n <= i && i <= \old(length()));
 @         ((List)\old(clone())).get(i).equals(get(i+1));
 @ ensures (\forall int j; 1 <= j && j <= (n-1);
 @         ((List)\old(clone())).get(j).equals(get(j)));
 @*/
void add(Object o, int n);
```

O método `add()` insere um elemento em qualquer posição. Este método é mais geral que os dois anteriores – `addBegin(o)` é igual a `add(o, 1)` e `addEnd(o)` é igual a `add(o, length())`. Existe uma pré-condição que verifica se o valor de  $n$  é válido. As duas últimas pós-condições comparam a estrutura interna da lista antes e depois da inserção. A lista é igual até ao índice anterior à posição onde o novo elemento foi inserido. Após o índice, as posições dos elementos avançaram uma unidade.

```
/**
 * Remover o primeiro elemento
 */
/*@ requires !isEmpty();
/*@ ensures equals((List)\old(clone()).tail());
void removeBegin();
```

A remoção do elemento inicial possui uma restrição: a lista não pode estar vazia. A pós-condição reflecte o axioma da especificação: a lista final tem de ser igual à cauda da lista inicial.

```
/**
 * Remover o último elemento
 */
/*@ requires !isEmpty();
@ ensures length() == \old(length()) - 1;
@ ensures (\forall int i; 1 <= i && i <= length()-1;
@           ((List)\old(clone())).get(i).equals(get(i)));
@*/
void removeEnd();
```

O mesmo raciocínio é aplicado ao método `removeBegin()` que remove do fim da lista.

```
/**
 * o objecto está na lista?
 * @param o A referência do nó a ser procurado
 * @return TRUE se existe, FALSE c.c.
 */
/*@ ensures \result == (\exists int i; 1<=i && i<=length();
@                   get(i).equals(o));
@*/
/*@ pure @*/ boolean contains(Object o);
```

Pela primeira vez é usado um quantificador existencial: a interrogação `contains(o)` será verdade se existir pelo menos um objecto na lista igual ao objecto  $o$ . Outra forma de expressar a mesma pós-condição, mais próxima do TDA é a seguinte:

```
/*@ ensures \result == (!isEmpty() &&
@                   (o.equals(head()) || tail().contains(o)));
@*/
```

Sempre que possível deve escolher uma asserção que reflecta o TDA que a interface refina. Segundo este critério, a segunda opção é preferível.

```
/**  
 * Devolver a posição da primeira referência do objecto na  
 lista. Se o objecto não existir, devolver zero.  
 * @param o A referência do objecto a procurar  
 * @return A posição do 1º objecto igual a 'o', senão zero  
 */  
  
/*@ requires o != null;  
 @ ensures !contains(o) ==> \result == 0;  
 @ ensures contains(o) && o.equals(head()) ==> \result==1;  
 @ ensures contains(o) && !(o.equals(head())) ==>  
 @ \result == 1 + tail().indexOf(o);  
 @*/  
/*@ pure @*/ int indexOf(Object o);
```

As pós-condições reflectem os dois condicionais do axioma que define o comportamento da operação `indexOf`.

```
/**  
 * Remover a 1ª ocorrência de um elemento  
 * @param o O objecto a remover  
 */  
  
//@ requires o != null;  
//@ ensures \old(contains(o))==>length()==\old(length())-1;  
void remFirst(Object o);  
  
/**  
 * Remover todas as ocorrências de um elemento  
 * @param o O objecto a remover  
 */  
  
//@ requires o != null;  
//@ ensures !contains(o);  
void remAll(Object o);
```

Depois de remover todas as ocorrências do elemento, este deixou de existir na lista. Não se pode calcular quantos elementos foram removidos porque não existem interrogações apropriadas para formular essa asserção.

```
/**  
 * São as duas listas iguais?  
 * @param l A lista a ser comparada  
 * @return TRUE se são iguais, FALSE c.c  
 */  
  
//@ also  
/*@ ensures isEmpty() ==> \result == l.isEmpty();
```

```

@ ensures !isEmpty() ==> \result == ( !(l.isEmpty())
@                               && head().equals(l.head())
@                               && tail().equals(l.tail()) );
@*/
/*@ pure @*/ boolean equals(Object l);

```

A interrogação `equals()` deve comparar o próprio objecto com uma outra lista. As duas pós-condições reflectem a expressão lógica do respetivo axioma. No entanto, pretende-se rescrever o método da superclasse `Object`, daí a assinatura receber uma referência desse tipo. A primeira linha do contrato (`//@ also`) informa que as asserções que se seguem são adicionadas às eventuais asserções que o método `equals()` possa já ter das superclasses (neste caso, nenhuma pois não alterámos a classe `Object`, mas o JML exige-o).

```

/**
 * Concatenar duas listas
 * @param l A lista a ser concatenada no fim desta lista
 */
/*@ requires l != null;
@ ensures length() == \old(length()) + l.length();
@ ensures (\forall int i; 1 <= i && i <= \old(length()));
@       ((List)\old(clone())).get(i).equals(get(i));
@ ensures (\forall int j; \old(length()) < j && j <= length());
@       get(j).equals(l.get(j - (\old(length()))));
@*/
void concat(List l);

```

A concatenação de duas listas. As pós-condições verificam se a lista que entra como argumento é inserida no final do objecto, para satisfazer a especificação. A segunda pós-condição verifica se a lista original mantém as suas posições. A terceira pós-condição verifica que a lista `l` foi correctamente adicionada no fim.

```

/**
 * Inverter a lista
*/
/*@ ensures length() == \old(length());
@ ensures (\forall int i; 1 <= i && i <= length());
@   ((List)\old(clone())).get(i).equals(get(length() - i + 1));
@*/
void reverse();

```

A inversão da lista. A primeira pós-condição verifica se a dimensão da lista não se altera. A segunda pós-condição verifica se os elementos ficaram correctamente invertidos. Para efectuar a comparação, criou-se uma cópia da lista inicial.

```

/**
 * Devolver uma cópia da estrutura.

```

```

    * @return devolve uma referência para a cópia
    */
    // @ also
    // @ ensures (\result != this) && equals(\result);
    /*@ pure @*/ Object clone();
}

```

Finalmente, o método para clonar a lista. Mais uma vez, a primeira linha do contrato (`// @ also`) informa que as asserções que se seguem são adicionadas às eventuais asserções que o método `clone()` possa já ter (o método `clone` é definido na classe `Object` da qual esta classe deriva). A pós-condição confirma que o objecto produzido é igual ao original, `equals(\result)` mas não é o mesmo, `\result != this`.

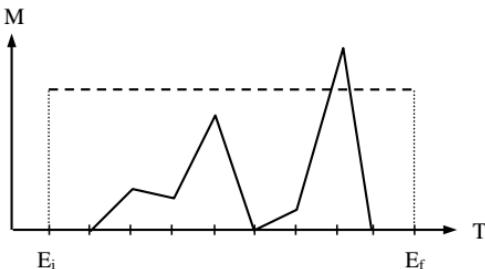
O método `clone()` não se encontra na especificação. Não é incorrecto adicionar funcionalidades dado que a interface é um refinamento da especificação. O contrário seria incorrecto: a interface não referenciar funções especificadas no TDA. A interface refina, não generaliza.

```
} //endInterface List
```

## 14.3 Uma Implementação Dinâmica da Lista

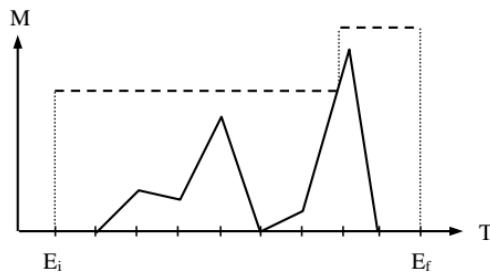
Uma vez definida a interface é chegado o próximo passo: o desenvolvimento de uma implementação. Em primeiro lugar é necessário optar por uma representação de memória adequada para representar o estado dos objectos da classe lista. A partir daí, os algoritmos que implementam os métodos reflectem a representação escolhida. Uma boa escolha na representação implica algoritmos mais simples, uma má escolha na representação pode mesmo comprometer o desenvolvimento da classe em questão.

Nesta secção optámos por uma **representação dinâmica** da informação: a memória usada pelo objecto, consoante as necessidades, cresce e diminui em tempo de execução. Ao contrário de uma **representação estática** – como no uso de um vector onde a dimensão máxima é determinada no momento da construção do objecto – uma estrutura de dados dinâmica varia de dimensão com a execução do programa. Graficamente podemos observar que a memória gasta (eixo M) por uma representação estática (linha a tracejado), desde o momento de criação do objecto ( $E_i$ ) até ao momento de remoção ( $E_f$ ) é sempre constante.



Uma representação dinâmica é mais versátil ocupando só a memória necessária ao estado do programa. O limite para armazenar informação é determinado pelos recursos do computador e não por um limite artificialmente imposto como na representação estática. No gráfico acima, houve uma situação onde a memória gasta pela representação dinâmica ultrapassou a capacidade de representação estática. Se, nesse caso, tivesse sido usada a representação estática teria ocorrido um problema de memória (em inglês, *memory overflow*).

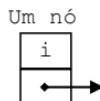
Porém, quando conhecemos um limite máximo da informação a armazenar, uma representação estática pode ser mais eficiente. O acesso e alteração de elementos num vector é mais rápido que numa estrutura dinâmica. Para além disso é sempre possível, quando se detecta este problema, criar um novo vector com maior capacidade e actualizá-lo com os elementos armazenados.



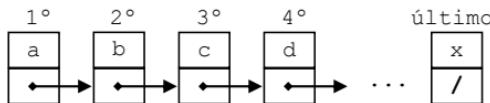
Veremos ao longo desta 3<sup>a</sup> parte diferentes exemplos que favorecem uma das abordagens em detrimento da outra.

## A Classe Nó

Voltando às listas, começamos por definir um novo tipo designado por nó. Um objecto nó armazena duas informações: (i) um objecto de um tipo qualquer e (ii) uma referência nula ou uma referência para um outro objecto nó. A parte (ii) mostra que este novo tipo é recursivo (a base da recursão é a referência nula). Graficamente, um nó pode ser descrito por:



A utilidade de um nó é armazenar um elemento da lista. O próximo elemento da lista encontra-se num outro nó referenciado por este e assim sucessivamente:

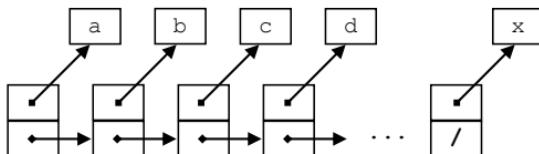


Observemos primeiro o código fonte associado aos nós:

```
package dataStructures; // ficheiro DList.java
/**
 * <p>Título: A classe Nó</p>
 * <p>Descrição: Usado para armazenar informação em estruturas
 * de dados dinâmicas</p>
 */
class Node implements Cloneable {
    public Object item;
    public Node next;
```

O atributo `item` guarda a referência para o objecto armazenado e `next` guarda a referência para o próximo nó. Neste caso, os atributos da classe `Node` não são privados dado que esta classe só é utilizada na classe lista (sendo incluída no ficheiro da classe pública `DList`) e deste modo simplificamos a codificação de vários métodos.

A configuração de um objecto nó é constituída por duas referências. O diagrama de uma sequência de nós poderia também ser descrito assim:



Preferimos a representação do diagrama anterior por motivos de clareza mas é importante não ficar uma impressão errada do uso das referências.

```
/**
 * @param o O objecto a armazenar
 * @param n A referência para o próximo nó
 */
```

```

//@ requires o != null;
//@ ensures item.equals(o);
//@ ensures n!=null ==> next.equals(n);
public Node(Object o, Node n) {
    item = o;
    next = n;
}

```

O construtor recebe dois parâmetros para inicializar os valores dos atributos que definem o estado do novo objecto. Mesmo sem existir uma interface para esta classe é possível associar contratos à mesma.

```

/***
 * @param n O nó a ser comparado
 * @return TRUE se ambos os nós referenciarem o mesmo
         objecto e o mesmo nó seguinte, senão FALSE
 */
/*@ pure */ public boolean equals(Node n) {
    if (n==null)
        return false;
    if (next==null)
        return n.next==null;
    return item.equals(n.item) && next.equals(n.next);
}

```

Considerámos que um nó é igual a outro se a informação armazenada for igual (ou seja, a execução da interrogação `equals()` entre os dois objectos armazenados for verdadeira) e os próximos nós forem iguais.

```

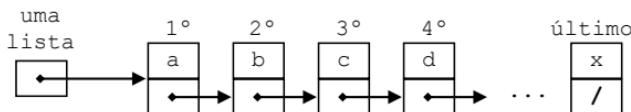
//@ also
//@ ensures (\result != this) && equals((Node)\result);
/*@ pure */ public Object clone() {
    Node copy = new Node(item,next);
    if (next!=null)          // clonagem recursiva...
        copy.next = (Node)next.clone();
    return copy;
}
} //endClass Node

```

A clonagem do nó é recursiva. Primeiro invoca o construtor da classe para criar um novo objecto. A partir daí, se existir um próximo nó (o valor de `next` é diferente de `null`), é executada a clonagem do nó seguinte e o endereço produzido é armazenado no atributo `next`. Essa segunda clonagem eventualmente executará a clonagem do terceiro nó, o terceiro do quarto, etc.

## A Implementação da Lista

Estamos em condições de implementar dinamicamente o tipo lista. Neste contexto, como é constituída a configuração de um objecto deste tipo? Nada mais que uma referência para um único nó. Qual? Para o nó que armazena o primeiro elemento da lista.



Olhando a figura é fácil observar que os elementos estão dispostos linearmente e que a dimensão da estrutura não é fixa, mas varia com o número de elementos armazenados. De notar que esta estrutura, designada por **lista ligada** (do inglês, *linked list*), é uma implementação possível do TDA lista.

```
// ... continuação do ficheiro DList.java
/**
 * <p>Título: A classe Lista</p>
 * <p>Descrição: Usado para armazenar informação numa
 * estrutura de nós cuja dimensão é dinâmica</p>
 */
public class DList implements List, Cloneable {
```

Esta classe é uma implementação da interface `List`. Consideraremos igualmente que é uma implementação da interface `Cloneable`, a classe indica a capacidade de ser clonável. O nome escolhido – `DList` – respeita uma convenção seguida nesta parte: o prefixo indica o tipo de representação de memória usado (D para representação dinâmica, V para representação vectorial).

```
private Node theHead;
```

O único atributo da classe como referido no início da secção.

```
public DList() {
    theHead = null;
}

public List empty() {
    return new DList()
}

public void cons(Object o) {
    addBegin(o);
}
```

O primeiro método é o construtor da classe. O construtor da especificação `empty()` devolve um novo objecto invocando o construtor da classe.

```
public boolean isEmpty() {  
    return theHead == null;  
}
```

Para verificar se a lista é vazia, é suficiente determinar se a cabeça da lista referencia algum nó.

```
public Object head() {  
    return theHead.item;  
}
```

O método `head()` devolve o objecto à cabeça como especificado.

```
public int length() {  
    int i = 0;  
    for(Node aux=theHead; aux!=null; aux=aux.next)  
        i++;  
    return i;  
}
```

Para determinar a dimensão da lista criamos uma referência auxiliar `aux` iniciada na cabeça da lista. Enquanto `aux` não for nula actualizamos o contador `i` e avançamos para o próximo nó. Como? Armazenando em `aux` a referência do próximo nó. Qual é a complexidade temporal do método `length()`? A complexidade do algoritmo é  $O(n)$ , sendo  $n$  o número de elementos da lista.

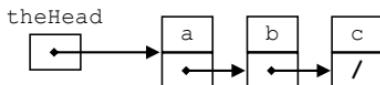
```
public Object get(int n) {  
    Node aux=theHead;  
    for(int i=1;i<n;i++)  
        aux=aux.next;  
    return aux.item;  
}
```

Para obter a referência do  $n$ -ésimo objecto basta iniciar a referência auxiliar `aux` com a cabeça da lista e avançar  $n-1$  vezes para o próximo nó. Se o parâmetro  $n$  for igual a 1 o ciclo não é executado (a guarda  $i < n$  é imediatamente falsa).

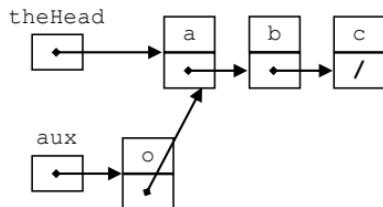
```
public void addBegin(Object o) {  
    Node aux = new Node(o, theHead);  
    theHead = aux;  
}
```

Inserir no início da lista tem duas fases: (i) construir um novo nó que guarde a informação a ser inserida (i.e., a referência `o` para o objecto) e (ii) actualizar a lista de forma adequada. A tarefa (i) e parte da (ii) são executadas pela primeira instrução. A actualização da lista é concluída na segunda instrução. Graficamente:

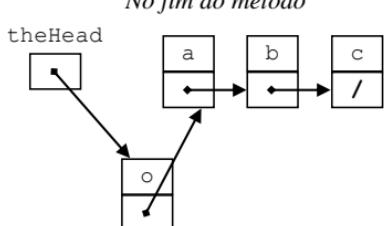
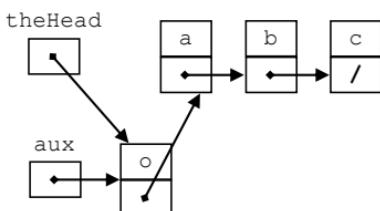
*A lista antes da invocação do método*



*Depois da 1ª instrução*



*Depois da 2ª instrução*



Quando o método termina, a variável interna `aux` é apagada da memória. A lista tem agora um novo objecto referenciado no início da lista. De facto, a variável `aux` não é necessária podendo o conteúdo do método resumir-se a: `theHead = new Node(o, theHead)`.

```

public void addEnd(Object o) {
    Node n = new Node(o, null);
    if (isEmpty())
        theHead = n;
    else {
        Node aux=theHead;
        while (aux.next!=null)
            aux=aux.next;
        aux.next = n;
    }
}
  
```

A inserção no fim da lista é mais complicada. Cria-se um novo nó com a referência para o objecto. Como será o último elemento da lista, a referência para o próximo nó é nula. Trata-se inicialmente o caso da lista vazia (onde se armazena o nó na cabeça da lista). Já para o caso geral (a lista não vazia) é preciso percorrer a lista até chegar ao último nó. Para que funcione é preciso que a guarda do ciclo seja `aux.next!=null` e não `aux!=null` (porquê?). Quando chegamos ao último nó é actualizada a referência para o próximo nó para armazenar a referência do nó recentemente criado (`aux.next = n`).

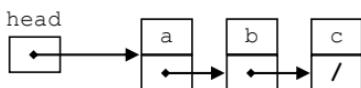
```
public void add(Object o, int n) {
    if (n==1)
        addBegin(o);
    else {
        Node aux = theHead;
        for (int i=1;i<n-1;i++)
            aux = aux.next;
        aux.next = new Node(o,aux.next);
    }
}
```

O método `add()` é uma generalização dos dois anteriores: inserir um elemento na n-ésima posição.

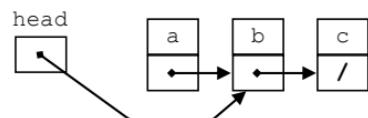
```
public void removeBegin() {
    theHead = theHead.next;
}
```

Para remover o nó do início, o método `removeBegin()` conta com o mecanismo de recolha de lixo do Java.

*A lista antes da invocação do método*



*Depois da 1ª instrução*



Depois da primeira instrução ser executada, o nó que estava anteriormente à cabeça não possui qualquer referência para a sua posição de memória podendo ser apagado na próxima recolha de lixo.

```
public void removeEnd() {
    if (theHead.next==null)
        theHead=null;
    else {
        Node aux = theHead;
        while (aux.next.next != null)
            aux = aux.next;
        aux.next = null;
    }
}
```

O método `removeEnd()` também funciona recorrendo à recolha de lixo, mas efectua essa tarefa no fim da lista. Como a pré-condição do método assume que a lista não é vazia, é desnecessário tratar esse caso particular.

```
public List tail() {
    DList l = new DList();
    l.theHead = theHead.next;
    return l;
}
```

Este método devolve uma nova lista igual ao resto da lista antiga, i.e., com todos os elementos menos o primeiro. Foi tomada a opção de não efectuar uma cópia da lista. A nova lista foi iniciada com a referência do segundo nó. As duas listas partilham a mesma memória.

Dependendo do problema, esta propriedade pode não ser satisfatória. Nesse caso, seria necessário criar uma cópia da lista original (usando o método `clone()`) para de seguida remover o primeiro nó. Se a configuração dos objectos `DList` fosse constituída por dois atributos (por exemplo, a cabeça `theHead` e o número de elementos `size`) seríamos obrigados a criar uma cópia. Porquê? Observe as seguintes linhas:

```
DList x = new DList();
x.addBegin(o1);
x.addBegin(o2);
x.addBegin(o3);
DList y = (DList)x.tail();
y.removeBegin();
```

O que ocorreu? No fim desta execução, a variável `y` refere uma lista constituída por um elemento (`o1`) e a variável `x` uma lista com dois elementos (`o1` e `o3`). Porém o estado da lista referenciada por `x` é inválido: o atributo `size` é igual a 3 e deveria ser 2 (porque o elemento `o2` foi removido através de `y` e não de `x`). Ocorreu uma violação do contrato através de uma excessiva exposição da representação. Neste caso, a operação `tail()` deveria criar e devolver um clone.

Outro detalhe: o método `tail()` devolve um objecto da classe `DList`. Na especificação tinha de ser devolvido um elemento do tipo `List` (está correcto porque, por derivação, um objecto do tipo `DList` é igualmente do tipo `List`). No entanto, para usar o método é necessário realizar uma coerção de tipos, como no exemplo: `DList y = (DList)x.tail()`.

```
public boolean contains(Object o) {
    for(Node aux=theHead; aux!=null; aux=aux.next)
        if (aux.item.equals(o))
```

```

        return true;
    return false;
}

public int indexOf(Object o) {
    Node aux = theHead;
    int p = 0;
    while (aux!=null) {
        p++;
        if (aux.item.equals(o))
            return p;
        aux = aux.next;
    }
    return 0;
}

```

Os métodos `contains()` e `indexOf()` procuram sequencialmente um dado elemento na lista. O primeiro método verifica se o elemento existe, enquanto o segundo devolve a primeira posição onde se encontra um elemento igual ao fornecido no argumento.

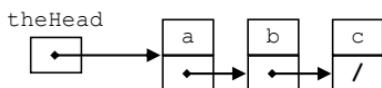
```

public void remFirst(Object o) {
    if (theHead.item.equals(o))
        theHead = theHead.next;
    else {
        Node aux = theHead;
        while (aux.next!=null &&
               !aux.next.item.equals(o))
            aux = aux.next;
        if (aux.next!=null)
            aux.next = aux.next.next;
    }
}

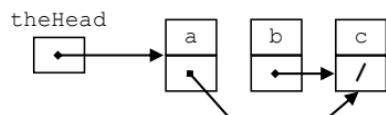
```

A remoção da primeira ocorrência de um elemento: percorre a lista e se o encontrar altera a referência `next` do nó anterior para armazenar o endereço do nó seguinte. Graficamente:

*A lista antes da invocação do método  
para eliminar o elemento 'b'*



*Antes de terminar o método*



```
public void remAll(Object o) {
    // remover todas as ocorrências do início da lista
    while (theHead!=null &&
           theHead.item.equals(o))
        theHead = theHead.next;

    if (theHead!=null) {          // se ainda há elementos...
        Node aux=theHead;

        while (aux.next!=null)
            if (aux.next.item.equals(o))
                aux.next = aux.next.next;
            else
                aux = aux.next;
    }
}
```

Na remoção de todos os elementos, o primeiro passo é remover os elementos (iguais ao argumento) que se encontrarem no início da lista. Em seguida é realizado o processo de remoção para a restante lista. Uma forma alternativa de implementar este método:

```
public void remAll(Object o) { // 2ª versão
    while (contains(o))
        remFirst(o);
}
```

Esta é uma implementação mais simples, legível e que utiliza métodos já definidos. O problema desta opção é o desempenho do algoritmo descrito. Enquanto a primeira versão atravessa a lista uma única vez, esta segunda versão atravessa a lista tantas vezes quantas as ocorrências do elemento a eliminar.

```
public boolean equals(Object l) {
    if (!(l instanceof List))
        return false;

    if (isEmpty() && ((List)l).isEmpty())
        return true;
        // versão rápida: mesma implementação
    if (l instanceof DList) {
        Node aux1 = theHead,
             aux2 = ((DList)l).theHead;

        while (aux1!=null && aux2!=null &&
               aux1.item.equals(aux2.item)) {
            aux1=aux1.next;
            aux2=aux2.next;
        }
    }
}
```

```
    return (aux1==null && aux2==null);
}
// versão lenta: implementações diferentes
int s = length();
if (s != ((List)l).length())
    return false;
for(int i=1;i<=s;i++)
    if (!get(i).equals(((List)l).get(i)))
        return false;
return true;
}
```

O método `equals()` compara duas listas. Se o objecto referido não for do tipo `List`, os dois objectos não podem ser iguais. Considera-se que duas listas são iguais se forem ambas vazias, ou sendo do mesmo tamanho têm os mesmos valores nas mesmas posições. Existe uma forma rápida de avaliar essa igualdade: (i) iniciar duas referências auxiliares com os inícios das duas listas, (ii) comparar os objectos referenciados pelos nós e caso sejam iguais actualizar ambos para os próximos nós e (iii) iterar a tarefa (ii) até uma das listas terminar. O resultado depende se a outra lista está vazia (são iguais) ou não (são diferentes porque têm dimensões diferentes). Dado  $n_1$  e  $n_2$  as dimensões das duas listas, a complexidade temporal é  $O(\min(n_1, n_2))$ .

Encontramos agora um problema: o algoritmo descrito só funciona se ambas as listas forem do tipo `DList`. Mas a interface que esta classe implementou refere que o argumento é do tipo `List`, o que não é a mesma coisa. Se o argumento é de outro tipo derivado, o método acima não funciona (a estrutura de informação da outra lista pode ser totalmente diferente). O que fazer? Optamos por uma solução mista. Verificamos se o parâmetro lista é ou não do tipo `DList` (através de: `l instanceof DList`). Se for, executa o método descrito acima. Caso contrário utiliza um método mais geral, mas mais lento: (i) verifica a dimensão da lista, (ii) se forem da mesma dimensão, itera para cada posição e compara os  $i$ -ésimos elementos das duas listas. Este algoritmo funciona para qualquer tipo de lista porque utiliza somente métodos descritos na interface. Porém, dentro do ciclo `for` é invocado o método `get()` (com complexidade linear), ou seja, este segundo algoritmo possui complexidade quadrática.

Um comentário final: poder comparar duas listas de classes diferentes não é necessariamente óbvio. Essa é uma das vantagens do uso de interfaces para definir o comportamento de um dado tipo de dados sem se estar limitado à implementação escolhida para uma dada classe. O método funcionará com outras classes desde que implementem a interface `List`.

Vejamos agora como concatenar duas listas:

```
public void concat(List l) {
    if (l.isEmpty())
        return;

    boolean wasEmpty = isEmpty();
    int length = l.length();
    Node aux = theHead;

    if (!wasEmpty)
        while (aux.next!=null)
            aux=aux.next;
            //aux referencia o ultimo nó da lista
    else
        aux = theHead = new Node(l.head(),null);
    for(int i=wasEmpty?2:1;i<=length;i++) {
        aux.next = new Node(l.get(i),null);
        aux = aux.next;
    }
}
```

Na concatenação surge o mesmo problema que na igualdade. Optámos (para não carregar demasiado o código fonte) pelo algoritmo geral: o método concatena o objecto com o argumento independentemente da sua representação (desde que implemente `List`). O método copia o conteúdo dos elementos da lista do argumento, um por um, adicionando-os no fim.

O próximo método inverte uma lista:

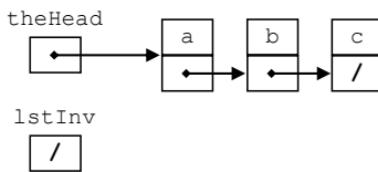
```
public void reverse() {
    if (theHead==null)
        return;

    DList lstInv = new DList();
    Node aux;

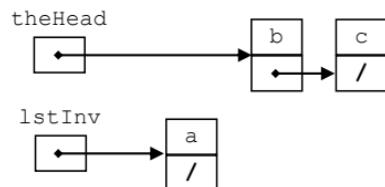
    while (theHead!=null) {
        aux = lstInv.theHead;
        lstInv.theHead = theHead;
        theHead = theHead.next;
        lstInv.theHead.next = aux;
    }
    theHead = lstInv.theHead;
}
```

O método inicia a execução verificando se a lista é vazia. Se for, termina (o inverso de uma lista vazia é uma lista vazia). Se a lista possui elementos é criada uma nova lista `lstInv` vazia. A iteração pode ser descrita da seguinte forma: enquanto a lista original não está vazia remover a cabeça da lista original e colocá-la no início da nova lista.

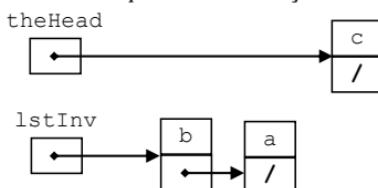
*As listas antes da 1<sup>a</sup> iteração*



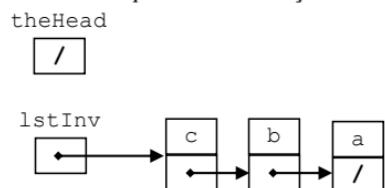
*Depois da 1<sup>a</sup> iteração*



*Depois da 2<sup>a</sup> iteração*



*Depois da 3<sup>a</sup> iteração*



Este algoritmo inverte as posições dos elementos da lista. O primeiro elemento a sair de `theHead` foi o primeiro a entrar em `lstInv`. Quando o ciclo terminar, esse primeiro elemento vai encontrar-se no fim. Finalmente, quando a lista original está vazia (`theHead==null`) basta associar a cabeça à nova lista invertida.

```
public Object clone() {
    DList copy = new DList();
    if (theHead!=null)
        copy.theHead = (Node)theHead.clone();
    return copy;
}
```

A cópia da lista tem o trabalho facilitado devido à forma como a clonagem dos nós se realiza (por cópia recursiva). O método cria uma nova lista (armazenada na referência `copy`) e caso a lista original não seja vazia é realizada a clonagem do primeiro nó. Este nó provoca a clonagem recursiva dos outros nós da lista. Resta colocar a referência dessa cópia na cabeça da nova lista e devolver o resultado.

```
/**
 * Traduz a lista numa string
 * @return a string que descreve a lista, eg. [3,4,6]
 */
```

```
public String toString() {
    if (isEmpty())
        return "[]";
    StringBuffer result =
        new StringBuffer("[" + theHead.item);
    Node aux = theHead.next;
    while(aux!=null) {
        result.append(", " + aux.item);
        aux = aux.next;
    }
    return result.toString() + "]";
}
```

A lista e todas as outras classes que implementem TDAs incluirão o método `toString()`. Este método traduz o estado do objecto para uma *string*.

```
} // endClass DList
```

## Usar a classe *DList*

No seguinte exemplo, as duas listas armazenam números inteiros (i.e., referências para objectos da classe `Integer`):

```
public static void main (String[] args) {
    DList l = new DList(), m = new DList();
    l.cons(new Integer(1));
    l.cons(new Integer(2));
    l.cons(new Integer(3));
    m.addBegin(new Integer(5));
    m.addBegin(new Integer(6));
    m.addEnd(new Integer(4));
    System.out.println(l);
    System.out.println(m);
    m.concat(l.tail());
    l.cons(new Integer(0));
    System.out.println(l);
    m.reverse();
    System.out.println(m);
} //end main()
```

□ [3,2,1] ↪  
[6,5,4]

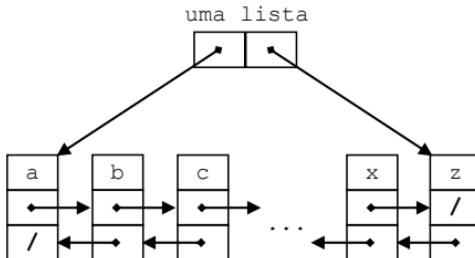
$[0, 3, 2, 1] \leftarrow$   
 $[1, 2, 4, 5, 6] \leftarrow$

## 14.4 Outros tipos de Listas

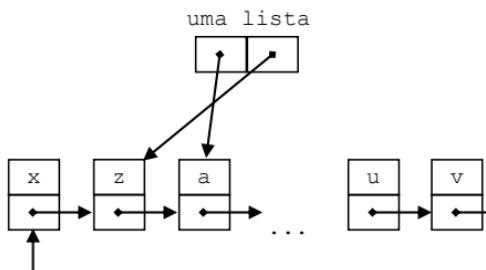
A lista especificada neste capítulo permite um acesso ao  $i$ -ésimo elemento. A lista ligada restringe a disciplina de acesso: a informação armazenada é acedida por um único caminho possível, através de um acesso sequencial: da primeira informação armazenada acede-se à segunda, a partir da segunda acede-se à terceira e assim sucessivamente... A lista ligada é uma **estrutura de dados linear**.

Existem outras possibilidades para implementar listas que podem melhorar o desempenho global da estrutura de dados. Normalmente, o preço a pagar pela melhoria de tempo é o aumento da memória necessária para representar um objecto lista. Um exemplo é o método `length()`. O método percorre a lista (de dimensão  $n$ ) sendo um algoritmo de complexidade  $\Theta(n)$ . Se decidirmos aumentar a configuração da classe lista de modo a incluir um atributo que guarde a informação sobre o número de nós actual, a resposta da invocação do método `length()` é imediata (basta consultar o valor do atributo). A complexidade neste caso é constante, i.e.,  $\Theta(1)$ . Qual foi o preço? O aumento da memória necessária para armazenar a configuração da lista. O compromisso entre tempo e espaço (uma maior rapidez paga-se com mais memória e um menor espaço paga-se com maior espera) é assunto que surge recorrentemente no desenvolvimento da maioria das estruturas de dados. A prática da programação e a experiência daí decorrente são, provavelmente, as melhores conselheiras.

A **lista duplamente ligada** (do inglês, *double linked list*) é uma lista constituída por nós com duas referências: uma para o nó seguinte e outra para o nó anterior. Isto permite andar para trás e para a frente entre os elementos da lista de uma forma mais versátil e eficiente. A lista duplamente ligada é configurada por dois atributos: uma cabeça e uma cauda. Permite aceder em tempo constante ao fim da lista (a inserção/remoção do fim da lista é tão rápida como no início da lista).



Uma outra possibilidade é a **lista circular** (do inglês, *circular list*). Na lista circular todos os elementos estão ligados entre si (o próximo elemento do último é o primeiro) e existem (por exemplo) duas referências que armazenam onde a lista começa e onde acaba.



É possível combinar estes dois últimos conceitos para obter uma lista circular duplamente ligada.

---

## Exercícios

1. Estenda a especificação do TDA lista para incluir as operações: (i) `subList`, que dado duas listas verifica se a segunda é uma sublista da primeira; (ii) `lastIndexOf`, que dado uma lista e um objecto devolve a última posição desse objecto na lista; (iii) `includes`, que dado duas listas verifica se os elementos da segunda pertencem todos à primeira.
2. Estenda a interface `List` e a implementação `DList` tendo em conta o exercício anterior.
3. Determine a complexidade temporal dos métodos apresentados na classe `DList`.
4. Discuta como as listas duplamente ligadas e circulares poderiam diminuir a complexidade temporal dos métodos analisados no exercício anterior.
5. Implemente o método `normalize()` onde dado uma lista de objectos da classe `Integer` devolve uma nova lista com os elementos normalizados entre 0 e 1. Para normalizar é necessário descobrir o maior valor usado para dividir todos os valores da lista.
6. Implemente o método `swap()` onde dada uma lista e duas posições válidas devolve uma nova lista onde foram trocados os elementos dessas duas posições.
7. Implemente o método `merge()` onde dadas duas listas de objectos da classe `Integer` com os elementos ordenados de forma crescente, devolve uma nova lista com todos os elementos das duas listas iniciais e igualmente ordenada de forma crescente.
8. Implemente um método `split()` que recebe uma lista de objectos da classe `Integer` e um valor, crie duas novas listas (uma com todos os elementos menores da lista inicial e

outra com os restantes valores) e devolva as referências dessas novas listas (como devolver dois objectos, quando os métodos Java apenas permitem devolver um?).

9. Discuta formas de implementar listas onde a dimensão estimada se situe na ordem das centenas de milhar de elementos. Como fazer para melhorar o desempenho no acesso a este tipo específico de listas? (*pista:* os elementos de uma lista podem armazenar listas). De que modo a solução modificaria a complexidade temporal dos algoritmos associados?

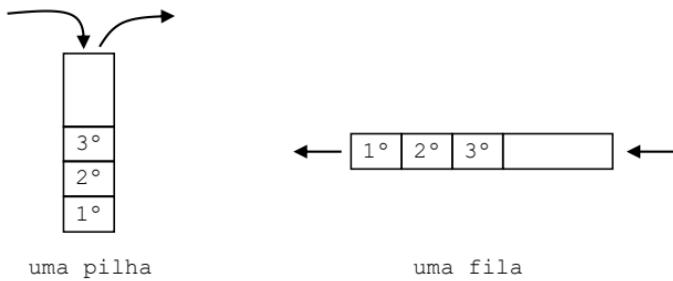


# 15 – Pilhas e Filas

---

Veremos neste capítulo duas estruturas de informação linear: a pilha e a fila. Como nas listas ligadas, estas estruturas armazenam os elementos de forma sequencial. Para aceder ao segundo elemento tem de se aceder ao primeiro. A diferença fundamental entre a lista e estes dois tipos é uma maior restrição no acesso à informação. Enquanto na lista é possível aceder livremente ao i-ésimo elemento, na fila e na pilha não é possível.

Na **pilha** (do inglês, *stack*) a informação é armazenada de forma a aceder somente ao último elemento inserido. Este modo de acesso designa-se por LIFO (do inglês, *Last In First Out*). Este género de restrição é bastante comum (por exemplo, empilhar caixotes, “desfazer” a última acção nos processadores de texto, na entrada e saída de automóveis de um barco com um só acesso...).



Na **fila** (do inglês, *queue*) a informação é armazenada de forma a aceder somente ao primeiro elemento inserido. É o acesso FIFO (do inglês, *First In First Out*). Esta restrição

é igualmente comum na nossa actividade diária (por exemplo, para pagar no supermercado, nas portagens, na entrada e saída de automóveis num barco com a porta de entrada no lado oposto à da saída...).

## 15.1 Pilha

### Especificação

É comum designar por *push* o processo de inserção na pilha e por *pop* o processo de remoção. Costuma designar-se por topo (do inglês, *top*) o último elemento inserido. Desta forma, o *pop* remove o topo da pilha e o *push* insere um novo elemento no topo da pilha (o topo anterior passa a ficar na segunda posição). Usaremos estes nomes na especificação.

A representação de uma pilha é definida recursivamente a partir de duas operações: *empty* que representa a pilha vazia e *push* que representa uma pilha constituída por uma informação (de um qualquer tipo *Element*) no topo e uma pilha restante. Estas duas operações são os construtores do tipo.

As outras operações definidas sobre este TDA são:

- *isEmpty* – verifica se a pilha está vazia
- *top* – devolve o elemento que está no topo da pilha
- *pop* – devolve a pilha sem o topo
- *equals* – verifica se duas pilhas são iguais

Segue o Tipo de Dados Abstracto:

```
especificação Stack<Element> =
importa Boolean
géneros Stack
operações
    empty : → Stack
    push : Stack Element → Stack
    isEmpty : Stack → Boolean
    top : Stack ↗ Element
    pop : Stack ↗ Stack
    equals : Stack Stack → Boolean
axiomas
    isEmpty(empty)      = TRUE
    isEmpty(push(S,I)) = FALSE
    top(push(S,I))   = I
```

```
pop(push(S,I)) = S
equals(S1,S2) =
  if isEmpty(S1)
    then isEmpty(S2)
  else not isEmpty(S2) and
    (equals(top(S1),top(S2)) and
      equals(pop(S1),pop(S2)))
```

Duas pilhas são iguais se e só se são ambas vazias ou contêm os mesmos elementos pela mesma ordem.

#### **pré-condições**

```
pop(S) requer not isEmpty(S)
top(S) requer not isEmpty(S)
```

#### **fim-especificação**

Seria admissível pensar que as funções *pop* e *top* fossem executadas por uma mesma função que devolveria o elemento do topo e a pilha sem esse elemento. A assinatura seria:

```
ttop: Stack → Stack * Element
```

O símbolo \* representa o produto cartesiano dos dois tipos, ou seja, a função devolve um tuplo (neste caso, o tuplo é um par). A assinatura é associada ao axioma  $ttop(push(S,I)) = (S,I)$ . Porém, consideramos que cada função deve devolver um e um só tipo de dados. Quando é necessário devolver duas ou mais informações diferentes, criam-se duas ou mais operações distintas.

## A Interface

Obtida a especificação, construímos a interface seguinte (no ficheiro *Stack.java*):

```
package dataStructures;

/**
 * <p>Titulo: A interface do TDA Pilha</p>
 * <p>Descrição: O desenho da Pilha (Stack) com contratos</p>
 */
public interface Stack {
  //@ public invariant !isEmpty() ==> top() != null;

  É definida a mesma invariante das listas: a estrutura não armazena referências nulas.
  /**
   * Criar uma Pilha vazia
   * @return uma Pilha vazia
   */
  //@ ensures \result.isEmpty();
```

```
/*@ pure */ Stack empty();
/**
 * Inserir um elemento no topo da Pilha
 * @param item A referência do elemento a inserir no topo
 */
/*@ requires item != null;
 @ requires !isEmpty() ==>
 @         item.getClass() == top().getClass();
 @ ensures top().equals(item);
 */
void push(Object item);
```

Na pós-condição do construtor `push` garantimos que uma vez concluída a execução do método, o elemento está no topo da pilha (como especificado no TDA). A segunda pré-condição verifica se o novo elemento é do tipo dos elementos armazenados. A estrutura de dados é polimórfica (dado guardar elementos de qualquer classe deriva de `Object`) mas passa a estar restrita a elementos do mesmo tipo.

```
/**
 * A Pilha está vazia?
 * @return TRUE se está vazia, FALSE c.c.
 */
/*@ pure */ boolean isEmpty();
/**
 * Devolver o topo da Pilha
 * @return a referência para o elemento no topo da Pilha
 */
//@ requires !isEmpty();
/*@ pure */ Object top();
/**
 * Remover o topo de uma Pilha
 */
//@ requires !isEmpty();
//@ ensures (* POP(PUSH(S,I)) = S *);
void pop();
```

Observemos o comentário sobre a pós-condição não verificável do método. Essa pós-condição seria semelhante a: `\old(this).equals(this.push(\old(top())))`, a pilha inicial deve ser igual à pilha final onde fosse empilhado o topo da pilha inicial. Isto não pode ser testado porque a asserção, a ser avaliada, produziria o efeito secundário de modificar o próprio objecto. Mas as asserções não podem criar efeitos secundários. Nem sempre é possível expressar nos contratos o que foi especificado.

```
 /**
 * Serão as duas Pilhas iguais?
 * @param s A pilha a ser comparada
 * @return TRUE se forem iguais, FALSE c.c.
 */
//@ also
//@ requires s != null;
/*@ pure @*/ boolean equals(Object s);
/**
 * Devolver uma cópia da estrutura.
 * @return a referencia para a cópia
 */
//@ also
//@ ensures (\result != this) && equals(\result);
/*@ pure @*/ Object clone();
} // endInterface Stack
```

## Uma Implementação Estática

Esta implementação usa uma representação estática de informação, nomeadamente um vector para armazenar os dados da pilha. Este vector está relacionado com um atributo inteiro que determina o topo actual. O valor desse atributo será incrementado ou decrementado consoante sejam inseridos ou removidos elementos.

```
package dataStructures;
/** Titulo: Uma implementação vectorial da Pilha */
public class VStack implements Stack,Cloneable {
```

Seguindo a convenção, dado tratar-se de uma implementação estática do tipo Stack, a classe denomina-se VStack.

```
private final int DELTA = 128;
private Object[] theStack;
private int theTop;
```

A configuração da classe é constituída por três atributos: (i) uma constante inteira DELTA que define a dimensão do vector bem como o seu eventual crescimento (o vector pode esgotar-se, sendo necessário substitui-lo por um maior), (ii) um vector de referências theStack para os objectos que a pilha armazena e (iii) o atributo inteiro theTop que indica o índice da referência para o objecto do topo.

Para caracterizar a pilha são apenas necessários os dois últimos atributos. O atributo DELTA diz respeito à implementação do referido crescimento. Este exemplo mostra a existência de dois géneros de atributos: os que definem a configuração da classe e cujos

valores determinam os estados dos objectos (como `theStack` e `theTop`) e os úteis na construção da classe (como `DELTA`) mas não fazem parte da configuração.

```
public VStack(int cap) {
    theStack = new Object[DELTA];
    theTop    = -1;
}
public Stack empty() {
    return new VStack();
}
```

É inicializada a configuração do objecto para descrever uma pilha vazia. O atributo do topo é negativo quando não existem objectos armazenados. Convencionamos este valor de `-1` a ser utilizado na interrogação `isEmpty()`:

```
public boolean isEmpty() {
    return theTop == -1;
}
public void push(Object item) {
    if (theTop+1 == theStack.length)
        grow();
    theTop++;
    theStack[theTop] = item;
}
```

A inserção de um novo elemento passa primeiro por conferir se a capacidade actual da pilha é suficiente. Se houver espaço, basta incrementar o atributo `theTop` (onde vai se situar o próximo topo) e inserir a referência do objecto nessa posição do vector (o novo objecto do topo). Se não houver espaço, o método vai reservar mais espaço através do método privado `grow()` criando um vector maior para onde se copia o conteúdo da pilha:

```
private void grow() {
    Object[] newStack =
        new Object[theStack.length + DELTA];
    for(int i=0;i<theStack.length;i++)
        newStack[i] = theStack[i];
    theStack = newStack;
}
public Object top() {
    return theStack[theTop];
}
```

```
public void pop() {  
    theStack[theTop--] = null;  
}
```

No método `pop()` é conveniente colocar as posições não utilizadas a um valor nulo. Caso contrário, enquanto o vector da pilha não fosse ocupado por nova informação, os objectos anteriormente armazenados na pilha ficariam com uma referência que impediria a sua eventual libertação pela recolha de lixo.

```
public boolean equals(Object s) {  
    if (!(s instanceof Stack))  
        return false;  
  
    int i = theTop;  
    Stack cp = (Stack) (((Stack)s).clone());  
  
    while (!cp.isEmpty()) {  
        if (i<0 || !theStack[i].equals(cp.top()))  
            return false;  
        cp.pop();  
        i--;  
    }  
  
    return i<0;  
}
```

Há um problema semelhante ao `equals()` das listas: não temos a certeza que a pilha dada pelo parâmetro é do tipo `VStack`. Consideramos aqui apenas o caso geral. Como não se deve alterar a pilha dada, faz-se uma cópia da mesma para se remover os elementos um por um. Para cada elemento retirado verificamos se os elementos na mesma posição são iguais (é usado o método `equals()` dos objectos armazenados). Se uma dessas comparações for falsa, ou as dimensões das pilhas forem diferentes, o método devolve falso. Caso contrário, o método devolve verdadeiro.

```
public Object clone() {  
    VStack cp = new VStack();  
    cp.theStack = new Object[theStack.length];  
    for(int i=0;i<=theTop;i++)  
        cp.theStack[i] = theStack[i];  
    cp.theTop = theTop;  
    return cp;  
}
```

A clonagem da pilha passa por construir uma nova pilha com a mesma capacidade e armazenar as referências dos objectos na nova pilha. A clonagem da pilha não duplica os objectos, apenas duplica as referências para esses objectos. Porquê tomar esta decisão na

clonagem? Existem tipos de objectos que necessitam de enormes recursos de memória, quando se implementa a clonagem não se pode saber se a pilha não vai armazenar elementos desse género. Clonar tudo pode ser incompatível, enquanto clonar apenas as referências custa sempre o mesmo (nos sistemas operativos de 32 bits, cada referência ocupa 32 bits).

```
public String toString() {
    if (this.isEmpty())
        return "[]>";
    StringBuffer result =
        new StringBuffer("[ " + theStack[0]);
    for (int i=1;i<=theTop;i++)
        result.append(", " + theStack[i]);
    return result.toString() + "]>";
}
```

A representação textual é semelhante à usada para a lista, mas adiciona o carácter ‘>’ como indicação do topo da pilha.

```
} // endClass VStack
```

## Usar a classe **VStack**

Segue um exemplo de uso desta classe. As duas pilhas criadas armazenam referências para objectos da classe `Integer`:

```
public static void main(String[] args) {
    VStack v1 = new VStack(),
        v2 = new VStack();
    v1.push(new Integer(3));
    v1.push(new Integer(4));
    v2.push(new Integer(3));
    v2.push(new Integer(4));
    v1.pop();
    System.out.println(v1 + " " + v2.clone());
    System.out.print(v1.equals(v2) ? "==" : "!=");
}
□ [3]> [3,4]> ↵
!=
```

## 15.2 Fila

### Especificação

É usual designar por *enqueue* o processo de inserção na fila e por *dequeue* o processo de remoção. Usaremos estes nomes na especificação. A representação de uma fila é definida recursivamente a partir de duas operações: *empty* que representa a fila vazia e *enqueue* que representa uma fila constituída por uma informação (de um qualquer tipo *Element*) situada no fim e a restante fila. Estas duas operações são os construtores do tipo.

As outras operações definidas sobre este TDA são:

- *isEmpty* – verifica se a fila está vazia
- *front* – devolve o elemento que está na frente da fila
- *dequeue* – devolve a fila sem o elemento inicial
- *equals* – verifica se duas filas são iguais

Segue o Tipo de Dados Abstracto:

```
especificação Queue<Element> =
importa Boolean
géneros Queue
operações

    empty : → Queue
    enqueue : Queue Element → Queue
    isEmpty : Queue → Boolean
    front : Queue → Element
    dequeue : Queue → Queue
    equals : Queue Queue → Boolean

axiomas

    isEmpty(empty)           = TRUE
    isEmpty(enqueue(Q, I))  = FALSE
    front(enqueue(Q, I))   =
        if isEmpty(Q)
        then I
        else front(Q)
    dequeue(enqueue(Q, I)) =
        if isEmpty(Q)
        then empty
        else enqueue(dequeue(Q), I)
```

Os axiomas das operações `front` e `dequeue` são mais complicados que os axiomas das operações `top` e `pop` da pilha, dado ser necessário aceder ao primeiro elemento inserido na estrutura. Na pilha, o elemento a consultar ou remover é o último inserido (sendo de mais fácil acesso).

```
equals(Q1,Q2) =  
    if isEmpty(Q1)  
        then isEmpty(Q2)  
        else not isEmpty(Q2) and  
            (equals(front(Q1), front(Q2)) and  
             equals(dequeue(Q1), dequeue(Q2)))
```

Existe uma descrição alternativa para a função `equals` mais eficiente do ponto de vista computacional (qual?). Porém, na especificação não estamos preocupados com a rapidez dos eventuais algoritmos descritos<sup>30</sup>. Apenas pretendemos definir (sem ambiguidades) a semântica das operações.

#### **pré-condições**

```
front(S) requer not isEmpty(Q)  
dequeue(S) requer not isEmpty(Q)
```

#### **fim-especificação**

## A Interface

Com a especificação definimos a interface (no ficheiro `Queue.java`):

```
package dataStructures;  
  
/**  
 * <p>Tituto: A interface do TDA Fila</p>  
 * <p>Descrição: O desenho da Fila (Queue) com contratos</p>  
 */  
  
public interface Queue {  
    //@ public invariant !isEmpty() ==> front() != null;  
    /**  
     * Criar uma Fila vazia  
     * @return uma Fila vazia  
     */  
    //@ ensures \result.isEmpty();  
    /*@ pure @*/ Queue empty();
```

---

<sup>30</sup> Estas descrições das operações podem ser codificadas quase directamente numa linguagem funcional, como por exemplo no ML ou no Haskell.

```
/**
 * Inserir um elemento no fim da Fila
 * @param item A referência do elemento a inserir no fim
 */
/*@ requires item != null;
 @ requires !isEmpty() ==>
 @ ensures isEmpty();
 @ ensures \old(isEmpty()) ==> front().equals(item);
 @ ensures !(\old(isEmpty())) ==>
 @ ensures front().equals(\old(front()));
 */
void enqueue(Object item);
```

Para além da pré-condição referente à invariante (não permite a inserção de referências nulas), o construtor garante nas pós-condições que: (i) a fila não fica vazia; (ii) se a fila estava vazia o elemento inserido encontra-se na frente da fila e (iii) se a fila não estava vazia, o elemento na frente da fila não se alterou com a execução do método.

```
/**
 * A Fila está vazia?
 * @return TRUE se está vazia, FALSE c.c.
 */
/*@ pure */ boolean isEmpty();
/***
 * Devolver o primeiro elemento da Fila
 * @return uma referência do elemento na frente da Fila
 */
/*@ requires !isEmpty();
 /*@ pure */ Object front();

/***
 * Remover o primeiro elemento da Fila
 */
/*@ requires !isEmpty();
 /*@ ensures (*
 @ dequeue(enqueue(Q,I)) =
 @         if isEmpty(Q) then empty
 @         else enqueue(dequeue(Q),I) *)
 */
void dequeue();

/***
 * São as duas Filas iguais?
 * @param q A fila a ser comparada
 * @return TRUE se forem iguais, FALSE c.c.
 */

```

```

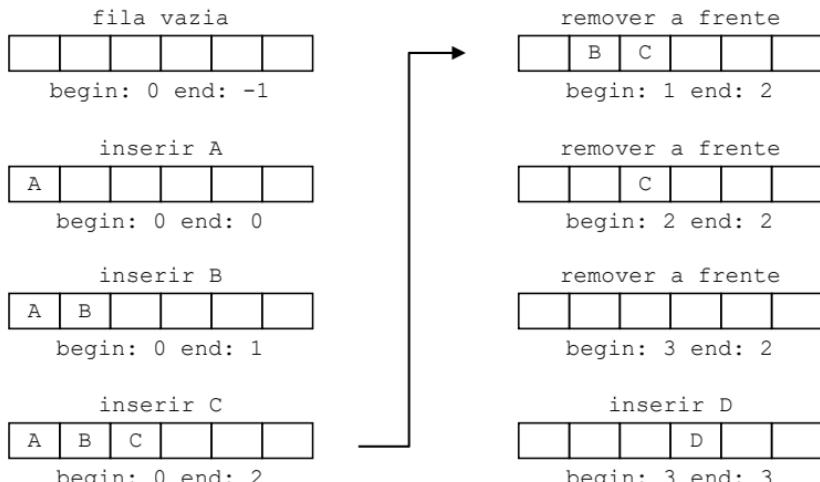
//@ also
//@ requires q != null;
/*@ pure @*/ boolean equals(Object q);
/**
 * Devolver uma cópia da estrutura.
 * @return a referência para a cópia
 */
//@ also
//@ ensures (\result != this) && equals(\result);
/*@ pure @*/ Object clone();
} // endInterface Queue

```

## Uma Implementação Estática

A escolha nesta implementação recaiu sobre uma representação estática: um vector. No entanto, a gestão da memória é um pouco mais complicada que a efectuada na pilha. Não se pode usar somente um inteiro para indicar onde se deve inserir e remover elementos porque agora estes estão em dois locais diferentes (insere-se no fim, remove-se do início). Nesta representação utilizamos dois atributos: um para indicar o índice do vector onde se encontra o primeiro objecto da fila (designado *begin*) e outro para indicar o índice do vector onde se encontra o último objecto (designado *end*).

Vejamos um exemplo:

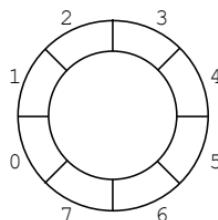


Algumas notas sobre esta representação da fila:

- Quando se insere um elemento incrementa-se *end*.

- Quando se remove um elemento incrementa-se *begin*.
- A fila está vazia quando o valor *end* é igual ao valor *begin* menos um.
- Depois das sete inserções/remoções do exemplo, os valores *begin* e *end* são iguais a 3. Como o vector tem uma dimensão de seis posições, metade da capacidade já está inutilizada (os índices 0 a 2) pois ainda não referimos nenhuma forma de reutilizar essa memória.

Como resolver este problema? Convencionando que se num incremento dos atributos *begin* ou *end*, o seu valor for igual ao último índice, o próximo valor desse atributo é zero. Desta forma é sempre possível aproveitar todo o espaço disponível porque o vector é visto como um **vector circular**. Em termos gráficos, um exemplo de um vector circular com oito posições:



Existe uma outra questão: numa fila vazia, o valor de *end* é igual ao valor de *begin* menos um (como vimos atrás); numa fila cheia, o valor de *end* é igual ao valor de *begin* menos um (foi dada uma volta inteira). Como distinguir estas duas situações tão opostas? Dois métodos: (i) deixar uma célula do vector por usar (diminuindo a capacidade global da fila) ou (ii) usar um terceiro atributo (o número de elementos da fila) para distinguir as duas situações (aumentando a configuração dos objectos). Na implementação desta secção é utilizado um vector circular com a opção (i). Assim, a célula que fica por usar é a célula imediatamente antes da primeira informação. A fila está cheia quando o fim estiver a uma célula de distância do princípio (verificar a condição inicial do método enqueue() ).

```
package dataStructures;  
  
/**  
 * <p>Titolo: Uma implementação vectorial da Fila</p>  
 * <p>Descrição: A implementação usa um vector circular</p>  
 */  
  
public class VQueue implements Queue, Cloneable {  
    private final int DELTA = 128;  
    private Object[] theQueue;  
    private int begin, end;  
  
    public VQueue() {  
        theQueue = new Object[DELTA];  
    }
```

```
begin      = 0;
end        = theQueue.length - 1;
}

public Queue empty() {
    return new VQueue();
}
```

Foi novamente definido o construtor da classe e o construtor da especificação. Numa fila vazia, o atributo `end` deve ser igual ao valor de `begin` menos um. Se o `begin` é zero, o `end` seria `-1`. Porém, numa aritmética de módulo `theQueue.length`, o predecessor de zero é `theQueue.length-1`.

```
public boolean isEmpty() {
    return (end+1)%theQueue.length == begin;
}

private void grow() {
    Object[] newQueue =
        new Object[theQueue.length + DELTA];
    for (int i=begin, j=0;
          i!=(end+1)%theQueue.length;
          i=(i+1)%theQueue.length, j++)
        newQueue[j] = theQueue[i];
    begin      = 0;
    end        = theQueue.length-2;
    theQueue = newQueue;
}

public void enqueue(Object item) {
    if ((end+2)%theQueue.length == begin)
        grow();
    end           = (end+1) % theQueue.length;
    theQueue[end] = item;
}
```

No método `enqueue()` é incrementado o valor de `end` e armazenada a referência. A actualização de `end` é executada em aritmética módulo `theQueue.length`. Esta aritmética é implementada através do operador `%` que devolve o resto de divisão inteira. Os valores possíveis do resto da divisão dado um divisor  $N$  são os valores  $0, 1, \dots, N-1$ . Um exemplo para  $N=5$ :  $0\%5=0$ ,  $1\%5=1$ ,  $2\%5=2$ ,  $3\%5=3$ ,  $4\%5=4$ ,  $5\%5=0$ ,  $6\%5=1$ ,  $7\%5=2$ ,  $8\%5=3$ ,  $9\%5=4$ ,  $10\%5=0$ , ...

```
public Object front() {
    return theQueue[begin];
}
```

```
public void dequeue() {
    theQueue[begin] = null;
    begin = (begin+1) % theQueue.length;
}
```

No método `dequeue()` a remoção de um elemento faz-se da forma semelhante à inserção. É actualizado o valor do `begin` (aumentar uma unidade em aritmética modular) e decrementado o número de elementos.

```
public boolean equals(Object q) {
    if (!(q instanceof Queue))
        return false;
    int actual = begin;
    Queue cp = (Queue)((Queue)q).clone();
    while (!cp.isEmpty()) {
        if (actual==(end+1)%theQueue.length ||
            !theQueue[actual].equals(cp.front()))
            return false;
        cp.dequeue();
        actual = (actual+1)%theQueue.length;
    }
    return actual==(end+1)%theQueue.length;
}
```

A comparação entre dois objectos do tipo `Queue` (não do tipo `VQueue`) passa pela criação de um clone do argumento. Do clone são retirados todos os elementos para se compararem com os elementos da estrutura interna do objecto que invocou o método.

```
public Object clone() {
    VQueue copy = new VQueue();
    copy.theQueue = new Object[theQueue.length];
    copy.begin    = begin;
    copy.end      = end;
    for(int i=0;i<theQueue.length;i++)
        copy.theQueue[i] = theQueue[i];
    return copy;
}
```

O método `clone()` limita-se a percorrer todos os elementos armazenados na fila enquanto insere as referências na nova fila. Pelos motivos já referidos, não são copiados os objectos, apenas as referências.

```
public String toString() {
    if (this.isEmpty())
        return "<[]<";
    StringBuffer result =
        new StringBuffer("<[" + theQueue[begin]);
    for(int i=(begin+1)%theQueue.length;
        i!=(end+1)%theQueue.length;
        i=(i+1)%theQueue.length)
        result.append(", " + theQueue[i]);
    return result.toString() + "]<";
}
```

A representação textual das filas é semelhante à das listas, mas com o caracter < da esquerda a marcar o início da fila e o < da direita a marcar o fim.

```
} // endClass VQueue
```

## Usar a classe **VQueue**

Segue-se um exemplo do uso desta classe:

```
public static void main(String[] args) {
    VQueue v1 = new VQueue(),
           v2 = new VQueue();
    v1.enqueue(new Integer(5));
    v1.enqueue(new Integer(4));
    v1.enqueue(new Integer(3));
    v2.enqueue(new Integer(5));
    v2.enqueue(new Integer(4));
    v2.enqueue(new Integer(3));
    System.out.println(v1.equals(v2) ?
                        "v1==v2":"v1!=v2");
    v1.dequeue();
    VQueue v3 = (VQueue)v1.clone();
    v1.dequeue();
    System.out.println(v1 + " " + v2 + " " + v3);
}
□ v1==v2 ←
<[3]< <[5,4,3]< <[4,3]< ←
```

## Filas de Espera com Prioridade

Normalmente, as filas de espera não são tão simples como o descrito nas secções anteriores. Objectos diferentes podem possuir prioridades diferentes não sendo inseridos no fim da fila. Exemplos: os carros prioritários no código da estrada; as filas nos supermercados para menos de dez artigos; os assentos de grávidas, crianças ao colo e idosos nos transportes públicos; os processos executados por um sistema operativo; as filas de urgência num hospital. Para modelar este tipo de situações é conveniente utilizar uma **fila de espera com prioridade** (do inglês, *priority queue*).

Nas filas de espera com prioridade, cada objecto vem associado a um valor (numérico ou de outro tipo) que define a sua prioridade. O objecto ultrapassa todos os objectos com uma prioridade inferior até encontrar alguém com igual ou maior prioridade. Numa fila de trânsito inteiramente constituída por ambulâncias, uma nova ambulância não tem prioridade de passar à frente.

Um cuidado especial: deve-se evitar que um dado objecto de menor prioridade fique retido indefinidamente na fila (em inglês, *starvation*) devido à constante chegada de elementos de prioridade mais elevada. Duas soluções possíveis:

- Define-se que existe um tempo máximo de espera. Qualquer objecto que chegue a esse limite é imediatamente processado.
- A cada período de espera, os objectos aumentam de prioridade. Assim, há uma segurança que ao fim de um intervalo suficiente, o objecto é tratado devido à alta prioridade (é conveniente existir uma prioridade máxima).

---

## Exercícios

1. Implemente um método `checkPars()` que dada uma *string* constituída por vários parênteses – nomeadamente, `O[]{}()` – verifica se os parênteses estão bem posicionados. Por exemplo, `checkPars("([])")` devolveria `true` e `checkPars("([)]")` devolveria `false`.
2. Implemente um método `eval()` que calcule o valor de uma dada *string* contendo uma expressão aritmética (onde se incluem números positivos e os cinco operadores `+`, `-`, `*`, `/` e `%`) em notação polaca invertida (uma notação onde não são precisos parênteses). Por exemplo, a expressão “`(1+2)*3`” em notação polaca invertida é “`3 2 1 + *`”; a expressão “`(1+2)*3%(4+5)`” traduz-se por “`3 2 1 + * 4 5 + %`”.
3. Implemente um método `palin()` que dada uma *string* verifica se esta é um palíndromo (uma frase lida igualmente da esquerda para a direita e vice-versa, onde os acentos e pontuações não são levados em conta). Por exemplo, “`somos`”, “`O galo nada no lago`” ou “`Ramon: 'Atira o remo, Homero, à Rita, no mar!'`” são palíndromos.

4. Estenda a especificação do TDA pilha para incluir as funções: (i) `length`, que dada uma pilha devolve o número de elementos que contém; (ii) `reverse`, que dada uma pilha devolve uma pilha com o conteúdo invertido; (iii) `queue2stack`, que dada uma fila devolve a pilha onde o elemento do topo é o elemento da frente da fila.
5. Estenda a interface `Stack` e a implementação `VStack` tendo em conta o exercício anterior.
6. Determine a complexidade temporal dos métodos apresentados na classe `VStack`.
7. Estenda a especificação do TDA fila para incluir as funções: (i) `length`, que dada uma fila devolve o número de elementos que contém; (ii) `reverse`, que dada uma fila devolve uma fila com o conteúdo invertido; (iii) `stack2queue`, que dada uma pilha devolve a fila onde o elemento da frente é o elemento do topo da pilha e (iv) `concatenate`, que dadas duas filas devolve a concatenação das duas (i.e., o primeiro elemento da segunda fila fica a seguir ao último elemento da primeira fila).
8. Implemente a classe `DStack` que implemente a interface `Stack` através de uma representação dinâmica.
9. Estenda a interface `Queue` e a implementação `VQueue` tendo em conta o exercício anterior.
10. Determine a complexidade temporal dos métodos apresentados na classe `VQueue`.
11. Implemente a classe `DQueue` que implementa a interface `Queue` através de uma representação dinâmica.
12. Apresente a especificação da fila de prioridade (defina o TDA `pqueue`) que inclui as seguintes funções: (i) `empty`, para criar uma nova fila, (ii) `insert`, para inserir um novo elemento (do tipo `Element`) com uma dada prioridade (do tipo `Key`), (iii) `isEmpty`, que verifica se a fila está vazia, (iv) `length`, que devolve o número de elementos da fila, (v) `max`, que devolve o elemento com a maior prioridade, (vi) `extract`, que remove o elemento com a maior prioridade e (vii) `equals`, que compara se duas filas são iguais (duas filas são iguais se possuem os mesmos elementos com as mesmas prioridades).
13. Implemente a interface `PQueue` a partir da especificação anterior.
14. Implemente a classe `DPQueue` que implemente `PQueue` através de uma representação dinâmica.

# 16 – Conjuntos

---

Um **conjunto** (do inglês, *set*) é uma colecção sem repetições de objectos de um determinado Universo (ou nosso caso, tipo). Estes objectos são designados por membros ou elementos do conjunto.

Existe uma terminologia e uma notação de uso comum associada à Teoria dos Conjuntos. Se um objecto  $x$  pertence a um conjunto  $A$  escreve-se  $x \in A$ , senão escreve-se  $x \notin A$ . O conjunto com zero elementos,  $\emptyset$ , denota-se por **conjunto vazio**. A **cardinalidade** de um conjunto  $A$  é o número de objectos que pertencem a  $A$  e denota-se por  $|A|$ . Um conjunto é **finito** se contém um número finito de objectos, caso contrário, o conjunto é **infinito**<sup>31</sup>. É possível associar diversas operações aos conjuntos. Sejam  $A$  e  $B$  dois conjuntos:

- Está contido:  $A \subseteq B \Leftrightarrow \forall_{x \in A} x \in B$  e  $A \subset B \Leftrightarrow A \subseteq B \wedge A \neq B$
- Igual:  $A = B \Leftrightarrow A \subseteq B \wedge B \subseteq A$
- Diferente:  $A \neq B \Leftrightarrow \neg(A = B)$
- União:  $A \cup B = \{ x : x \in A \vee x \in B \}$
- Intersecção:  $A \cap B = \{ x : x \in A \wedge x \in B \}$
- Subtracção:  $A - B = \{ x : x \in A \wedge x \notin B \}$
- Remoção:  $A \setminus \{a\} = \{ x : x \in A \wedge x \neq a \}$
- Produto Cartesiano:  $A \times B = \{ (x,y) : x \in A \wedge y \in B \}$

---

<sup>31</sup> Um conjunto infinito com a mesma cardinalidade dos números naturais ( $N = \{0, 1, 2, 3, \dots\}$ ) diz-se **contável**, senão diz-se **não contável** (por exemplo, o conjunto dos números reais é não contável).

- Complemento (no universo U):  $\bar{A} = \{ x : x \in U \wedge x \notin A \}$

Dois conjuntos A e B são **disjuntos** se  $A \cap B = \emptyset$ .

## 16.1 A Especificação do Conjunto

Das operações referidas, especificamos as seguintes:

- `empty` – cria um conjunto vazio
- `insert` – dado um conjunto C e um elemento E devolve a união  $C \cup \{E\}$
- `isEmpty` – verifica se o conjunto está vazio
- `remove` – dado um conjunto C e um elemento E, devolve  $C \setminus \{E\}$  (se o elemento não pertencer ao conjunto, devolve C)
- `belongs` – verifica se um elemento pertence a um conjunto
- `length` – devolve a cardinalidade de um conjunto
- `contains` – verifica se um conjunto está contido noutra
- `equals` – verifica se dois conjuntos são iguais
- `union` – devolve a união de dois conjuntos
- `intersection` – devolve a intersecção de dois conjuntos
- `subtraction` – devolve a subtracção de dois conjuntos

Segue o Tipo de Dados Abstracto:

```
especificação Set<Element> =
importa Integer, Boolean
géneros Set
operações

    empty : → Set
    insert : Set Element → Set
    isEmpty : Set → Boolean
    belongs : Set Element → Boolean
    length : Set → Integer
    remove : Set Element → Set
    contains : Set Set → Boolean
    equals : Set Set → Boolean
    union : Set Set → Set
    intersection : Set Set → Set
    subtraction : Set Set → Set
```

**axiomas**

Escolhem-se os comandos `empty` e `insert` como construtores. Mas o uso sem restrições destes construtores define por indução elementos que não identificamos como conjuntos. Existem duas propriedades inerentes à noção de conjunto: (i) a ordem de inserção dos elementos é irrelevante e (ii) não existem repetições. Sem restrições sobre as construturas, as duas expressões seguintes seriam verdade:

```
insert(insert(empty,1),1) ≠ insert(empty,1)
insert(insert(empty,1),2) ≠ insert(insert(empty,1),2)
```

Respectivamente  $\{1,1\} \neq \{1\}$  e  $\{1,2\} \neq \{2,1\}$  o que não corresponde à noção de conjunto. Assim, são incluídos axiomas que limitam o conjunto induzido pelas duas operações:

```
insert(insert(S,I),I) = insert(S,I)
insert(insert(S,J),I) = insert(insert(S,I),J)
```

A primeira expressão garante que  $\{1,1\} = \{1\}$ . A segunda garante que a ordem de inserção é irrelevante,  $\{1,2\} = \{2,1\}$ . Com estas duas propriedades obtemos a definição pretendida para os conjuntos.

```
isEmpty(empty)      = TRUE
isEmpty(insert(S,I)) = FALSE
belongs(empty,I)    = false
belongs(insert(S,J),I) =
    equals(I,J) or belongs(S,I)
length(empty)       = 0
length(insert(S,J)) =
    if belongs(S,J)
        then length(S)
    else 1 + length(S)
```

Este axioma considera as propriedades da operação `insert` foi definida. Apesar das repetições serem irrelevantes, a descrição do conjunto pode contê-las. Por exemplo, o conjunto  $\{1\}$  pode ser descrito pela expressão `insert(insert(empty,1),1)`. A operação `length` leva este facto em conta.

O mesmo se passa com a operação de remoção.

```
remove(empty,I)      = empty
remove(insert(S,J),I) =
    if equals(I,J)
        then remove(S,I)
    else insert(remove(S,I),J)
contains(S,empty)     = true
```

```
contains(S, insert(T, I)) =  
    belongs(S, I) and contains(S, T)  
equals(S, T) =  
    contains(S, T) and contains(T, S)  
union(S, empty)      = S  
union(S, insert(T, I)) = insert(union(S, T), I)  
intersection(S, empty)     = empty  
intersection(S, insert(T, I)) =  
    if belongs(S, I)  
        then insert(intersection(S, T), I)  
    else intersection(S, T)  
subtraction(S, empty)      = S  
subtraction(S, insert(T, I)) =  
    if belongs(S, I)  
        then remove(subtraction(S, T), I)  
    else subtraction(S, T)  
pré-condições  
fim-especificação
```

## 16.2 A Interface Conjunto

Este TDA é diferente dos anteriores pelo seguinte motivo: não existe uma ordem de consulta aos elementos da estrutura. Na lista pode-se consultar qualquer elemento; na pilha há acesso ao elemento do topo; na fila ao elemento da frente. Nos conjuntos não há uma operação que nos permita consultar o “primeiro” elemento do conjunto.

### Iteradores

Para resolver este problema associamos ao tipo `Set` uma outra capacidade. Esta capacidade designada **iterador** é capaz de consultar o primeiro elemento (a partir de uma qualquer ordem) do conjunto bem como o elemento seguinte (se existir) ao anteriormente consultado. Um objecto do tipo iterador tem a capacidade de devolver todos os elementos contidos nele. A biblioteca `java.util` inclui uma interface representativa deste serviço: `Iterator`. Esta interface possui três operações: (i) `hasNext()` que verifica se o iterador ainda tem de percorrer mais elementos, (ii) `next()` devolve o próximo elemento, (iii) e `remove()` que apaga o elemento actual.

Uma classe que providencie este serviço deve possuir um método `iterator()` que devolve um iterador sobre os seus elementos. O exemplo seguinte mostra uma utilização de um iterador:

```
void imprimir(Set conjunto) {  
    Iterator it = conjunto.iterator();  
    while (it.hasNext())  
        System.out.println(it.next());  
}
```

O novo comando `for` inserido na versão Java 5 itera automaticamente sobre um iterador. Por exemplo, o método anterior pode ser expresso da seguinte forma.

```
void imprimir(Set conjunto) {  
    for (Object e: conjunto)  
        System.out.println(e);  
}
```

Os tipos enumerados também devolvem iteradores (através do método `values()`). Um exemplo de uso:

```
public class C {  
    public enum Dias { Dom, Seg, Ter, Qua, Qui, Sex, Sab };  
    ...  
    for(Dias d: Dias.values())  
        System.out.print(d);  
}  
□ DomSegTerQuaQuiSexSab
```

## A Interface Conjunto

Considere a seguinte interface do TDA conjunto (ficheiro `Set.java`):

```
package dataStructures;  
import java.util.*;  
  
/**  
 * <p>Titulo: A interface do TDA Conjunto</p>  
 * <p>Descrição: O desenho Conjunto (Set) com contratos</p>  
 */  
public interface Set {  
    /**  
     * Criar um Conjunto vazio  
     * @return um Conjunto vazio  
     */  
    // Ensures \result.isEmpty();  
    /*@ pure @*/ Set empty();  
    /**  
     * O conjunto está vazio?  
     * @return TRUE se está vazio, FALSE c.c.  
    */
```

```
/*
/*@ pure @*/ boolean isEmpty();
/**
 * Devolver a cardinalidade do conjunto
 * @return o número de elementos no conjunto
 */
//@ ensures isEmpty() ==> \result == 0;
//@ ensures !isEmpty() ==> \result > 0;
/*@ pure @*/ int length();
/**
 * O elemento pertence ao conjunto?
 * @param item A referência do elemento a procurar
 * @return TRUE se pertence, FALSE c.c.
 */
//@ requires item != null;
/*@ pure @*/ boolean belongs(Object item);
/**
 * Inserir o item no conjunto sem duplicações
 * @param item A referência do elemento a inserir
 */
/*@ requires item != null;
@ ensures belongs(item);
@ ensures !\old(belongs(item)) ==>
@           length() == \old(length()) + 1;
@ ensures \old(belongs(item)) ==>
@           equals(\old(clone()));
*/
void insert(Object item);
```

A terceira pós-condição indica que o conjunto não conta com as duplicações. Se o elemento pertence ao conjunto, o conjunto não é modificado. Se não pertencer (a segunda pós-condição) o número de elementos deve aumentar numa unidade (porque o elemento foi inserido).

```
/**
 * Remover o item do conjunto
 * @param item A referência do elemento a remover
 */
/*@ requires item != null;
@ ensures \old(belongs(item)) ==> !belongs(item);
@ ensures !\old(belongs(item)) ==>
@           equals(\old(clone()));
*/
void remove(Object item);
```

A remoção possui pós-condições semelhantes à inserção. Se o elemento pertencer ao conjunto antes da execução do método, depois da execução já não pertence. Se o elemento não pertencer ao conjunto, o conjunto não é alterado.

Infelizmente, não existe uma forma directa de expressar o comportamento das operações sobre conjuntos. O JML permite definir métodos internos que só são usados nos contratos, aumentando a expressividade da linguagem. A título de exemplo (não usaremos esta funcionalidade) podemos observar como se faria a pós-condição para a união de conjuntos:

```

/**
 * Inserir os elementos do conjunto 's'
 * @param s A referência do conjunto a unir
 */
//@ requires s != null;
/*@ public pure model boolean haveAll(Set s) {
    @ for (Iterator item = s.iterator(); item.hasNext(); )
    @   if (!belongs(item.next()))
    @     return false;
    @   return true;
    @ }
    @ ensures haveAll(s);
    @*/
void union(Set s);

/**
 * Remover os elementos que não pertencem a 's'
 * @param s A referência do conjunto a intersecutar
 */
//@ requires s != null;
void intersection(Set s);

/**
 * Remover os elementos que pertencem a 's'
 * @param s A referência do conjunto a subtrair
 */
//@ requires s != null;
void subtraction(Set s);

/**
 * O conjunto 's' está contido neste conjunto?
 * @param s O conjunto que está ou não contido
 * @return TRUE se estiver contido, FALSE c.c.
 */
//@ requires s != null;
/*@ pure @*/ boolean contains(Set s);

/**
 * É o conjunto 's' igual a este?

```

```

    * @param s O conjunto a ser comparado
    * @return TRUE se forem iguais, FALSE c.c.
    */
    // @ also
    // @ requires s != null;
    // @ ensures \result == contains((Set)s) &&
    //           ((Set)s).contains(this);
    /*@ pure @*/ boolean equals(Object s);
    /**
     * @return Um iterator para os elementos do conjunto
     */
    Iterator iterator();
}

```

Esta última assinatura exige que a classe que implemente a interface seja capaz de criar um objecto iterador.

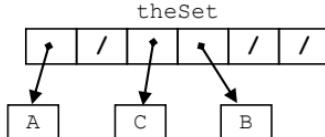
```

    /**
     * Devolver uma cópia da estrutura.
     * @return uma referencia para a cópia
     */
    // @ also
    // @ ensures (\result != this) && equals(\result);
    /*@ pure @*/ Object clone();
} // endInterface Set

```

### 16.3 Uma Implementação Estática

Optou-se por uma representação estática do tipo conjunto. As referências dos objectos são armazenadas num vector designado `theSet`. Quando uma posição do vector armazena a referência nula, essa posição está desocupada. No seguinte exemplo, o vector representa o conjunto {A,B,C}.



Existem ainda um atributo que guarda o número de elementos actual do conjunto.

```

package dataStructures;
/**
 * <p>Titolo: Uma implementação vectorial do Conjunto</p>
 * <p>Descrição: Esta implementação usa uma representação
 *               estática para guardar a informação</p>
 */

```

```

public class VSet implements Set,Cloneable {
    private final int DELTA = 128;
    private int nElems;
    private Object[] theSet;

    public VSet() {
        theSet = new Object[DELTA];
        nElems = 0;
    }

    public Set empty() {
        return new VSet();
    }

    public boolean isEmpty() {
        return nElems == 0;
    }

    public int length() {
        return nElems;
    }

    public boolean belongs(Object item) {
        for (int i=0; i<theSet.length; i++)
            if (isInSet(i) && theSet[i].equals(item))
                return true;
        return false;
    }
}

```

A ordem dos elementos de um conjunto não é relevante, por exemplo o conjunto {1,3,5} é igual ao conjunto {3,1,5}. Já foi referido que uma interrogação não deve produzir efeitos secundários (deve responder à pergunta sem alterar o estado do objecto). Porém, o estado a que a frase anterior se refere é o estado do objecto do ponto de vista do TDA (ou seja, a **representação abstracta** do objecto) e não os valores da configuração da classe (a **representação concreta**). Por exemplo, se quisermos trocar a ordem dos elementos do vector theSet, colocando o objecto mais recentemente pesquisado no início do vector (para melhorar o desempenho de futuras pesquisas), estariamos somente a provocar um **efeito secundário benevolente**. O valor do objecto (neste caso, o conjunto que representa) não foi afectado. Resumindo, as interrogações podem alterar a representação concreta do objecto desde que não modifiquem a representação abstracta do mesmo.

```

private boolean isInSet(int n) {
    return theSet[n] != null;
}

```

O método privado `isInSet()` verifica se uma dada posição do vector está disponível.

```
private void grow() {
    Object[] newSet =
        new Object[theSet.length + DELTA];
    for(int i=0;i<theSet.length;i++)
        newSet[i] = theSet[i];
    theSet = newSet;
}
```

O método `grow()` cria um vector maior e actualiza-o com o conteúdo do vector original.

```
public void insert(Object item) {
    if (belongs(item))
        return;
    if (nElems == theSet.length)
        grow();
    int i;
    for (i=0; isInSet(i); i++) //ciclo vazio
        theSet[i] = item;
    nElems++;
}
```

O método `insert()` verifica se o elemento já existe no conjunto (nesse caso, o conjunto não é alterado). Caso contrário, procura-se um índice disponível e armazena-se a referência do objecto nesse índice.

```
public void remove(Object item) {
    for (int i=0; i<theSet.length; i++)
        if (isInSet(i) && theSet[i].equals(item)) {
            theSet[i] = null;
            nElems--;
            return;
        }
}
```

Na remoção não se admite como pré-condição que o elemento deva existir. O método está construído de modo a que se o elemento não for encontrado no vector (não pertence ao conjunto) nada é alterado.

A implementação dos cinco métodos seguintes depara-se com um problema: o argumento recebido é um objecto do tipo `Set`, não `DSet`, para o qual não existem operações básicas que permitam aceder aos elementos que contém. É neste ponto que o processo de iteração é necessário.

```
public void union(Set s) {
    Iterator it = s.iterator();
    while (it.hasNext())
        insert(it.next());
}
```

Observamos a capacidade de iteração em funcionamento: para unir os dois conjuntos são percorridos os elementos do conjunto *s* para os inserir no conjunto actual.

```
public void intersection(Set s) {
    Iterator it = iterator();
    while (it.hasNext()) {
        Object elem = it.next();
        if (!s.belongs(elem))
            remove(elem);
    }
}
```

A intersecção é semelhante à união: todos os elementos que não pertençam igualmente a *s* são removidos do conjunto. É necessário alguma atenção quando se remove elementos e se itera ao mesmo tempo. Eventualmente a alteração da estrutura tem consequências no processo de iteração.

```
public void subtraction(Set s) {
    Iterator it = s.iterator();
    while (it.hasNext())
        remove(it.next());
}
```

No método da subtracção remove-se todos os elementos que pertençam a *s*.

```
public boolean contains(Set s) {
    Iterator it = s.iterator();
    while (it.hasNext())
        if (!belongs(it.next()))
            return false;
    return true;
}
```

No método `contains()` qualquer elemento do conjunto *s* que não pertença ao conjunto determina de imediato a resposta à questão (é falso). Se todos os elementos de *s* pertencerem ao conjunto, o ciclo termina e a resposta é verdadeira.

```
public boolean equals(Object s) {  
    if (!(s instanceof Set))  
        return false;  
    return contains((Set)s) && ((Set)s).contains(this);  
}
```

O método `equals()` utiliza a definição de igualdade entre conjuntos.

```
public Object clone() {  
    VSet cp = new VSet();  
    cp.theSet = new Object[theSet.length];  
    for (int i=0; i< theSet.length; i++)  
        if (isInSet(i))  
            cp.insert(theSet[i]);  
    return cp;  
}  
  
public String toString() {  
    if (isEmpty())  
        return "{}";  
    Iterator it = iterator();  
    StringBuffer result =  
        new StringBuffer("{" + it.next());  
  
    while (it.hasNext())  
        result.append(", " + it.next());  
  
    return result.toString() + "}";  
}
```

Resta agora implementar a capacidade de iterar. Neste exemplo, define-se uma classe interior `SetIterator` que implementa as funcionalidades do iterador. O método `iterator()` exigido pela interface `Set` apenas cria e devolve um objecto dessa classe.

```
public Iterator iterator() {  
    return new SetIterator();  
}
```

Segue a definição da classe `SetIterator`:

```
private class SetIterator implements Iterator {  
    private int nextElem;
```

O atributo `nextElem` servirá para indicar a posição actual do iterador e para determinar se ainda resta algum elemento a iterar na estrutura de dados. Se o conjunto estiver vazio, o valor do atributo será igual a `-1`.

```
private SetIterator() {
    nextElem = isEmpty() ? -1 : 0;
}

/* Devolve o próximo elemento, se não existir devolve NULL
 */
public Object next() {
    if (nextElem >= 0)
        for (int i=nextElem; i< theSet.length; i++)
            if (isInSet(i)) {
                nextElem = i+1;
                return theSet[i];
            }
    nextElem = -1;
    return null;
}
```

O método `next()` verifica se ainda existem posições para procurar no vector. Se não (`nextElem` igual a `-1`), termina devolvendo a referência nula. Caso contrário, encontra a referência do próximo elemento e devolve-a.

```
/* Verifica se há um próximo elemento
 */
public boolean hasNext() {
    if (nextElem != -1)
        for (int i=nextElem; i< theSet.length; i++)
            if (isInSet(i))
                return true;
    return false;
}
```

O método `hasNext()` é a guarda dos ciclos que utilizam a capacidade de iteração para percorrer os elementos da estrutura.

```
public void remove() {
    throw new UnsupportedOperationException();
}

} // endInnerClass SetIterator
```

O método `remove()` não foi implementado. No entanto, dado que a classe implementa a interface é necessário incluir a sua definição que se resume a lançar uma exceção não verificável caso o método seja invocado.

```
} // endClass VSet
```

## Usar a classe `VSet`

Segue-se um uso dos métodos desta classe:

```
public static void main(String[] args) {  
    VSet v1 = new VSet(),  
        v2 = new VSet();  
  
    v1.insert(new Integer(1));  
    v1.insert(new Integer(2));  
    v1.insert(new Integer(3));  
    v1.insert(new Integer(4));  
    v1.insert(new Integer(2));  
  
    v1.remove(new Integer(3));  
    v1.remove(new Integer(10));  
  
    System.out.println(v1);  
  
    v2.insert(new Integer(30));  
    v2.insert(new Integer(20));  
    v2.insert(new Integer(1));  
  
    v1.subtraction(v2);  
    v2.union(v1);  
  
    System.out.println(v1+" length="+v1.length());  
    System.out.println(v2+" length="+v2.length());  
}  
□ {1,2,4}←  
{2,4} length=2←  
{30,20,1,2,4} length=5←
```

---

## Exercícios

1. Implemente `Set` numa classe com representação dinâmica.
2. Estenda a especificação do TDA conjunto para incluir as operações: (i) `complement`, que dado dois conjuntos devolve o complemento do primeiro em relação ao segundo; (ii) `stack2set`, que dado uma pilha devolve o conjunto constituído por todos os elementos da pilha; (iii) `queue2set`, que dado uma fila devolve o conjunto constituído por todos os elementos da fila;.

3. Estenda a interface `Set` e a implementação `VSet` tendo em conta o exercício anterior.
4. Determine a complexidade temporal dos métodos apresentados na classe `VSet`.
5. Implemente uma classe com os métodos `set2stack` e `set2queue`. Pode utilizar todos os métodos descritos.
6. Introduza para as listas, o mecanismo de iteração apresentado neste capítulo.
7. Um saco é uma coleção com eventuais repetições de objectos. Realize uma especificação do saco (TDA `bag`) que inclua as operações: (i) `empty`, que cria um saco vazio; (ii) `insert`, que insere um novo elemento; (iii) `isEmpty`, que verifica se o saco está vazio; (iv) `remove`, que remove um elemento; (v) `length`, que devolve o número de elementos; (vi) `howMany`, que devolve o número de ocorrências de um elemento no saco; (vii) `equals`, verifica se dois sacos são iguais; (viii) `add`, que dado dois sacos devolve um saco com os elementos dos dois primeiros; (ix) `subtract`, que dado dois sacos, devolve um saco com os elementos do primeiro aos quais se retirou os elementos do segundo.
8. Implemente a interface `Bag` como sendo o desenho da especificação anterior.
9. Implemente a classe `DBag` que implemente a interface `Bag` através de uma representação dinâmica.
10. Um saco com recibos é uma variante do tipo anterior. Quando é inserido um elemento, o método devolve uma *string* denominada por *recibo*. Para remover um elemento é necessário fornecer o respectivo *recibo*. Especifique, escreva a interface e uma implementação dinâmica deste tipo.
11. Uma relação entre dois conjuntos A e B é um subconjunto de  $A \times B$ . Realize uma especificação para a relação (defina o TDA `relation`) que inclua as operações: (i) `empty`, que cria uma relação vazia; (ii) `insert`, que insere um novo par numa relação; (iii) `isEmpty`, que verifica se a relação está vazia; (iv) `remove`, que remove um dado par de uma relação; (v) `length`, que devolve o número de pares de uma relação; (vi) `belongs`, que verifica se um dado par pertence a uma relação; (vii) `contains`, se uma relação contém outra relação; (viii) `equals`, se duas relações são iguais. Especifique ainda as propriedades: se a relação é reflexiva, simétrica, transitiva, anti-simétrica e se é ordem parcial (estude estas definições num manual que introduza as relações matemáticas).
12. Implemente a interface `Relation` como sendo o desenho da especificação anterior.
13. Implemente a classe `DRelation` que implemente a interface `Relation` através de uma representação dinâmica.



# 17 – Árvores

---

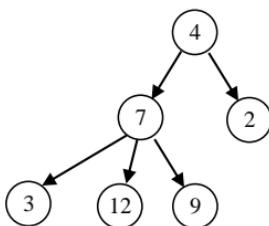
As listas apresentadas no capítulo 14, apesar de flexíveis, possuem uma série restrição: os elementos são acedidos sequencialmente. Para aceder ao  $n$ -ésimo elemento é necessário percorrer os  $n-1$  elementos anteriores. A complexidade temporal de acesso a uma lista é  $O(n)$ .

As árvores são **estruturas de dados não lineares**. Cada elemento da árvore tem zero, um ou mais elementos seguintes. Não existe uma única forma de percorrer uma árvore. Pode-se definir uma árvore recursivamente como:

- Uma estrutura vazia (diz-se uma árvore vazia).
- Um **nó** (que contém a informação a armazenar) designado **raiz** (do inglês, *root*) e um número finito de árvores (designadas **subárvores**).

Esta definição abrange a definição das listas (o número de árvores associadas é igual a 1) de onde se conclui que as listas são casos particulares de árvores, podendo ser vistas como **árvores degeneradas** (do inglês, *degenerate tree*).

Graficamente, essa associação é descrita por uma seta que liga a árvore a uma subárvore. O próximo exemplo mostra uma árvore que armazena seis inteiros.



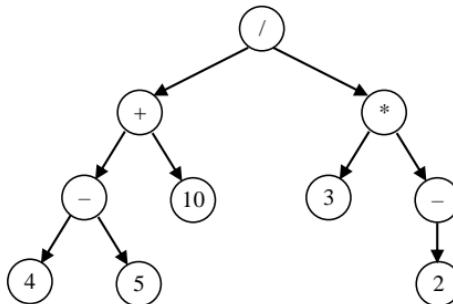
Alguma terminologia:

- Cada nó é **pai** dos nós incluídos nas respectivas subárvores, sendo estes os **filhos** do primeiro (no exemplo o nó 7 é pai de 3, 12 e 9, sendo estes os filhos de 7).
- Os **descendentes** (do inglês, *descendant*) de um nó são os seus filhos mais os descendentes desses filhos.
- Os **ascendentes** (do inglês, *ancestors*) de um nó são o conjunto dos nós para o qual este nó é descendente.
- Uma **folha** (do inglês, *leaf*) é um nó sem filhos (os nós 3, 12, 9 e 2 são todos folhas).
- Um **nó interior** é um nó que não é raiz nem folha.
- Um **caminho** (do inglês, *path*) é uma sequência de nós desde a raiz até uma das folhas (a sequência 4, 2 é um caminho, enquanto a sequência 4, 7 não é).
- A **altura** (do inglês, *height*) ou **profundidade** (do inglês, *depth*) de uma árvore é a dimensão do maior caminho (a árvore do exemplo tem altura três).
- O **nível** de um nó (do inglês, *level*) é igual à dimensão da sequência de nós desde a raiz até ao próprio nó. A raiz tem nível um. No exemplo, o nó 4 tem nível um, os nós 7 e 2 têm nível dois e os restantes nível três.
- O **grau** (do inglês, *degree*) ou a **aridade** (do inglês, *arity*) de um nó é igual ao seu número de filhos (o nó 4 tem grau dois, o nó 7 tem grau três, os nós 2, 3, 9 e 12 têm grau zero). O grau, ou aridade, de uma árvore é igual ao maior grau dos seus nós (a árvore anterior possui grau três).
- Uma **árvore binária** (do inglês, *binary tree*) é uma árvore de grau dois. Uma árvore binária contém exatamente duas subárvores (que podem ser vazias) designadas **subárvore da esquerda** e a **subárvore da direita**.

## 17.1 Árvores Binárias

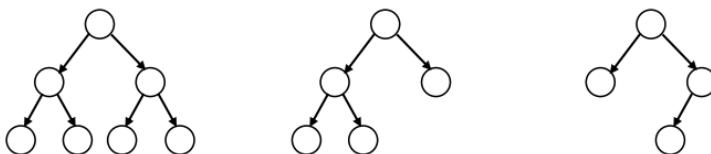
O estudo específico das árvores binárias é interessante porque: (i) são árvores com estruturas simples quando comparadas com árvores de maior grau e (ii) qualquer árvore pode ser representada por uma árvore binária (como?).

Um exemplo de utilização deste tipo de árvores é a representação de expressões aritméticas arbitrárias (os operadores aritméticos são unários ou binários). A seguinte árvore representa a expressão  $((4-5)+10)/(3*-2)$ :



Algumas definições sobre árvores binárias:

- Uma árvore binária está **equilibrada** se a diferença das alturas das subárvores não é superior a 1 e todas as subárvores estão equilibradas.
- Uma árvore binária está **cheia** se for vazia ou se as duas subárvores tiverem a mesma altura e se ambas são cheias. Mostra-se por indução que uma árvore cheia de altura  $h$  possui  $2^h - 1$  nós.
- Uma árvore binária de altura  $h$  está **completa** se estiver cheia até ao nível  $h-1$  e todos os nós do último nível estão o mais à esquerda possível.
- Uma árvore binária é **estritamente binária** se todos os nós possuírem grau 0 ou 2.



Nos diagramas acima, a árvore da esquerda é cheia, completa, equilibrada e estritamente binária. A árvore do meio é completa, equilibrada e estritamente binária. A árvore da direita é equilibrada.

## Percorrer uma Árvore Binária

As árvores não são estruturas lineares. Como atravessar todos os nós de uma árvore? Vamos referir as quatro formas mais naturais de percorrer uma árvore, nomeadamente o percurso prefixo, o percurso infixo, o percurso sufixo e o percurso por largura.

O **percurso prefixo** de uma árvore binária trata o conteúdo do nó actual para, de seguida, percorrer a subárvore esquerda e depois a subárvore direita. O **percurso infixo** percorre primeiro a subárvore esquerda, depois trata o conteúdo e por fim percorre a subárvore

direita. O **percurso sufixo** percorre primeiro as subárvore esquerda e direita, para depois tratar o conteúdo. A diferença entre estes três métodos reside no momento em que o tratamento do conteúdo do nó actual é efectuado (no início para o prefixo, no meio para o infixo, no fim para o sufixo).

Consideremos a árvore da expressão  $((4-5)+10)/(3*-2)$  da página anterior. O percurso dessa árvore consoante o método usado devolveria os resultados:

- Prefixa: / + - 4 5 10 \* 3 - 2
- Infixa: 4 - 5 + 10 / 3 \* - 2
- Sufixa: 4 5 - 10 + 3 2 - \* /

A notação matemática para representar expressões, tais como  $((4-5)+10)/(3*-2)$ , é uma notação infixa que utiliza parênteses para forçar precedências.

Estes três percursos são designados por **percursos em profundidade**, devido ao método de descer às folhas da árvore para depois, por retrocesso, visitar os nós ainda não percorridos. Um outro modo designa-se por **percurso em largura**, na qual é visitada a raiz, depois os nós de nível 2 (digamos, da esquerda para a direita), em seguida os de nível 3 e assim por diante. No exemplo anterior, o percurso em largura dos nós seria:

- Largura: / + \* - 10 3 - 4 5 2

Este tipo de percurso possui uma desvantagem. Enquanto que nos percursos em profundidade basta saber qual o caminho actual e qual a próxima subárvore a percorrer, no percurso em largura é necessário armazenar todos os nós do nível actual de forma a percorrer os nós do próximo nível. Para árvores de dimensão elevada esta desvantagem transforma-se num ónus muito importante dado que o número de nós a armazenar cresce de forma exponencial.

## A Especificação da Árvore Binária

A representação de uma árvore binária é definida recursivamente a partir de duas operações: `empty` que representa a árvore vazia e `constr` que representa uma árvore construída por uma informação (como sempre, designada por `Element`) e por duas árvores. Estas duas operações são os construtores do tipo árvore. As outras operações definidas sobre este TDA são:

- `isEmpty` – verifica se a árvore está vazia
- `root` – acede à informação do nó
- `left` – acede à subárvore da esquerda
- `right` – acede à subárvore da direita
- `isLeaf` – verifica se a árvore é uma folha
- `length` – devolve o número de nós da árvore
- `height` – devolve a altura da árvore

- equals – verifica se duas árvores são iguais

Segue o Tipo de Dados Abstracto:

```
especificação BinTree<Element> =
importa Integer, Boolean
géneros BinTree
operações

    empty : → BinTree
    constr : BinTree BinTree Element → BinTree
    isEmpty : BinTree → Boolean
    root : BinTree ↗ Element
    left : BinTree ↗ BinTree
    right : BinTree ↗ BinTree
    isLeaf : BinTree → Boolean
    length : BinTree → Integer
    height : BinTree → Integer
    equals : BinTree BinTree → Boolean

axiomas

    isEmpty(empty) = TRUE
    isEmpty(constr(TL,TR,I)) = FALSE
    root(constr(TL,TR,I)) = I
    left(constr(TL,TR,I)) = TL
    right(constr(TL,TR,I)) = TR
    isLeaf(T) =
        not isEmpty(T) and
        isEmpty(left(T)) and isEmpty(right(T))
    length(T) =
        if isEmpty(T)
        then 0
        else 1 + length(left(T)) + length(right(T))
    height(T) =
        if isEmpty(T)
        then 0
        else 1 + max(height(left(T)),height(right(T)))
    equals(T1,T2) =
        if isEmpty(T1) or isEmpty(T2)
```

```
then isEmpty(T1) and isEmpty(T2)
else equals(root(T1), root(T2)) and
    equals(left(T1), left(T2)) and
    equals(right(T1), right(T2))
```

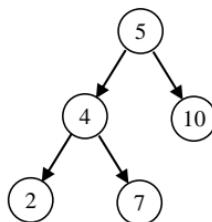
**pré-condições**

```
root(T)   requer not isEmpty(T)
left(T)   requer not isEmpty(T)
right(T)  requer not isEmpty(T)
```

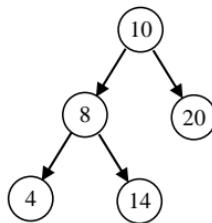
**fim-especificação**

## A Interface Árvore Binária

Vamos adicionar três novas operações que permitem percorrer a árvore binária de forma prefixa, infixa e sufixa. Porquê? Para poder aplicar operações sobre os nós da árvore. Por exemplo, seja a seguinte árvore:



Seria interessante se fosse possível modificar cada um dos nós de forma individual. Por exemplo, para duplicar o valor de cada nó da árvore bastaria executar um dos percursos referidos atrás e aplicar a operação dobro a cada valor. Obter-se-ia assim, a árvore:



Para este efeito, vamos definir um novo tipo denominado `Visitor` relacionado com as futuras funções de percurso da árvore.

```
package dataStructures;
public interface Visitor {
    /**
     * Executar a operação
     * @param info A referência da informação a ser tratada
     */
    void visit(Object info);
    /**
     * Devolver o resultado
     * @return O resultado da(s) visita(s)
     */
    Object result();
} // endInterface Visitor
```

A classe que implementa este operador define a operação a aplicar sobre a árvore (o somatório dos nós, a duplicação dos nós, etc.). Veremos um exemplo na secção seguinte.

Considere a seguinte interface do TDA Árvore Binária:

```
package dataStructures;
import java.util.*;

/**
 * <p>Titolo: A interface do TDA Árvore Binária</p>
 * <p>Descrição: O desenho Árvore Binária com contratos</p>
 */

public interface BinTree {
    //@ public invariant !isEmpty() ==> root() != null;
    /**
     * Criar uma Árvore vazia
     * @return uma Árvore vazia
     */
    //@ ensures \result.isEmpty();
    /*@ pure */ BinTree empty();
    /**
     * A árvore está vazia?
     * @return TRUE se está vezia, FALSE c.c.
     */
    /*@ pure */ boolean isEmpty();
    /**
     * Construir uma árvore
     * @param item A referência do objecto a ser guardado
     * @param left A referência da árvore da esquerda
     * @param right A referência da árvore da direita
```

```
/*
 *@ requires right != null && !right.isEmpty() ==>
 *      item.getClass() == right.root().getClass();
 @ requires left != null && !left.isEmpty() ==>
 *      item.getClass() == left.root().getClass();
 @ ensures item != null ==> !isEmpty();
 @ ensures item != null ==> root().equals(item);
 @ ensures item != null ==> left().equals(left);
 @ ensures item != null ==> right().equals(right);
 */
void constr(Object item, BinTree left, BinTree right);
```

As pré-condições do método `constr()` garantem que a árvore armazena elementos do mesmo tipo. As quatro pós-condições dizem respeito aos axiomas das interrogações `cons`, `root`, `left` e `right` no TDA. É colocada uma guarda dado que este método pode ser invocado com referências nulas.

```
/**
 * Devolver a informação da raiz da árvore
 * @return A referência do objecto situado na raiz
 */
//@ requires !isEmpty();
/*@ pure @*/ Object root();

/**
 * Devolver a árvore da esquerda
 * @return A referência da subárvore da esquerda
 */
//@ requires !isEmpty();
/*@ pure @*/ BinTree left();

/**
 * Devolver a árvore da direita
 * @return A referência da subárvore da direita
 */
//@ requires !isEmpty();
/*@ pure @*/ BinTree right();

/**
 * Devolver o número de elementos da árvore
 * @return O número de elementos da árvore
 */
/*@ ensures isEmpty() ==> \result == 0;
 @ ensures !isEmpty() ==>
 *      \result == 1+left().length()+right().length();
 */
/*@ pure @*/ int length();
```

A pós-condição do método `length()` é muito semelhante ao respectivo axioma do TDA, o que reflecte a noção recursiva da definição de árvore. O mesmo se passa nos métodos seguintes:

```
/**
 * Devolver a altura da árvore (ie, a dim. do maior caminho)
 * @return A altura da árvore
 */
/*@ ensures isEmpty() ==> \result == 0;
 * ensures !isEmpty() ==> \result == 1 +
 * @           (left().height() > right().height() ?
 * @           left().height() : right().height());
 */
/*@ pure @*/ int height();

/**
 * É uma folha?
 * @return TRUE se é uma folha, FALSE c.c.
 */
/*@ ensures \result == (!isEmpty()
 * @           && left().isEmpty()
 * @           && right().isEmpty());
 */
/*@ pure @*/ boolean isLeaf();

/**
 * Verificar se as árvores são iguais
 * @param t A árvore a ser comparada
 * @return TRUE se forem iguais, FALSE c.c.
 */
/*@ also
 * requires t != null;
 * ensures isEmpty() <==> ((BinTree)t).isEmpty();
 * ensures !isEmpty() && ! ((BinTree)t).isEmpty() ==>
 * \result == (root().equals(((BinTree)t).root()) &&
 *           left().equals(((BinTree)t).left()) &&
 *           right().equals(((BinTree)t).right()));
 */
/*@ pure @*/ boolean equals(Object t);
```

Os próximos três métodos recebem um objecto do tipo `Visitor` e aplicam-no a cada nó da árvore.

```
/**
 * Percorrer a árvore de forma prefixa
 * @param op O operador a ser executado em cada nó
 */
```

```

//@ requires op != null;
void prefix(Visitor op);
/**
 * Percorrer a árvore de forma infixa
 * @param op O operador a ser executado em cada nó
 */
//@ requires op != null;
void infix(Visitor op);
/**
 * Percorrer a árvore de forma sufixa
 * @param op O operador a ser executado em cada nó
 */
//@ requires op != null;
void sufix(Visitor op);
/**
 * @return Um iterator para os elementos da árvore
 */
Iterator iterator();

```

Como nos conjuntos é disponibilizado um iterador sobre os elementos da árvore. A forma de implementar o iterador (que tipo de percurso escolher) dependerá da implementação.

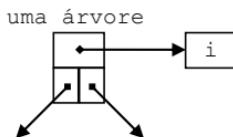
```

/**
 * Devolver uma cópia da árvore.
 * @return uma referencia para a cópia
 */
//@ also
//@ ensures (\result != this) && equals(\result);
/*@ pure */ Object clone();
} // endInterface BinTree

```

## Uma Implementação Dinâmica

Este tipo de estrutura é apropriado para uma representação dinâmica. A partir da definição, uma árvore binária consiste num conteúdo e duas referências: uma para a subárvore esquerda e outra para a subárvore direita. Graficamente:

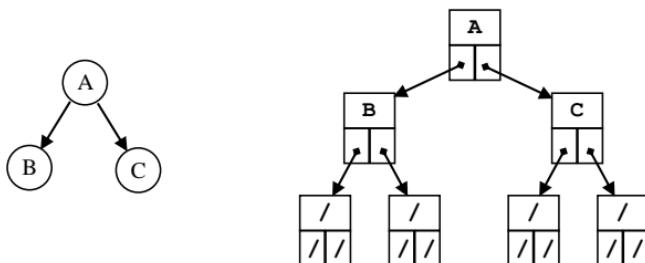


Na implementação da lista, esta é representada por uma referência para um nó (que por sua vez referencia o seguinte e assim por diante). Dessa forma a implementação não reflecte exactamente a definição recursiva da lista (uma lista é um conteúdo e uma lista).

A abordagem que descrevemos para as árvores é conforme a definição recursiva apresentada. A árvore é um nó explícito, não uma referência para um nó. As referências que contém são referências para outras árvores. O problema deste tipo de implementação é como representar a árvore vazia (no caso anterior, a lista vazia era identificada pela referência nula). Vamos considerar que a árvore vazia é representada por um nó onde todas as referências são nulas:



Convencionamos que uma folha é uma árvore que referencia duas árvores vazias (em vez de duas referências nulas). Esta abordagem tem a desvantagem de gastar mais memória, mas facilita a construção dos métodos sobre a estrutura da árvore. No seguinte exemplo, observamos a estrutura em memória (à direita) do respectivo diagrama (à esquerda):



```

package dataStructures;
import java.util.*;
/** Titulo: Uma implem. dinâmica da Árvore Binária */
public class DBinTree implements BinTree, Cloneable {
    private DBinTree leftTree, rightTree;
    private Object infoTree;
  
```

A configuração referida atrás: duas referências para duas subárvores e a referência para a informação armazenada.

```

public DBinTree() {
    infoTree = null;
    leftTree = rightTree = null;
}
  
```

```
public DBinTree(Object o, BinTree left,
                BinTree right) {
    constr(o, left, right);
}

public BinTree empty() {
    return new DBinTree();
}

public void constr(Object item, BinTree left,
                    BinTree right) {
    infoTree = item;
    leftTree = (DBinTree)left;
    rightTree = (DBinTree)right;
}
```

O segundo construtor da classe (com três parâmetros) permite construir árvores a partir de árvores já existentes.

```
public boolean isEmpty() {
    return infoTree == null;
}
```

Seguindo o convencionado, uma árvore está vazia se o conteúdo for nulo. Não invalida a invariante da interface dado que só exige que a referência do conteúdo seja não nula no caso da árvore não estar vazia.

```
public Object root() {
    return infoTree;
}

public BinTree left() {
    return leftTree;
}

public BinTree right() {
    return rightTree;
}

public int length() {
    if (isEmpty())
        return 0;
    return 1 + left().length() + right().length();
}
```

O cálculo da dimensão da árvore é traduzido quase directamente da própria especificação, resultado do carácter recursivo deste tipo de dados e da operação. Não é necessário testar se existem as subárvores esquerda e direita antes de invocar novamente o método, dada a

existência dos referidos nós vazios. Como cada nó é percorrido uma e uma só vez, a complexidade temporal deste algoritmo é  $\Theta(n)$ .

```
public int height() {  
    if (isEmpty())  
        return 0;  
  
    return 1 + Math.max(right().height(),  
                        left().height());  
}
```

O algoritmo para determinar a altura da árvore é muito semelhante ao especificado. Como no método anterior, a complexidade temporal deste algoritmo é  $\Theta(n)$ . Podemos comparar este resultado com a altura de uma árvore com  $n$  nós limitada inferiormente por  $\Omega(\log(n))$  (uma árvore equilibrada) e superiormente por  $O(n)$  (uma árvore degenerada, i.e., uma lista).

```
public boolean isLeaf() {  
    if (isEmpty())  
        return false;  
  
    return left().isEmpty() && right().isEmpty();  
}
```

Como foi referido, para verificar se um nó é uma folha temos de verificar se as duas referências seguintes são vazias.

```
public boolean equals(BinTree t) {  
    if (!(t instanceof BinTree))  
        return false;  
  
    if (isEmpty() && ((BinTree)t).isEmpty())  
        return true;  
  
    if (isEmpty() || ((BinTree)t).isEmpty())  
        return false;  
  
    return root().equals(((BinTree)t).root()) &&  
          left().equals(((BinTree)t).left()) &&  
          right().equals(((BinTree)t).right());  
}
```

A noção de igualdade é recursiva: duas árvores são iguais se ambas são vazias ou ambas possuem o mesmo conteúdo e duas subárvores iguais.

```
public void prefix(Visitor op) {
    if (!isEmpty())
        op.visit(root());
    left().prefix(op);
    right().prefix(op);
}
```

Este é o primeiro método de percurso da árvore. Ao receber a referência para o objecto operador, executa a operação desejada sobre o conteúdo antes de visitar as subárvores. Os dois métodos seguintes apresentam os percursos infixo e sufixo. Como todos os nós são percorridos, uma e uma só vez, a complexidade temporal destes algoritmos é  $\Theta(n)$ .

```
public void infix(Visitor op) {
    if (!isEmpty())
        left().infix(op);
    op.visit(root());
    right().infix(op);
}

public void sufix(Visitor op) {
    if (!isEmpty())
        left().sufix(op);
    right().sufix(op);
    op.visit(root());
}
```

Mas é necessário que estes métodos sejam obrigatoriamente recursivos? O próximo método (que não faz parte da interface) é um exemplo demonstrativo de como se transforma um método recursivo (neste caso, o método `prefix()`) num método iterativo utilizando uma pilha para simular invocações recursivas:

```
public void prefixIterative(Visitor op) {
    if (isEmpty())
        return;
    VStack stack = new VStack();
    stack.push(this);
    while (!stack.isEmpty()) {
        DBinTree actual = (DBinTree) stack.top();
        stack.pop();
        op.visit(actual.root());
```

```
    if (!actual.right().isEmpty())
        stack.push(actual.right());
    if (!actual.left().isEmpty())
        stack.push(actual.left());
}
}
```

O método `prefixIterative()` começa por inserir a própria árvore no topo da pilha. O ciclo itera enquanto existirem nós por processar. Respeitando a travessia prefixa, é realizada a operação sobre o objecto do topo (sendo retirado da pilha). De seguida, são inseridos na pilha a subárvore direita e depois a subárvore esquerda (desde que não sejam árvores vazias). Como a subárvore esquerda foi a última a ser inserida, será a próxima a processar.

Seguem-se os métodos que criam o iterador da árvore. O critério escolhido foi o percurso em largura, dado que os percursos em profundidade foram utilizados nos métodos anteriores. Deste modo, o primeiro nó é a raiz, depois seguem-se os filhos (os nós de nível 2) da esquerda para a direita, depois os netos (os nós de nível 3) da esquerda para a direita e assim por diante.

```
public Iterator iterator() {
    return new BinTreeIterator(this);
}
```

O método `iterator()` cria um novo objecto da classe interior que inclui as funcionalidades do iterador.

```
private class BinTreeIterator implements Iterator {
    private VQueue qNodes;
```

A fila `qNodes` armazena os nós do nível actual ainda não visitados mais todos os filhos dos nós já visitados deste nível. Cada vez que um nó é visitado (a frente da fila) é retirado e os filhos são colocados no fim da fila.

```
private Object actualObject;
private BinTreeIterator(DBinTree t) {
    actualObject = null;
    qNodes      = new VQueue();
    qNodes.enqueue(t);
}
```

A lista é inicializada com a referência da árvore principal.

```
public boolean hasNext() {
    return !qNodes.isEmpty();
};
```

```
public Object next() {
    if (qNodes.isEmpty())
        actualObject = null;
    return null;
}
DBinTree t = (DBinTree) (qNodes.front());
if (!t.leftTree.isEmpty())
    qNodes.enqueue(t.leftTree);
if (!t.rightTree.isEmpty())
    qNodes.enqueue(t.rightTree);
qNodes.dequeue();
actualObject = t.infoTree;
return actualObject;
};
```

O método `next()` utiliza o algoritmo descrito: vai buscar à frente da lista o próximo nó, actualiza o objecto actual para o conteúdo do nó recolhido e coloca no fim da lista os eventuais filhos desse nó. Se a fila estiver vazia, o método devolve a referência nula.

```
public void remove() {
    throw new UnsupportedOperationException();
}
} // endInnerClass BinTreeIterator
public String toString() {
    return isEmpty() ? "[]" : makeTree(0, this);
}
```

O método `toString()` devolve uma descrição textual da estrutura da árvore. Utiliza os seguintes métodos privados que construem recursivamente uma descrição indentada da árvore.

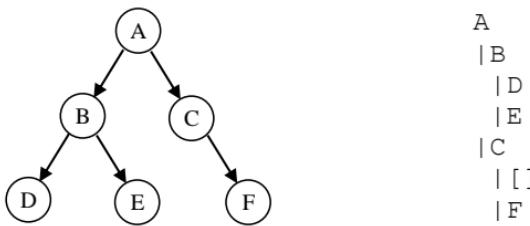
```
private String makeTree(int level, DBinTree T) {
    if (T.isEmpty())                                // base da recursão
        return mark(level) + "[]\n";
    if (T.isLeaf())                                 // base da recursão
        return mark(level) + T.root() + "\n";
                                                // passo da recursão
    return mark(level) + T.root() + "\n" +
           makeTree(level+1, (DBinTree)T.left()) +
           makeTree(level+1, (DBinTree)T.right());
}
```

```

private String mark(int level) {
    String s="";
    // indentar o símbolo '|'
    for (int i=0;i<level-1;i++)
        s += " ";
    return s + ((level>0)? "|" :"");
}

```

Por exemplo, para a próxima árvore da esquerda obteríamos a descrição da direita:



```

public Object clone() {
    if (isEmpty())
        return new DBinTree();
    return new DBinTree(root(),
                        (DBinTree)(left().clone()),
                        (DBinTree)(right().clone()));
}

```

Finalmente, o método `clone()` devolve uma cópia exacta da árvore (o método não copia os objectos armazenados, apenas a estrutura da árvore)

```
} // endClass DBinTree
```

## Usar a interface **Visitor** e a classe **DBinTree**

Vamos definir um operador: concatenar os conteúdos da árvore a percorrer.

```

class Concat implements Visitor {
    private String result = "";
    public void visit(Object info) { // da interface
        result += (String)info;
    }
    public Object result() { // da interface
        return result;
    }
}

```

```
public void reset() {  
    result = "";  
}  
}
```

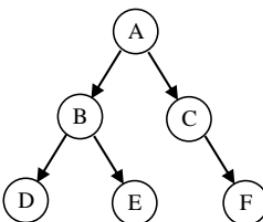
No método `main()` é construído um único nó vazio (referenciada pela variável `tv`) partilhado por todas as folhas poupando-se memória na representação da estrutura de dados. A árvore construída neste exemplo é igual à do diagrama anterior.

```
public static void main(String[] args) {  
    Concat cc = new Concat();  
    DBinTree tv = new DBinTree();  
    DBinTree t1 = new DBinTree("F", tv, tv);  
    DBinTree t2 = new DBinTree("E", tv, tv);  
    DBinTree t3 = new DBinTree("D", tv, tv);  
    DBinTree t4 = new DBinTree("C", tv, t1);  
    DBinTree t5 = new DBinTree("B", t3, t2);  
    DBinTree t6 = new DBinTree("A", t5, t4);  
  
    System.out.println(t6);  
    t6.prefix(cc);  
    System.out.print(cc.result() + " ");  
    cc.reset();  
    t6.infix(cc);  
    System.out.print(cc.result() + " ");  
    cc.reset();  
    t6.sufix(cc);  
    System.out.print(cc.result() + " ");  
    Iterator it = t6.iterator();  
    while (it.hasNext())  
        System.out.print(it.next() + " / ");  
}  
□ A ↲  
| B ↲  
| D ↲  
| E ↲  
| C ↲  
| [] ↲  
| F ↲  
ABDEF DBEACF DEBFCA ABCDEF
```

## Uma Representação Estática

É possível representar vectorialmente o conteúdo e a estrutura de uma árvore. Cada posição do vector é constituída por: (i) uma referência para o conteúdo, (ii) um inteiro que representa o índice no vector da subárvore esquerda e (iii) um inteiro que representa o índice no vector da subárvore direita (se não existir subárvore é igual a -1).

Uma árvore binária e a respectiva representação por vector:



0	A	1	2
1	B	3	4
2	C	-1	5
3	D	-1	-1
4	E	-1	-1
5	F	-1	-1

## Outra Representação Dinâmica

Existe outra forma de representar dinamicamente uma árvore. Da mesma forma que no capítulo das listas usou-se uma classe auxiliar nó para guardar a informação, podemos usar uma classe semelhante para armazenar a informação das árvores. Assim, uma árvore vazia é definida por uma referência nula (da mesma forma que a referência nula representava uma lista vazia). A principal vantagem desta opção é não ser necessário criar objectos vazios para representar árvores vazias (poupa-se memória). Porém, os algoritmos para executar as operações sobre árvores são mais complicados.

## 17.2 Árvores Binárias de Pesquisa

Uma vantagem da estrutura não linear das árvores é na pesquisa de elementos. Se existir um critério que nos permita decidir em qual das duas subárvores procurar, a eficiência deste tipo de pesquisa é muito superior à pesquisa linear característica das listas.

O jogo seguinte ilustra esta questão: Alguém pensa num número entre 1 e 100. Uma outra pessoa tenta descobrir qual o número pensado podendo perguntar se um dado número é maior, menor ou igual que o número inicial. Existem duas estratégias distintas:

- Não utilizar a informação obtida na resposta. Limitar-se a dizer 1, 2, 3, 4, ... até acertar no número. Em média, este método precisa de 50 perguntas.
- Usar a resposta obtida para escolher a próxima pergunta. Neste caso começar com o número que garante a eliminação do maior número de elementos, 49 ou 50. Se escolher 50 e a resposta for “menor” responder com o número entre 1 e 49, ou seja, 24. Senão, com o número entre 51 e 100, ou seja, 75 ou 76. A cada nova pergunta

o espaço de possibilidades é reduzido em cerca de metade. No pior caso são necessárias  $\lceil \log_2(100) \rceil = 7$  questões para adivinhar o número.

O primeiro método descreve uma **pesquisa linear** (do inglês, *linear search*) ou **pesquisa sequencial**, usada normalmente nas listas. O segundo método descreve uma **pesquisa binária** (do inglês, *binary search*) ou **pesquisa dicotómica**.

O próximo método procura sequencialmente por um valor num vector de inteiros (devolve o índice onde o valor se encontra, ou devolve -1 se o valor não existir):

```
public static int serial (int[] v, int value) {
    for (int i=0;i<v.length;i++)
        if (v[i]==value)
            return i;
    return -1;
}
```

O método seguinte resolve o mesmo problema através da pesquisa binária (é no método privado recursivo que se encontra o algoritmo de pesquisa). Este método assume que o vector está ordenado de forma crescente.

```
public static int binary (int[] v, int value) {
    return binary(v,value,0,v.length);
}

private static int binary (int[] v,    int value,
                         int begin, int length) {
    if (length<=0)
        return -1;
    int middle = begin + length/2;
    if (v[middle]==value)
        return middle;
    if (value<v[middle])
        return binary(v,value,begin,length/2);
    return binary(v,value,middle+1,(length-1)/2);
}
```

Se for possível estabelecer um critério semelhante sobre os elementos armazenados numa árvore, pode-se usar uma pesquisa binária. Como a altura mínima de uma árvore binária com  $n$  nós é  $\Theta(\log(n))$  – i.e., uma árvore equilibrada – o melhor desempenho possível de uma procura binária para uma árvore equilibrada é limitado superiormente por  $O(\log(n))$ . Este resultado é melhor que a pesquisa linear de complexidade  $O(n)$ .

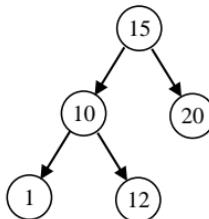
Considere a definição de árvore ordenada:

- Uma árvore binária diz-se **ordenada** (do inglês, *ordered tree*) se existir um critério que ordene os conteúdos dos descendentes de cada nó da árvore e os relate com alguma forma com o conteúdo do nó pai. Este critério estabelece uma ordem total<sup>32</sup> entre os elementos armazenados na árvore.

Considere a definição de árvore binária de pesquisa:

- Uma **árvore binária de pesquisa** (do inglês, *binary search tree*) é uma árvore binária e ordenada pelo seguinte critério: (i) o conteúdo de qualquer nó da subárvore esquerda é menor ou igual que o conteúdo do nó pai, (ii) o conteúdo de qualquer nó da subárvore direita é maior que o conteúdo do nó pai e (iii) ambas as subárvores são árvores binárias de pesquisa.

Um exemplo de árvore binária:



## Especificação da Árvore Binária de Pesquisa

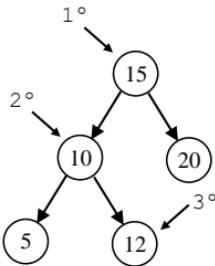
Existem três operações básicas que se podem efectuar sobre uma árvore binária de pesquisa (estes três algoritmos, para uma árvore com  $n$  nós, são  $\Omega(\log(n))$ ):

- Procurar um elemento – começando na raiz e enquanto não encontrar o elemento, escolher qual a subárvore para continuar a pesquisa.
- Inserir um elemento – novamente pela raiz, descer por uma das subárvores até encontrar uma folha. Criar a subárvore adequada e inserir o elemento.
- Remover um elemento – procurar o elemento. Substituir o conteúdo desse nó pelo maior elemento da subárvore esquerda (i.e., o nó mais à direita da subárvore esquerda). Fazer a actualização necessária caso o nó desse maior elemento tenha uma subárvore esquerda.

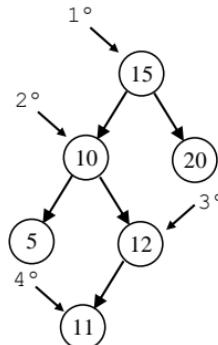
---

<sup>32</sup> Uma *ordem parcial* é uma relação reflexiva, anti-simétrica e transitiva definida a partir do conjunto dos elementos considerados. Um exemplo de ordem parcial é o operador menor ou igual,  $\leq$ , sobre os números inteiros. Uma *ordem total* é uma ordem parcial se para todo o  $X$  e  $Y$  desse conjunto, ou  $X \leq Y$  ou  $Y \leq X$ .

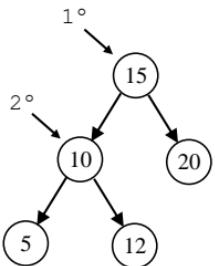
Considerando o diagrama da árvore anterior, considere os seguintes três exemplos: (i) procurar o número 12, (ii) inserir o número 11, (iii) remover o número 10.



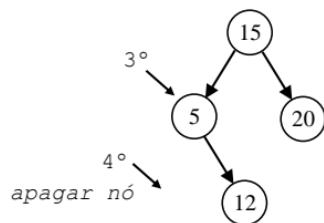
*procurar o número 12*



*inserir o número 11*

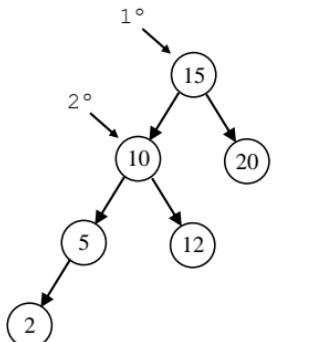


*remover o número 10 (I)*

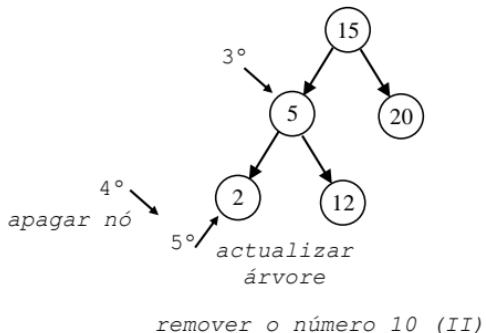


*remover o número 10 (II)*

Outro exemplo de remoção onde o maior elemento da subárvore esquerda (o nó com o valor 5) tem uma subárvore esquerda:



*remover o número 10 (I)*



*remover o número 10 (II)*

Segue a especificação destas operações:

```

especificação SearchBinTree<Element> =
importa BinTree<Element>
géneros SearchBinTree ≤ BinTree
  
```

Esta especificação é uma extensão do TDA árvore binária. O novo tipo que queremos especificar – árvore binária de pesquisa – é um subtipo de árvore binária. A noção de subtipo é representada pelo símbolo  $\leq$  no TDA. Esta noção de subtipo (como referido no capítulo 1) possui características comuns com a noção de herança explicada no capítulo 10. De notar que a operação `constr` não deve ser usada pelos clientes deste tipo (dado poder criar árvores que não são de pesquisa). A única forma de inserir elementos deve ser através da operação `insert`.

#### **operações**

```
isSearchable : BinTree → Boolean
```

Para além das três já referidas, especificamos uma operação que aceita uma árvore binária qualquer (o domínio é `BinTree` e não `SearchBinTree`) e verifica se essa árvore é de pesquisa ou não (i.e., se respeita o critério de ordenação entre os nós pais e os nós filhos).

```

occurs : SearchBinTree Element → Boolean
insert : SearchBinTree Element → SearchBinTree
remove : SearchBinTree Element → SearchBinTree
  
```

**auxiliares**

```
_gt : BinTree Element → Boolean  
_lt : BinTree Element → Boolean  
_rightNode : BinTree ↗ Element  
_remRightNode : BinTree ↗ BinTree
```

Estas quatro funções auxiliares realizam o seguinte: (i) a função `_gt` verifica se um dado elemento é maior ou igual que todos os elementos de uma dada árvore, (ii) a função `_lt` verifica se um dado elemento é menor que todos os elementos de uma dada árvore, (iii) a função `_rightNode` devolve o elemento do nó mais à direita de uma árvore e (iv) a função `_remRightNode` dada uma árvore devolve essa árvore após remover o nó mais à direita. Estas duas últimas funções são usadas na operação remoção: 1º) descobrir o maior elemento menor que o valor a remover (i.e., o nó mais à direita da subárvore esquerda), 2º) substituir o valor a remover por esse elemento e 3º) remover o nó que armazenava esse elemento.

**axiomas**

```
_gt(empty, I) = TRUE  
_gt(constr(TL, TR, J), I) =  
  if isEmpty(TR) then J ≤ I  
  else _gt(TR, I)
```

Representa-se por  $\leq$  a relação de ordem entre os elementos armazenados na árvore. Não confundir com o símbolo usado na secção géneros para definir subtipos.

```
_lt(empty, I) = TRUE  
_lt(constr(TL, TR, J), I) =  
  if isEmpty(TL) then not J ≤ I  
  else _lt(TL, I)
```

As operações `_gt` e `_lt` assumem que a árvore binária é de pesquisa, procurando apenas o nó mais à esquerda e mais à direita respectivamente. Elas são utilizadas na operação seguinte:

```
isSearchable(empty) = TRUE  
isSearchable(constr(TL, TR, J)) = isSearchable(TL)  
                                and isSearchable(TR)  
                                and _gt(TL, J)  
                                and _lt(TR, J)
```

Uma árvore binária é de pesquisa se e só se a raiz for maior que todos os elementos da subárvore esquerda, for menor que os da subárvore direita e ambas as subárvores forem de pesquisa. A base da recursão informa que uma árvore vazia é, por definição, uma árvore de pesquisa.

```

occurs(empty, I) = FALSE
occurs(constr(TL, TR, J), I) = equals(I, J)
                           or (I ≤ J and occurs(TL, I))
                           or ((not I ≤ J) and occurs(TR, I))

```

Descrevemos no axioma `occurs` a pesquisa binária. Se não for igual à raiz, o elemento pertence à árvore desde que se encontre na subárvore esquerda (se for menor) ou na subárvore direita (se for maior).

```

insert(empty, I) = constr(empty, empty, I)
insert(constr(TL, TR, J), I) =
  if I ≤ J
    then constr(insert(TL, I), TR, J)
  else constr(TL, insert(TR, I), J)

```

O elemento `I` deve ser inserido no local correcto, de modo a que a árvore resultante continue a ser de pesquisa. Dessa forma, o passo da recursão determina a inserção à esquerda (se o elemento for menor ou igual que a raiz) ou à direita (se o elemento for maior que a raiz).

```

_rightNode(constr(TL, TR, I)) =
  if isEmpty(TR)
    then I
  else _rightNode(TR)
_remRightNode(constr(TL, TR, I)) =
  if isEmpty(TR)
    then TL
  else constr(TL, _remRightNode(TR), I)

```

Para remover o nó mais à direita é necessário verificar se a subárvore direita não é vazia. Se for, o nó mais à direita é a própria raiz, basta devolver a subárvore esquerda. Se a subárvore direita não for vazia invoca-se a função para essa subárvore e refaz-se a árvore final com esse resultado.

```

remove(empty, I) = empty
remove(constr(TL, TR, J), I) =
  if equals(I, J) then
    if isEmpty(TL)
      then TR
    else if isEmpty(TR)
      then TL
    else
      constr(_remRightNode(TL), TR,
             _rightNode(TL))

```

```
else if I≤J
    then constr(remove(TL,I),TR,J)
    else constr(TL,remove(TR,I),J)
```

A remoção tem duas partes. Se não tiver encontrado o elemento invoca-se recursivamente para a subárvore adequada. Quando encontrar, verifica se alguma das subárvores está vazia (o resultado é imediato). Se ambas as subárvores não forem vazias, executa o método já referido de substituir o valor a remover pelo nó mais à direita, apagando de seguida esse nó.

#### **pré-condições**

```
_rightNode(T)    requer not isEmpty(T)
_remRightNode(T) requer not isEmpty(T)
```

Estas pré-condições não se reflectem na interface porque referem-se a operações auxiliares da especificação.

#### **fim-especificação**

## **A Interface da Árvore Binária de Pesquisa**

```
package dataStructures;
/**
 * <p>Título: A interface da Árvore Binária de Pesquisa</p>
 * <p>Descrição: O desenho (SearchBinTree) com contratos</p>
 */
public interface SearchBinTree extends BinTree {
```

Como especificado, este novo tipo é um subtipo de `BinTree`.

```
 /**
 * A árvore dada é de pesquisa?
 * @return TRUE se t for de pesquisa, FALSE c.c.
 */
//@ requires t != null;
//@ ensures t.isEmpty() ==> \result;
/*@ pure */ boolean isSearchable(BinTree t);
```

A interrogação `isSearchable()` é usada como pré e pós-condição dos três métodos seguintes. A opção de criar uma invariante com esta interrogação não é aconselhável dado que seria necessário resolver o problema do cliente invocar `constr()` do tipo `BinTree`, o que eventualmente resultaria num objecto inválido. Com esta interface, o cliente está impedido de invocar os métodos seguintes se executar `constr()` incorrectamente (pois violaria a pré-condição `isSearchable(this)`).

```

/***
 * O objecto existe na árvore?
 * @param o O objecto a ser procurado
 * @return TRUE se existe, FALSE c.c.
 */
/*@ requires o != null;
@ requires isSearchable(this);
@ ensures isSearchable(this);
@ ensures isEmpty() ==> !\result;
@ ensures !isEmpty() ==> \result == (o.equals(root()) || 
@           (o.compareTo(root())<0 &&
@            ((SearchBinTree)left()).occurs(o)) ||
@           (o.compareTo(root())>0 &&
@            ((SearchBinTree)right()).occurs(o)));
@*/
/*@ pure @*/ boolean occurs(Comparable o);

```

Os objectos armazenados têm de ser comparáveis, i.e., os objectos têm de ser instâncias de classes que implementem a interface Comparable. Esta interface possui o método `compareTo()` que verifica se o objecto é maior, menor ou igual a outro objecto dado. Na especificação, esta característica foi assumida quando os elementos foram comparados com o operador  $\leq$ .

```

/***
 * Inserir um objecto
 * @param o O objecto a ser inserido
 */
//@ requires o != null;
//@ requires !isEmpty() ==> o.getClass()==root().getClass();
//@ requires isSearchable(this);
//@ ensures isSearchable(this);
//@ ensures occurs(o);
//@ ensures length() == \old(length()) + 1;
void insert(Comparable o);

```

A segunda pré-condição determina que os elementos da árvore têm de ser do mesmo tipo. Na terceira pós-condição é possível usar o método `length()` pois esta interface estende a interface BinTree onde `length()` foi declarado.

```

/***
 * Remover um objecto
 * @param o O objecto a ser removido
 */
/*@ requires o != null;
@ requires isSearchable(this);

```

```

@ ensures  isSearchable(this);
@ ensures  \old(isEmpty()) ==> isEmpty();
@ ensures  \old(!isEmpty()) && o.equals(\old(root())) &&
@           \old(left()).isEmpty() ==>
@           equals(\old(right())));
@ ensures  \old(!isEmpty()) && o.equals(\old(root())) &&
@           \old(right()).isEmpty() ==>
@           equals(\old(left())));
@*/
void remove(Comparable o);

} // endInterface SearchBinTree

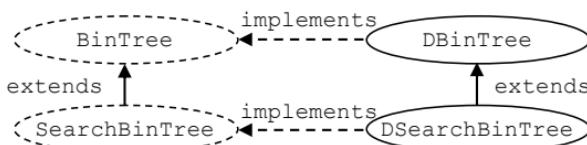
```

Apesar de não ser possível expressar o axioma da operação `remove` no JML (provocaria efeitos secundários) testamos a parte inicial dos condicionais quando o objecto a remover é igual à raiz da árvore.

As funções auxiliares do TDA não são expressas no desenho da interface. Podem servir de inspiração para métodos privados mas não existe qualquer obrigação de os implementar.

## Estender a classe `DBinTree`

Como a interface `SearchBinTree` estende `BinTree`, a nova classe `DsearchBinTree` estende a classe `DBinTree`. Graficamente podemos descrever as relações entre estas quatro entidades da seguinte forma (as elipses pontilhadas representam interfaces):



```

package dataStructures;

/** <p>Titolo: Implementação da Árv. Bin. de Pesquisa</p> */
public class DSearchBinTree extends DBinTree
    implements SearchBinTree {

    public DSearchBinTree() {
        super();
    }

    //@ requires isSearchable(t);
    public DSearchBinTree(BinTree t) {
        constr(t.root(),t.left(),t.right());
    }
}

```

Este construtor utiliza uma árvore binária (desde que seja de pesquisa) para inicializar o objecto. Se a pré-condição for satisfeita, este é um processo de tradução de um objecto da superclasse para um objecto da subclasse (algo muito diferente do *downcasting* de uma referência – conferir a página 195).

```
/**  
 * Verificar se 'o' é < que os valores da árvore  
 * @param t A arvore a procurar  
 * @param o O Objecto a comparar  
 * @return TRUE se 'o' é menor que o mínimo, FALSE c.c.  
 */  
private boolean lt(BinTree t, Comparable o) {  
    if (t.isEmpty())  
        return true;  
  
    return t.left().isEmpty() ?  
        o.compareTo(t.root())<0 : lt(t.left(),o);  
}
```

O método `lt()` é semelhante à operação `_lt` da especificação. O símbolo  $\leq$  foi substituído pelo método `compareTo()` que recebe como argumento um objecto do mesmo tipo e devolve um valor negativo se o objecto for menor que o argumento, positivo se for maior ou zero se for igual.

```
/**  
 * Verificar se 'o' é  $\geq$  que os valores da árvore  
 * @param t A arvore a procurar  
 * @param o O Objecto a comparar  
 * @return TRUE se 'o' é  $\geq$  que o máximo, FALSE c.c.  
 */  
private boolean gt(BinTree t, Comparable o) {  
    if (t.isEmpty())  
        return true;  
  
    return t.right().isEmpty() ?  
        o.compareTo(t.root())>=0 : gt(t.right(),o);  
}  
  
public boolean isSearchable(BinTree t) {  
    if (t.isEmpty())  
        return true;  
  
    if (!(t.root() instanceof Comparable))  
        return false;  
  
    return isSearchable(t.left()) &&  
        isSearchable(t.right()) &&  
        gt(t.left(),(Comparable)t.root()) &&
```

```
        lt(t.right(), (Comparable)t.root()));  
    }  
}
```

Se os objectos armazenados na árvores não forem comparáveis, a árvore não é de pesquisa. O método `isSearchable()` percorre recursivamente as subárvores esquerda e direita para detectar: (i) se o elemento na raiz é maior ou igual que os elementos à esquerda e menor que os da direita, (ii) se as próprias subárvores são de pesquisa.

Existe, ainda para o método `isSearchable()`, uma solução alternativa à apresentada com menor complexidade. Se após uma travessia infixa à árvore, a sequência de visitas apresentar uma ordem crescente, então a árvore é de pesquisa. Deixamos esta solução como exercício.

```
public boolean occurs(Comparable o) {  
    SearchBinTree t = this;  
  
    while (!t.isEmpty())  
        if (t.root().equals(o))  
            return true;  
        else  
            t = (SearchBinTree)(o.compareTo(t.root())<0 ?  
                                t.left() : t.right());  
    return false;  
}
```

À excepção dos casos base (chegou a uma árvore vazia ou encontrou o elemento) o método `occurs()` procura o objecto indo para a esquerda ou para a direita consoante a resposta a uma pergunta semelhante à do jogo do início da secção: o objecto da raiz da árvore actual é maior ou menor que o objecto que se procura?

```
public void insert(Comparable o) {  
    SearchBinTree t = this;  
    while (!t.isEmpty())  
        t = (SearchBinTree)(o.compareTo(t.root())<=0 ?  
                            t.left() : t.right());  
  
    t.constr(o, new DSearchBinTree(),  
             new DSearchBinTree());  
}
```

O método `insert()` percorre o caminho apropriado até encontrar um nó vazio. Actualiza o conteúdo do nó com a referência do objecto a inserir e invoca o método `constr()` (da superclasse) para criar dois novos nós vazios. O nó vazio transforma-se, assim, numa folha.

```
private Object rightNode() {
    return right().isEmpty() ?
        root() :
        ((DSearchBinTree)right()).rightNode();
}

private void removeRightNode() {
    if (right().isEmpty()) {
        if (left().isEmpty())
            constr(null,null,null);
        else
            constr(left().root(),
                  left().left(), left().right());
    }
    else
        ((DSearchBinTree)right()).removeRightNode();
}
```

Acima, estão os dois métodos auxiliares da remoção. A semântica é idêntica à das funções auxiliares especificadas no TDA. Com esses dois métodos é possível implementar a remoção, como se verifica no método seguinte:

```
public void remove(Comparable o) {
    SearchBinTree t = this;
    while (!t.isEmpty())
        if (t.root().equals(o)) {
            if (t.left().isEmpty() && t.right().isEmpty())
                t.constr(null,null,null);
            else if (t.left().isEmpty())
                t.constr(t.right().root(),
                        t.right().left(),t.right().right());
            else if (t.right().isEmpty())
                t.constr(t.left().root(),
                        t.left().left(),t.left().right());
            else {
                t.constr(((DSearchBinTree)t.left()).rightNode(),
                        t.left(),t.right());
                ((DSearchBinTree)t.left()).removeRightNode();
            }
            break;
        }
```

```
    else
        t = (SearchBinTree)(o.compareTo(t.root())<0 ?
                            t.left() : t.right());
}
```

O uso dos métodos `rightNode()` e `removeRightNode()` implica que se executa duas vezes o percurso até nó mais à direita. Para aumentar o desempenho do método, as duas tarefas deveriam ser implementadas ao mesmo tempo.

```
public Object clone() {
    DSearchBinTree result = new DSearchBinTree();
    if (!isEmpty())
        result.constr(root(),
                      (SearchBinTree)left().clone(),
                      (SearchBinTree)right().clone());
    return result;
}
} // endClass DSearchBinTree
```

## Usar a classe *DSearchBinTree*

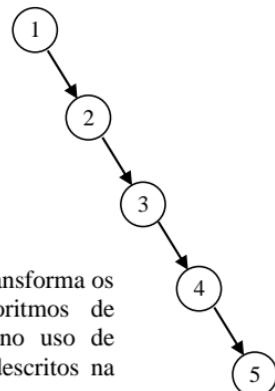
Segue-se um uso dos métodos desta classe. Uma vez mais, utiliza-se a estrutura de dados para armazenar números inteiros.

```
public static void main(String[] args) {
    DSearchBinTree t = new DSearchBinTree();
    t.insert(new Integer(3));    t.insert(new Integer(2));
    t.insert(new Integer(5));    t.insert(new Integer(1));
    t.insert(new Integer(4));    System.out.print(t);
    t.remove(new Integer(5));   t.remove(new Integer(3));
    System.out.print(t);
    System.out.print(t.occurs(new Integer(8))?"y":"n");
    System.out.print(t.isSearchable()?"y":"n");
}
□ 3 ↵
| 2 ↵
| | 1 ↵
| | [] ↵
| 5 ↵
| | 4 ↵
| | [] ↵
2 ↵
| 1 ↵
| 4 ↵
ny
```

## Árvores Equilibradas

As árvores de pesquisa possuem um problema se a ordem de inserção dos elementos não for propícia. Considere este exemplo:

```
DSearchBinTree t = new DSearchBinTree();
t.insert(new Integer(1));
t.insert(new Integer(2));
t.insert(new Integer(3));
t.insert(new Integer(4));
t.insert(new Integer(5));
```



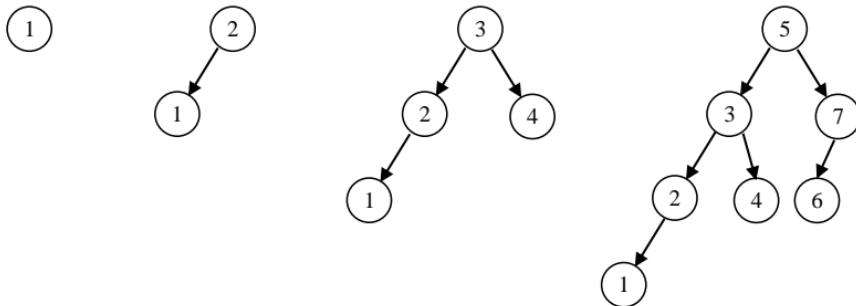
A árvore da direita é produzida pela execução deste código.

No pior caso possível, a árvore degenera numa lista o que transforma os algoritmos de procura, inserção e remoção em algoritmos de complexidade linear, perdendo as potenciais vantagens no uso de árvores. Os algoritmos de inserção, remoção e pesquisa descritos na última secção são  $\Omega(\log(n))$  e  $O(n)$ .

Existe alguma forma de transformar a procura, a inserção e a remoção em algoritmos de complexidade  $O(\log(n))$ ? Em princípio sim, pois a altura mínima de uma árvore binária com  $n$  nós é aproximadamente  $\log(n)$ . Quando é que esta altura mínima ocorre? Quando a árvore está equilibrada. Relembrando: uma árvore binária está equilibrada se a diferença das alturas das subárvore não for superior a 1 e todas as subárvore estão equilibradas.

Uma estrutura de dados concebida em 1962 por Adel'son-Vel'skiĭ e Landis, designada por **árvore AVL**, é uma árvore binária de pesquisa com operações de inserção e remoção cujas execuções mantêm a árvore equilibrada. Esta garantia de equilíbrio faz com que os algoritmos sejam  $O(\log(n))$  (os autores provaram que a altura de uma árvore AVL não excede mais do que em 45% a altura mínima possível).

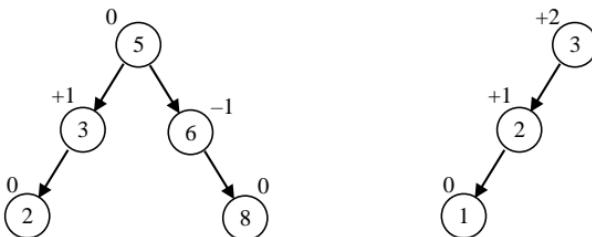
Qual é a “pior” árvore AVL possível com altura  $H$ ? É a árvore AVL com o menor número de nós dessa altura. Seguem as piores árvores AVL de alturas um a quatro:



Estas são as **árvores de Fibonacci** designadas assim devido à forma recursiva como podem ser definidas:

- A árvore vazia é uma árvore de Fibonacci, designada por  $T_0$ , de ordem 0.
- A árvore só com a raiz é uma árvore de Fibonacci, designada por  $T_1$ , de ordem 1.
- Se  $T_{H-1}$  e  $T_{H-2}$  são árvores de Fibonacci de ordem  $H-1$  e  $H-2$ , então  $\text{constr}(T_{H-1}, T_{H-2}, o)$  é uma árvore de Fibonacci, designada por  $T_H$ , de ordem  $H$ .

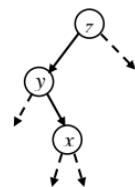
Como funciona a inserção numa árvore AVL? Cada nó numa árvore AVL possui um valor designado por factor de equilíbrio igual à altura da subárvore esquerda subtraída da altura da subárvore direita. Este valor só pode variar entre  $-1$  e  $+1$  se a árvore estiver equilibrada. No exemplo seguinte, a árvore da esquerda está equilibrada enquanto a da direita não está (os números fora de cada nó correspondem ao factor de equilíbrio):

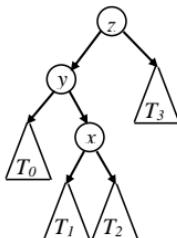


Consideremos a inserção referida nas árvores de pesquisa. Essa inserção pode destruir ou não o equilíbrio da árvore. Na árvore esquerda anterior, a inserção do número 4 resultaria numa árvore ainda equilibrada, enquanto a inserção do número 10 resultaria numa árvore não equilibrada. A inserção em árvores AVL é semelhante à inserção em árvores de pesquisa ao qual se adiciona um algoritmo de reequilíbrio (o mesmo se passará com a remoção).

Que factores de equilíbrio são alterados por uma inserção? Todos os que pertencem ao caminho C constituído pelos nós que vão desde o elemento inserido até à raiz. O algoritmo pode ser descrito da seguinte forma:

- Inserir o elemento como na árvore de pesquisa. Se a árvore está equilibrada terminar. Se não, continuar.
- Seja  $x$  o nó de maior nível de C cujo nó avô tem o factor de equilíbrio diferente de  $-1$ ,  $0$  ou  $+1$ . O nó  $x$  ou é o nó do próprio elemento inserido ou é um dos ascendentes.
- Seja o nó  $y$  o pai de  $x$  e o nó  $z$  o pai de  $y$ .



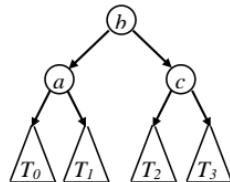


- Sejam as quatro subárvore restantes dos nós  $z$ ,  $y$  e  $x$ , designadas por  $T_0$ ,  $T_1$ ,  $T_2$  e  $T_3$ , ordenadas de forma que  $T_0$  tem os menores valores até  $T_3$  que tem os maiores valores (ver diagrama esquerdo).
- Ordenar os três nós  $x$ ,  $y$  e  $z$  por ordem crescente e renomeá-los  $a$ ,  $b$  e  $c$  (neste exemplo,  $a$  seria  $y$ ,  $b$  seria  $x$  e  $c$  seria  $z$ ).

- Substituir o nó  $z$  pelo nó  $b$ .
- Seja  $a$  a raiz da subárvore esquerda de  $z$  e  $T_0$  e  $T_1$  as respectivas subárvore.
- Seja  $c$  a raiz da subárvore direita de  $z$  e  $T_2$  e  $T_3$  as respectivas subárvore.

Para a remoção:

- Remove-se o elemento da árvore como se descreve na árvore binária de pesquisa. Se a árvore ainda estiver equilibrada o algoritmo termina. Se não, continuar.
- Seja  $z$  o nó de maior nível (do caminho desde o nó  $w$  que substituiu o elemento removido até à raiz) com o factor de equilíbrio diferente de  $-1$ ,  $0$  ou  $+1$ .
- Seja  $y$  o nó da subárvore de  $z$  com maior altura ( $y$  é o filho de  $z$  que não é ascendente de  $w$ ).
- Seja  $x$  o nó da subárvore de  $y$  com maior altura (ou um qualquer, se as duas subárvore tiverem a mesma altura).
- Repetir os cinco últimos pontos do algoritmo de inserção.



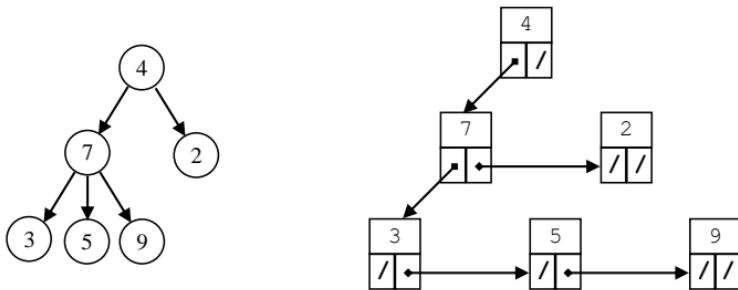
Na prática, qual é a altura das árvores AVL com  $n$  elementos? O estudo analítico é complicado mas testes empíricos mostraram que a altura média de uma árvore AVL é aproximadamente  $\log(n)+0.25$ . Assim, a complexidade da inserção e da remoção é muito próxima do óptimo teórico, sendo  $\Theta(\log(n))$ .

A árvore AVL, devido ao processo relativamente complexo de reequilíbrio, é preferencialmente usada em situações onde as pesquisas são mais frequentes que as inserções e remoções. No caso contrário, onde as inserções são muito mais frequentes que as consultas, a árvore binária de pesquisa pode ser suficiente. Existe ainda outra restrição, não é possível armazenar elementos repetidos numa árvore AVL. Não é possível (com esta representação) manter uma árvore binária de pesquisa com repetições e equilibrada simultaneamente.

## 17.3 Árvores Genéricas

Uma árvore binária é um caso particular de uma **árvore n-ária** em que os nós possuem, no máximo,  $n$  filhos. Uma árvore que tem um número finito mas não determinado de nós designa-se por **árvore genérica**. Nas árvores genéricas deixa de ser viável a existência de atributos específicos para identificar filhos (como as subárvores esquerda e direita das árvores binárias) porque não se sabe à partida qual o número máximo de filhos que cada nó da árvore suporta. É necessário um outro tipo de representação: em vez de referências só para nós filhos, a árvore possui uma referência para o *primeiro filho* e uma referência para o *próximo irmão* (em que primeiro e próximo são definidos por um determinado critério).

Graficamente, a árvore da esquerda seria representada pela estrutura da direita:



Desta forma é possível representar qualquer árvore (incluindo árvores binárias). É igualmente possível usar uma representação vectorial em que cada posição do vector (i.e., para cada nó) seja armazenado o índice do primeiro filho e o índice do próximo irmão.

### Árvores (a,b) e Árvores-B

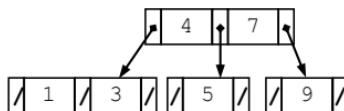
Um dos problemas práticos mais importantes no acesso é a leitura/escrita de informação em memória secundária (em discos rígidos, disquetes, CDs...). O acesso a informação externa é muito mais lento que o acesso à memória primária. Todos os mecanismos que minimizem o número de vezes que a memória externa é invocada são boas notícias para o desempenho geral do sistema. Iremos ter em conta os seguintes aspectos:

- Como referido no capítulo 7, por motivos de desempenho, a leitura e escrita da memória secundária ocorre em conjuntos de bytes e não byte a byte. Estes conjuntos de bytes (designados por blocos ou páginas) dependem do suporte físico, sendo normalmente potências de 2 (por exemplo, 1024, 2048, 4096 ou 8192 bytes).
- Devido às diferenças de velocidade, é vantajoso efectuar múltiplos acessos à memória primária se isso diminuir o número de acessos à memória secundária.

Estes factos apontam para um método de acesso: ler um conjunto de elementos de cada vez para a memória primária e, através de um algoritmo, decidir qual o próximo conjunto de elementos a ser lido e repetir até encontrar a informação necessária. Conhecendo os requisitos do suporte físico, determina-se o número mínimo e o número máximo de registo a serem lidos eficazmente de cada vez. Isto leva-nos à seguinte definição de árvore:

- Uma **árvore (a,b)** é uma árvore: (i) em que cada nó (exceptuando a raiz e as folhas) contém no mínimo  $a-1$  elementos e  $a$  filhos e no máximo  $b-1$  elementos e  $b$  filhos, com  $2 \leq a \leq (b+1)/2$  (ii) todas as folhas estão ao mesmo nível, (iii) os elementos a armazenar na árvore têm de ser comparáveis entre si, (iv) os elementos estão ordenados em cada nó e os elementos contidos em cada filho são maiores ou iguais ao elemento anterior e menores que o elemento seguinte.

Significa que a altura de uma árvore (a,b) que armazena  $n$  elementos é  $\Theta(\log(n))$ , ou mais especificamente, é  $\Omega(\log_b(n))$  e  $O(\log_a(n))$ . A principal diferença deste tipo de árvores e das referidas anteriormente encontra-se no número de elementos que cada nó armazena. As árvores binárias e n-árias armazenam apenas um elemento por nó, enquanto as árvores (a,b) armazenam de  $a-1$  a  $b-1$  elementos. A procura nas árvores (a,b) percorre o nó actual até encontrar o elemento; se o elemento não se encontra nesse nó, percorre o nó referenciado entre os dois elementos do nó actual no qual o elemento a procurar eventualmente se situa. No diagrama seguinte, ao procurar o número 5, o algoritmo iria requisitar o nó referenciado entre os números 4 e 7 para continuar a pesquisa. Com este método o número de nós percorridos (e carregados da memória secundária) pode ser substancialmente reduzido.



A maior complexidade encontra-se nos mecanismos de inserção e remoção quando se atinge o número máximo de nós (no caso da inserção) ou o número mínimo de nós (no caso da remoção). Nestes casos, é necessário um mecanismo de reequilíbrio para satisfazer a definição das árvores (a,b). A segunda condição (todas as folhas no mesmo nível) pode implicar, em certos casos, uma reestruturação da árvore.

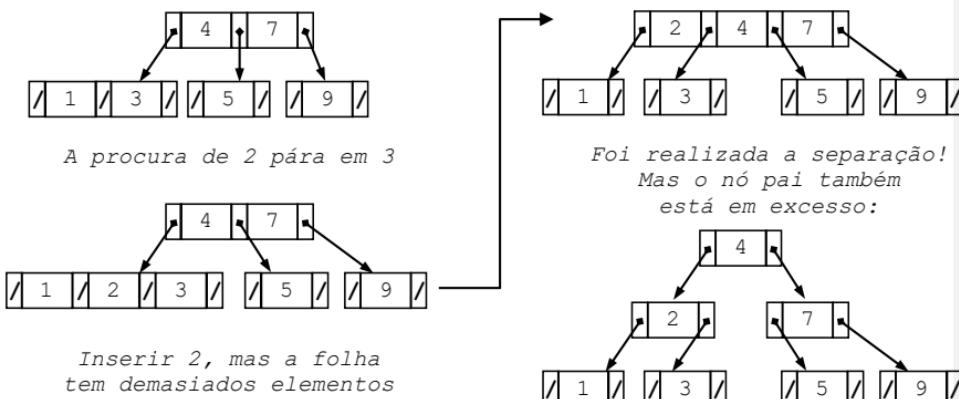
O algoritmo de inserção do elemento  $x$  pode ser descrito da seguinte forma:

- Pesquisar por  $x$ . A pesquisa terminará numa determinada folha (com insucesso se não for permitido repetições) numa determinada posição.
- Adicionar  $x$  nessa mesma folha respeitando a ordenação do nó.
- Se a folha, depois da inserção, tiver menos de  $b$  elementos, parar. Senão:
  - Escolher um elemento do meio (ou no caso do número de elementos ser par, escolher o primeiro elemento da segunda partição) designado  $w$ . Dividir essa

folha em duas novas folhas, F1 com os elementos (e filhos) até  $w$  e F2 com os elementos (e filhos) depois de  $w$ . É devido a este processo de divisão que se restringe os valores a  $2 \leq a \leq (b+1)/2$ .

- Passar  $w$  para o nó pai (se a folha a dividir era a raiz, criar uma nova raiz) e inseri-lo na posição correcta. O nó F1 será o filho antes de  $w$  e F2 o filho depois de  $w$ .
- Caso o nó pai, após a inserção de  $w$ , ficar com  $b$  elementos, repetir os dois últimos passos.

Considere o próximo exemplo com uma árvore (2,3) onde se insere o valor 2:

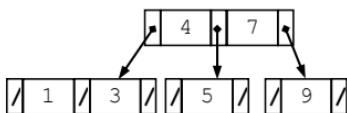


A operação de divisão modifica um número constante de elementos e, no máximo, há um número de divisões proporcional à altura da árvore. Assim a operação de inserção numa árvore (a,b) de  $n$  nós é  $O(\log(n))$ .

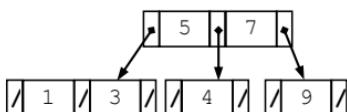
A operação de remoção de um elemento  $x$  pode ser descrita da seguinte forma:

- Encontrar o elemento  $x$ .
- Trocar  $x$  pelo próximo elemento, i.e., o elemento que está mais à esquerda da árvore da direita, ou se esta estiver vazia pelo próximo elemento. Repetir até  $x$  chegar a uma folha.
- Apagar  $x$ . Se a folha tiver  $a-1$  elementos ou mais parar. Senão:
  - Verificar se a soma dos elementos da folha anterior com os da folha actual é menor que  $b-1$ . Se for, fundir essas folhas juntamente com o elemento pai e formar uma nova folha (pode significar uma nova iteração de todo este processo, porque agora o nó pai ficou com menos um elemento). Se não for, transferir elementos de uma das folhas para a outra folha (incluindo o valor no nó pai) de modo a restabelecer o número mínimo de elementos por nó.

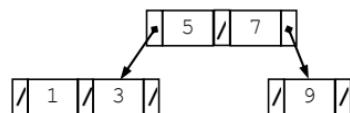
Considere a remoção do elemento 4 da seguinte árvore (2,3):



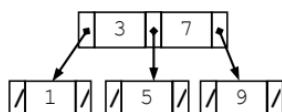
*Trocar 4 pelo próximo*



*Chegou a uma folha: apagar!*

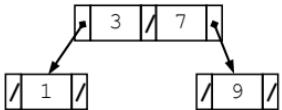


*Uma das folhas tem menos que o mínimo admissível!*

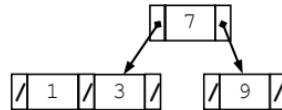


*Reagrupar com a folha anterior*

No exemplo anterior não houve necessidade de fusão de folhas. Veremos agora a remoção do elemento 5 da árvore anterior:



*Uma das folhas (entre 3 e 7) tem menos que o mínimo admissível!*



Neste exemplo, a folha (que ficou vazia), a folha anterior mais o elemento pai (o número 3) têm elementos suficientes para criar uma nova folha. O nó pai ficou com menos um elemento (neste exemplo ficou só com o número 7). Se esse nó pai ficar com menos elementos que o admitido, provoca-se um novo processo de fusão que pode ser propagado até à raiz. É somente nestes casos que as árvores (a,b) podem perder altura.

Tanto o processo de troca como o processo de fusão ocorrem, no máximo, um número de vezes igual à altura da árvore. Assim, a complexidade da remoção numa árvore (a,b) com n elementos é  $O(\log(n))$ .

Uma **árvore-B** (do inglês, *B-tree*), estrutura concebida por R. Bayer em 1970, é uma variante das árvores (a,b). Dado um valor inteiro e positivo  $n$ , a **árvore-B de ordem n** é uma árvore ( $\lceil n/2 \rceil, n$ ). Uma **árvore-B+** é uma variante da árvore-B onde as chaves dos elementos a procurar encontram-se nos nós intermédios (são elas que orientam a procura) e a informação dos elementos é armazenada somente nas folhas. As árvores-B+ são, possivelmente, a estrutura de dados mais usada para armazenar informação em memória secundária. O valor  $n$  deve ser igual ao número de elementos (incluindo as referências para

os filhos) que podem ser armazenados numa página, i.e., num acesso único de informação à memória secundária.

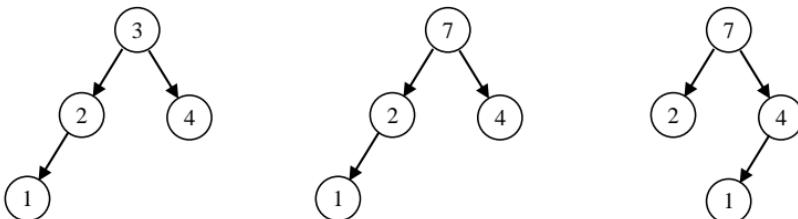
## 17.4 Amontoados

Um **amontoado** (do inglês, *heap*) ou **árvore binária com prioridade** é uma árvore binária com as seguintes restrições:

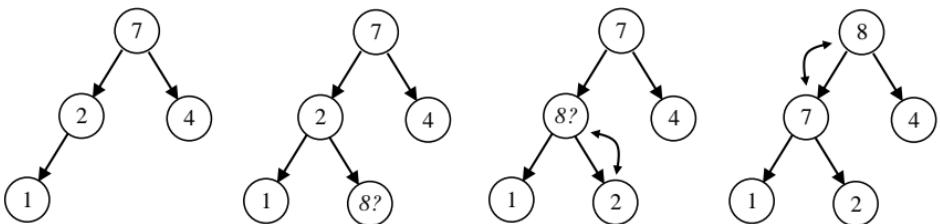
- As informações contidas nos nós da árvore são comparáveis entre si.
- A informação em cada nó pai é maior ou igual que as dos seus nós filhos.
- É uma árvore completa, i.e., está cheia até ao penúltimo nível e todos os nós do último nível estão dispostos da esquerda para a direita.

A segunda propriedade pode ser substituída requerendo que o valor do nó pai seja sempre menor ou igual que o valor dos nós filhos (as operações que se seguem mantêm-se idênticas bastando trocar os termos ‘maior’ e ‘menor’).

Das próximas três árvores, apenas a árvore do meio é um amontoado. A da esquerda possui um filho (4) maior que o pai (3) e a da direita não é uma árvore completa.



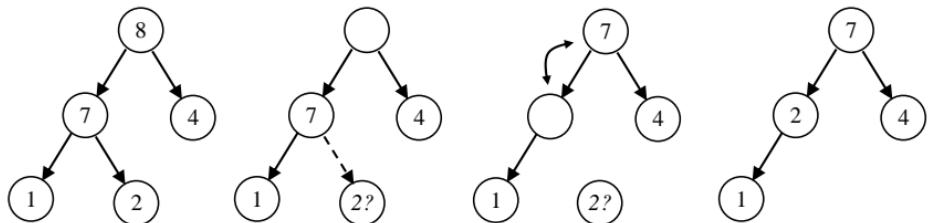
Como é necessário manter a árvore completa, a próxima informação deve ser colocada à direita do último nó inserido (se a árvore não estava cheia) ou no primeiro nó à esquerda de um novo nível (se a árvore estava cheia). O único problema é que a nova informação pode não ser menor que o seu pai, sendo necessário actualizar a árvore: 1º insere-se o nó no local referido, 2º enquanto nesse caminho o pai for menor que o filho, trocam-se os dois. O ciclo termina quando o pai for maior ou se for alcançada a raiz (neste caso, o novo elemento é maior ou igual que todos os outros). No exemplo seguinte observamos o processo de inserção do número 8 no amontoado.



Como a altura de uma árvore completa com  $n$  nós é  $\Theta(\log(n))$ , o número de trocas máxima é da mesma ordem. Logo, o processo de inserção num amontado é  $O(\log(n))$ .

Neste tipo de dados é suposto remover-se apenas a raiz. Para remover a raiz é preciso actualizar a árvore de modo a que se torne novamente num amontado. Como a estrutura perde um elemento, é preciso eliminar um nó da árvore. Esse nó, como na inserção, deve ser o mais à direita do último nível (para a árvore permanecer completa). Seja  $X$  a informação desse nó. Coloca-se na raiz o maior dos seus filhos e repete-se o processo para cada subárvore que perdeu a raiz até atingir uma folha ou um nó onde o maior dos filhos é menor que  $X$ . O valor  $X$  é inserido nesse nó.

Do exemplo anterior removemos a raiz (o valor 8):



Pelo mesmo motivo que na inserção, a complexidade da remoção da raiz de um amontado é  $O(\log(n))$ .

Duas aplicações para esta estrutura:

- Se removermos os elementos pela raiz, um a um, a sequência da remoção é ordenada. Esta é base para o algoritmo de ordenação *heapsort* (falaremos deste algoritmo no capítulo 19).
- Se interpretarmos a relação de ordem como uma prioridade da informação contida nesse nó, a remoção da raiz corresponde à remoção do elemento com maior prioridade. Este facto indica que um amontado é uma estrutura apropriada para representar filas de espera com prioridades.

## A Especificação Amontoado

Segue a especificação desta estrutura de dados.

```
especificação Heap<Element> =
importa BinTree<Element>
géneros Heap ≤ BinTree
```

Sendo um caso especial de árvore binária, o tipo amontoado é um subtipo de árvore binária. Como no TDA da árvore binária de pesquisa, a operação `constr` não deve ser usada pelo cliente.

### **operações**

```
isHeap : BinTree → Boolean
insert : Heap Element → Heap
remRoot : Heap ↗ Heap
```

Definem-se três operações: (i) uma interrogação que dada uma árvore binária, determina se é ou não um amontoado, (ii) inserção de um novo elemento e (iii) remoção do elemento da raiz.

### **auxiliares**

```
_isFull : BinTree → Boolean
_isComplete : BinTree → Boolean
_last : Heap ↗ Element
_remLast : Heap ↗ Heap
_makeHeap : Heap Heap Element ↗ Heap
```

Neste TDA existe um conjunto extenso de funções auxiliares para especificar as três operações. As duas primeiras determinam se uma árvore está cheia e/ou completa:

### **axiomas**

```
_isFull(empty) = TRUE
_isFull(constr(TL, TR, I)) =
    height(TL) = height(TR)
    and _isFull(TL)
    and _isFull(TR)
```

Os axiomas que definem uma árvore cheia correspondem à definição do início do capítulo. É um pouco mais complicado especificar a definição de árvore completa:

```
_isComplete(empty) = TRUE
_isComplete(constr(TL, TR, I)) =
    (height(TL) = height(TR) and
     _isFull(TL) and
```

```
_isComplete(TR))  
or  
(height(TL) = height(TR)+1 and  
_isComplete(TL) and  
_isFull(TR))
```

Determinar se uma árvore é um amontoado passa por verificar se a árvore é completa, se a raiz é maior que os filhos e se as subárvore são amontoados:

```
isHeap(empty) = TRUE  
isHeap(constr(TL,TR,J)) = _isComplete(constr(TL,TR,J))  
and root(TL) ≤ J  
and root(TR) ≤ J  
and isHeap(TL)  
and isHeap(TR)  
  
insert(empty,I) = constr(empty,empty,I)  
insert(constr(TL,TR,J),I) =  
  if height(TL) = height(TR)+1 and not isFull(TL)  
  then  
    if I ≤ J  
      then constr(insert(TL,I),TR,J)  
      else constr(insert(TL,J),TR,I)  
  else  
    if I ≤ J  
      then constr(TL,insert(TR,I),J)  
      else constr(TL,insert(TR,J),I)
```

O elemento a inserir é recursivamente colocado numa folha, depois é comparado com a raiz da última subárvore sendo trocado com ela (se for maior) e assim sucessivamente.

```
_last(constr(TL,TR,I)) =  
  if isEmpty(TL) and isEmpty(TR)  
  then I  
  else if height(TL) = height(TR)  
    then _last(TR)  
    else _last(TL)  
  
_remLast(constr(TL,TR,I)) =  
  if isEmpty(TL) and isEmpty(TR)  
  then empty  
  else if height(TL) = height(TR)  
    then constr(TL,_remLast(TR),J)  
    else constr(_remLast(TL),TR,J)
```

A operação `_last` devolve a informação que está no nó mais à direita no último nível e `_remLast` remove esse nó, esta operação substitui a raiz por um elemento adequado (uma das informações dos filhos ou a informação que se encontrava no nó removido).

No momento da remoção será necessário reconverter a árvore em amontoado, tarefa realizada pela seguinte operação auxiliar:

```
_makeHeap(empty,empty, I) =  
    constr(empty,empty,I)  
  
_makeHeap(constr(empty,empty,J),empty,I) =  
    if J≤I  
        then constr(constr(empty,empty,J),empty,I)  
    else constr(constr(empty,empty,I),empty,J)  
  
_makeHeap(constr(L1,R1,I1),constr(L2,R2,I2), I) =  
    if I1≤I and I2≤I then  
        constr(constr(L1,R1,I1),constr(L2,R2,I2),I)  
    else  
        if I2≤I1 then  
            constr(_makeHeap(L1,R1,I),constr(L2,R2,I2),I1)  
        else  
            constr(constr(L1,R1,I1),_makeHeap(L2,R2,I),I2)  
  
remRoot(T) = _makeHeap(left(_remLast(T)),  
                        right(_remLast(T)),  
                        _last(T))
```

A remoção da raiz define-se à custa das três operações auxiliares anteriores: construir um amontoado com as subárvore esquerda e direita da árvore onde foi removido o último elemento, juntamente com esse último elemento; a raiz é a única informação eliminada.

#### **pré-condições**

```
remRoot(T)  requer not isEmpty(T)  
  
_makeHeap(L,R,I)  requer isComplete(constr(L,R,I))  
_last(T)      requer isComplete(T) and not isEmpty(T)  
_remLast(T)   requer isComplete(T) and not isEmpty(T)
```

#### **fim-especificação**

## A Interface Amontoado

Esta interface corresponde ao TDA Heap da secção anterior.

```
package dataStructures;

/**
 * <p>Titulo: A interface do TDA Amontoado</p>
 * <p>Descrição: O desenho (Heap) com contratos</p>
 */

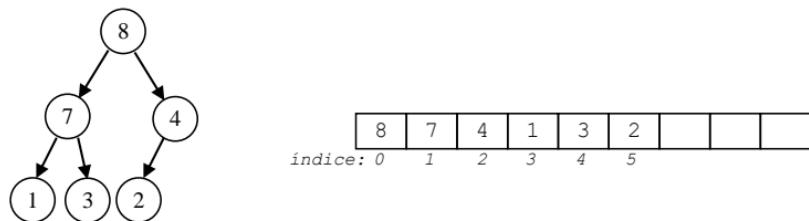
public interface Heap extends BinTree {
```

Como especificado, esta interface é uma extensão da árvore binária.

```
    /**
     * É a árvore um amontoado?
     * @return TRUE se for um amontoado, FALSE c.c.
     */
    //@ requires t != null;
    //@ ensures t.isEmpty() ==> \result;
    /*@ pure @*/ boolean isHeap(BinTree t);
    /**
     * Inserir um novo elemento
     * @param o A referência do elemento a ser inserido
     */
    //@ requires o != null;
    //@ requires isHeap(this);
    //@ ensures isHeap(this);
    void insert(Comparable o);
    /**
     * Remover o elemento da raiz
     */
    //@ requires !isEmpty();
    //@ requires isHeap(this);
    //@ ensures isHeap(this);
    void remRoot();
}
```

## Uma Implementação Estática

A definição do amontoado exige que a árvore seja completa. Ao percorrer a árvore em largura, a sequência resultante não efectua nenhum “salto” por entre referências vazias. Todos os níveis até ao penúltimo estão cheios e o último nível está organizado da esquerda para a direita. Esta organização da árvore é ideal para utilizar uma representação vectorial:



Como preencher o vector? Distribuir os elementos de forma a representar o percurso em largura do amontoado. Calcular os índices dos filhos de um nó pai é imediato: se o nó pai está no índice  $i$ , o filho da esquerda está em  $2.i+1$  e o filho da direita está em  $2.i+2$ . Dado um nó de índice  $i$ , o pai é dado pela expressão  $\lfloor(i-1)/2\rfloor$ . A implementação seguinte utiliza esta representação vectorial:

```

package dataStructures;

 $\ast \ast \ast$ 
 $\ast$  <p>Título: Uma implementação vectorial do Amontoado</p>
 $\ast$  <p>Descrição: Esta implementação usa uma implementação
 $\ast$            vectorial para representar amontoados</p>
 $\ast /$ 

public class VHeap implements Heap,Cloneable {
  
```

```

    private final int DELTA = 128;
    private int nElems;
    private Object[] theHeap;
  
```

O atributo `nElems` armazena o número actual de elementos no amontoado.

```

public VHeap() {
    theHeap = new Object[DELTA];
    nElems = 0;
}

//@ requires isHeap(t);
public VHeap(BinTree t) {
    this();
    constr(t.root(),t.left(),t.right());
}
  
```

Este segundo construtor traduz uma árvore binária num amontoado desde que a pré-condição seja satisfeita.

```

public BinTree empty() {
    return new VHeap();
}
  
```

Estamos a implementar a classe desde o início (não se estendeu a classe DBinTree porque VHeap usa uma representação vectorial). Como consequência é necessário implementar todos os métodos descritos nas interfaces Heap e BinTree. Existe, porém, o problema já referido do método `constr()` cuja invocação no contexto dos amontoados pode provocar uma violação no estado do objecto enquanto amontoado. É tarefa do cliente usar este método com cuidado (para não violar as pré-condições das operações do TDA Heap). A única forma de inserção que garante o estado válido do objecto é através do método `insert()`.

```
public void constr(Object item,
                    BinTree left, BinTree right) {
    theHeap = new Object[DELTA + left.length() +
                        right.length()];
    theHeap[0] = item;
    nElems = 1;
    VQueue qTrees = new VQueue(); // fila de árvores
    VQueue qIndex = new VQueue(); // fila de inteiros
    qTrees.enqueue(left);
    qIndex.enqueue(new Integer(1));
    qTrees.enqueue(right);
    qIndex.enqueue(new Integer(2));
    while (!qTrees.isEmpty()) {
        BinTree t = ((BinTree)(qTrees.front()));
        int i = ((Integer)(qIndex.front())).intValue();
        qTrees.dequeue();
        qIndex.dequeue();
        theHeap[i] = t.root();
        nElems = i+1;
        if (!t.left().isEmpty()) {
            qTrees.enqueue(t.left());
            qIndex.enqueue(new Integer(left(i)));
        }
        if (!t.right().isEmpty()) {
            qTrees.enqueue(t.right());
            qIndex.enqueue(new Integer(right(i)));
        }
    } //while
}
```

Como funciona o método `constr()`? É necessário traduzir o conteúdo de duas árvores binárias (das quais sabemos os métodos da interface `BinTree` mas não como são representadas) para a representação vectorial da classe. Para isso usou-se duas filas: uma que contém a próxima árvore a ser processada e outra que guarda o índice do vector onde a raiz dessa árvore será armazenada. Para cada árvore retirada da primeira fila, insere-se as suas subárvores não vazias e determina-se as posições das respectivas raízes (inserindo esses valores na segunda fila).

```
public boolean isEmpty() {  
    return nElems==0;  
}  
  
private int left(int index) {  
    return 2 * index + 1;  
}  
  
private int right(int index) {  
    return 2 * index + 2;  
}  
  
private int father(int index) {  
    return (index-1)/2;  
}
```

O método `father()` funciona porque o operador `/` para inteiros devolve apenas o quociente da divisão inteira (por exemplo,  $4/2$  é igual a  $5/2$ ).

```
private boolean isEmpty(int index) {  
    return index >= nElems || theHeap[index] == null;  
}
```

Considera-se que uma posição é vazia se for maior que os índices que armazenam a informação válida do amontado. De notar que existem dois métodos com nome `isEmpty()` (mas diferentes assinaturas). O primeiro (o que não tem argumentos) verifica se a estrutura de dados está vazia e este verifica se uma dada posição está ocupada ou não.

```
private boolean isLeaf(int index) {  
    return isEmpty(left(index)) &&  
           isEmpty(right(index));  
}
```

O método privado `isLeaf()` verifica se o nó no índice `index` é uma folha ou não. Como a raiz está no índice zero, o método público `isLeaf()` torna-se trivial:

```
public boolean isLeaf() {  
    return isLeaf(0);  
}
```

De seguida estão implementados as três operações da árvore binária: `root()`, `left()` e `right()`.

```
public Object root() {  
    return theHeap[0];  
}  
  
public BinTree left() {  
    int i=0;  
    VHeap result = new VHeap();  
    result.theHeap = new Object[nElems];  
    for(int start=2,size=1; start<=nElems;  
        start*=2,size*=2)  
        for(int j=start-1;j<start-1+size;j++) {  
            if (j>=nElems)  
                break;  
            result.theHeap[i++] = theHeap[j];  
        }  
    result.nElems = i;  
    return result;  
}
```

Os métodos `left()` e `right()` criam novos vectores com os elementos das respectivas subárvores. Para isso, é necessário ter em conta a disposição desses elementos no vector a partir da representação do amontoado escolhida para esta classe. Por exemplo, os elementos incluídos na subárvore esquerda encontram-se no vector, nos índices 1 (o filho da esquerda que se encontra no nível 2), 3 e 4 (os nós do nível 3), 7 a 10 (os nós do nível 4), 15 a 22 (os nós do nível 5)... Cada iteração do ciclo principal corresponde a um nível. A variável `start` indica onde se encontra o nó inicial desse nível e `size` indica quantos nós têm de ser copiados para o resultado final.

```
public BinTree right() {  
    int i=0;  
    VHeap result = new VHeap();  
    result.theHeap = new Object[nElems];  
    for(int start=2,size=1; start<=nElems;  
        start*=2,size*=2)  
        for(int j=start-1+size;j<start-1+2*size;j++) {  
            if (j>=nElems)  
                break;  
            result.theHeap[i++] = theHeap[j];  
        }  
    result.nElems = i;  
    return result;  
}
```

```
/**
 * A árvore está cheia?
 * @param t A árvore
 * @return TRUE se 't' é cheia, FALSE c.c.
 */
private boolean isFull(BinTree t) {
    return Math.pow(2,t.height())-1 == t.length();
}
```

O método `isFull()` foi implementado de forma diferente que a descrita no TDA (o relevante é respeitar o comportamento especificado). A complexidade do algoritmo descrito na especificação é exponencial, enquanto se verificarmos a relação que existe numa árvore cheia entre o número de nós e a sua altura ( $2^{\text{altura}} - 1 = \text{dimensão}$ ) obtemos um algoritmo de complexidade linear.

```
/**
 * A árvore está completa?
 * @param t A árvore
 * @return TRUE se 't' é completa, FALSE c.c.
 */
private boolean isComplete(BinTree t) {
    if (t.isEmpty())
        return true;

    BinTree l = t.left(),
           r = t.right();

    return (l.height() == r.height()
            && isFull(l) && isComplete(r))
        || (l.height() == r.height()+1
            && isFull(r) && isComplete(l));
}
```

Foram implementadas duas das operações auxiliares utilizadas no TDA, `isFull` e `isComplete`. No entanto (como já foi referido) não existe qualquer obrigação de implementar operações auxiliares de TDAs.

```
/**
 * A subárvore é de prioridade?
 * @param t A árvore
 * @return TRUE se 't' é de prioridade, FALSE c.c.
 */
private boolean isPriority(BinTree t) {
    if (t.isEmpty() || t.isLeaf())
        return true;
```

```
BinTree l = t.left(),
        r = t.right();

if (l.isEmpty())
    return ((Comparable)t.root()).compareTo(r.root())>=0;

if (r.isEmpty())
    return ((Comparable)t.root()).compareTo(l.root())>=0;

return ((Comparable)t.root()).compareTo(r.root())>=0 &&
        ((Comparable)t.root()).compareTo(l.root())>=0 &&
        isPriority(r) && isPriority(l);
}
```

Foram definidos dois métodos privados para determinar se uma subárvore é completa e/ou é de prioridade (i.e., todos os filhos são menores que o respectivo pai). Para determinar se a árvore é um amontoado verificamos ambos os resultados:

```
public boolean isHeap(BinTree t) {
    return isComplete(t) && isPriority(t);
}
```

Os dois métodos privados seguintes devolvem as alturas e dimensões de subárvores do amontoado sendo necessários na execução de outros métodos.

```
private int length(int index) {
    if (isEmpty(index))
        return 0;
    return 1+length(left(index))+length(right(index));
}

private int height(int index) {
    if (isEmpty(index))
        return 0;
    return 1 + Math.max(height(left(index)),
                         height(right(index)));
}

public int length() {
    return length(0);
}

public int height() {
    return height(0);
}
```

```
private void grow() {
    Object[] newHeap =
        new Object[theHeap.length + DELTA];
    for(int i=0;i<theHeap.length;i++)
        newHeap[i] = theHeap[i];
    theHeap = newHeap;
}
```

O método `grow()` aumenta a capacidade do objecto. Segue o método de inserção:

```
public void insert(Comparable o) {
    if (nElems==theHeap.length)
        grow();
    theHeap[nElems++] = o;
    moveUp();
}

private void moveUp() {
    int pos = nElems-1;
    while (pos != 0 && ((Comparable)theHeap[pos]).
            compareTo(theHeap[father(pos)]) > 0) {
        swap(pos, father(pos));
        pos = father(pos);
    }
}

private void swap(int i, int j) {
    Object tmp = theHeap[i];
    theHeap[i] = theHeap[j];
    theHeap[j] = tmp;
}
```

A inserção funciona precisamente como foi descrita no início da secção (e especificada no TDA Heap). O mesmo se passa com a remoção da raiz:

```
public void remRoot() {
    if (nElems>1) {
        theHeap[0] = theHeap[nElems-1];
        moveDown();
    }
    theHeap[--nElems] = null;
}
```

```
private void moveDown() {
    int maxChild, pos = 0;

    while (pos <= father(nElems-1)) {
        maxChild = left(pos);
        if (maxChild < nElems-1)
            if (((Comparable)theHeap[maxChild]).compareTo(theHeap[maxChild+1]) < 0)
                maxChild++;
        if (((Comparable)theHeap[pos]).compareTo(theHeap[maxChild]) > 0)
            break;
        swap(pos, maxChild);
        pos = maxChild;
    }
}
```

A interface BinTree ainda exige a implementação do iterador e das travessias em profundidade. Pede-se ao leitor, como exercício, para preencher estes quatro métodos:

```
public Iterator iterator() { return null; }
public void prefix(Visitor op) {};
public void sufix(Visitor op) {};
public void infix(Visitor op) {};
```

A classe termina com os métodos habituais (`equals()`, `toString()` e `clone()`):

```
public boolean equals(Object t) {
    if (!(t instanceof BinTree))
        return false;
    if (isEmpty())
        return ((BinTree)t).isEmpty();
    if (((BinTree)t).isEmpty())
        return false;
    return ((BinTree)t).root().equals(root()) &&
           ((BinTree)t).left().equals(left()) &&
           ((BinTree)t).right().equals(right());
}
```

No primeiro condicional do método `equals()` pergunta-se se o objecto referenciado por `t` é do tipo árvore binária e não somente do tipo amontoado. A pergunta é pertinente, um amontoado é uma árvore sendo admissível que se compare com qualquer árvore binária. De facto, a pergunta mais que pertinente é obrigatória, devido ao segundo construtor desta

classe que inicializa um objecto amontoado de acordo com a estrutura de uma árvore binária.

```
public String toString() {
    if (isEmpty())
        return "[]";
    StringBuffer result =
        new StringBuffer("[" + theHeap[0]);
    for (int i=1; i<nElems; i++)
        result.append(", " +
                      (theHeap[i]==null?"[]":theHeap[i]));
    return result.toString() + "]";
}

public Object clone() {
    VHeap newHeap = new VHeap();
    newHeap.theHeap = (Object[]) (theHeap.clone());
    newHeap.nElems = nElems;
    return newHeap;
}
} // endClass VHeap
```

## Usar a classe **VHeap**

Segue um exemplo do uso dos métodos desta classe:

```
public static void main(String[] args) {
    VHeap h = new VHeap();
    h.insert(new Integer(3));
    h.insert(new Integer(2));
    h.insert(new Integer(1));
    h.insert(new Integer(21));
    h.insert(new Integer(14));
    h.insert(new Integer(25));
    h.insert(new Integer(12));
    h.insert(new Integer(0));
    h.insert(new Integer(122));
    System.out.println(h);
    h.remRoot();
    System.out.println(h);
    System.out.print("height:" + h.height());
    System.out.print(" length:" + h.length());
}
```

```
□ [122, 25, 21, 14, 3, 1, 12, 0, 2] ←  
[25, 14, 21, 2, 3, 1, 12, 0] ←  
height:4 length:8
```

---

## Exercícios

1. Construa exemplos das seguintes árvores binárias: (i) completa com altura 4 e 10 nós, (ii) com 10 nós e o número mínimo de folhas, (iii) com altura 3 e 7 nós mas não completa.
2. Responda às seguintes questões:
  - i) Quantos nós existem numa árvore binária cheia de altura 4? E altura 12?
  - ii) Demonstrar que uma árvore binária cheia de altura  $H$  possui  $2^H - 1$  nós.
  - iii) Quantas folhas existem numa árvore binária cheia de altura 4? E de altura 12?
  - iv) Mostre que uma árvore estritamente binária (i.e., cada nó possui zero ou dois filhos) tem um número ímpar de nós.
  - v) Quantas folhas existem numa árvore estritamente binária com 17 nós? E com 713 nós?
  - vi) Mostre que uma árvore binária com  $N$  nós contém  $2N+1$  árvores (incluindo a árvore total). Quantas dessas árvores são vazias?
  - vii) Demonstrar que numa árvore binária, o número máximo de folhas é igual ao número de nós interiores mais um.
3. Implemente um método que receba uma árvore binária representando uma expressão aritmética (cada nó da arvore contém ou um inteiro ou uma das 4 operações básicas) e calcule o valor final. (*pista:* usar uma pilha e o percurso prefixo).
4. Estender a especificação `BinTree` com as funções: (i) `isFull`, verifica se a árvore dada está cheia; (ii) `isBalanced`, verifica se a dada árvore está equilibrada; (iii) `nChilds`, devolve o número de filhos da raiz da árvore; (iv) `strict`, verifica se a árvore é estritamente binária.
5. Estenda a interface `BinTree` e a implementação `DBinTree` tendo em conta o exercício anterior.
6. Estender a especificação `SearchBinTree` com as funções: (i) `min`, que devolve o valor mínimo da árvore; (ii) `max`, que devolve o valor máximo da árvore.
7. Estenda a interface `SearchBinTree` e a implementação `DSearchBinTree` tendo em conta o exercício anterior.
8. Implemente a interface `BinTree` com uma classe que use uma representação vectorial.
9. Implemente a interface `SearchBinTree` com uma classe que use uma representação vectorial.

10. Uma árvore binária é uma estrutura de dados eficiente para representar conjuntos. Construa uma implementação do tipo `Set` através de uma árvore binária.
11. Implemente a classe `DBalancedTree` sendo uma extensão de `DSearchBinTree`, onde são codificadas a inserção e remoção de árvores AVL.
12. Especifique o tipo `Tree` com operações semelhantes ao tipo `BinTree`, mas para uso de árvores n-árias. Defina a interface e realize uma implementação dinâmica deste tipo.
13. Qualquer vector ordenado de forma decrescente pode ser interpretado como um amontoado? Ser um vector decrescente é condição necessária para ser um amontoado?
14. Quais das sequências seguintes são amontoados? (i) 30 15 19 12 9 18, (ii) 30 15 19 12 19 18, (iii) 80 35 80 12 19 18 15 1 4 2 e (iv) 1.
15. Considere uma linha de produção de peças. As peças podem ser de tipos diferentes cada uma com um grau de prioridade distinto. São produzidas em primeiro lugar as peças com maior prioridade e dentro de cada tipo é irrelevante a ordem pela qual chegam os pedidos. Suponha que se utiliza um amontoado para armazenar a lista de pedidos de produção de peças. (i) Em que posição do vector está o pedido da próxima peça a ser produzida? Justifique, (ii) Diga, justificando, se a sequência 17 12 10 10 8 6 7 5 9 de prioridades correspondente a 9 pedidos de peças, é um amontoado, (iii) Descreva o algoritmo para remoção de pedidos do amontoado. Exemplifique retirando o elemento de maior prioridade da sequência da alínea anterior e (iv) Descreva um algoritmo para a inserção de pedidos no amontoado. Exemplifique inserindo primeiro 11 e depois 15.

# 18 – Tabelas

---

Um dos métodos para organizar um conjunto de elementos é associar a cada elemento um componente que o identifique. A este componente é costume designar-se **chave** (em inglês, *key*). Uma chave é necessariamente única para cada elemento. Um exemplo é o número do bilhete de identidade que todos os portugueses possuem. É fácil observar que se o número do BI não fosse único para cada indivíduo, a utilidade deste seria questionável. Neste caso, a chave (o número do BI) não tem nenhuma relação lógica com o elemento (a pessoa). A função injectiva que os relaciona é definida por extenso (i.e., caso a caso) e criada por processos administrativos.

Uma **tabela**, ou **dicionário** (em inglês, *dictionary*), é uma estrutura de dados que armazena um conjunto finito de **registos** (em inglês, *records*). Cada registo é constituído por um par de valores, nomeadamente uma chave e um elemento. O elemento (de um dado tipo) contém a informação do que se quer guardar. A chave identifica univocamente esse elemento em relação aos restantes elementos.

Um outro exemplo de tabela foi introduzido no capítulo 3 com o conceito de vector: a chave é o índice da posição onde o elemento se encontra. No vector não existe, à partida, qualquer relação entre a chave e o elemento. Por esse motivo a procura de um elemento específico dentro do vector pode ter de percorrer todos os elementos até encontrar o pretendido, possuindo uma complexidade linear. Um dos objectivos do capítulo é mostrar como se pode melhorar este desempenho.

Quando é possível ordenar as chaves (i.e., existe uma ordem que as relaciona) diminui-se a complexidade linear para uma complexidade logarítmica: armazena-se os elementos ordenados por chave e realiza-se uma pesquisa binária (como foi explicado na secção

17.2). O que permite a existência de dicionários e enclopédias é a capacidade de usar uma chave (a própria palavra) para ordenar (através da ordem lexicográfica) a informação associada à palavra em questão.

Mas será possível melhorar ainda mais a complexidade da busca? A verdade é que se estivermos dispostos a sacrificar outro tipo de recurso que não seja o tempo, a resposta é sim. O recurso que falamos, como já ocorreu noutros casos, é o espaço. Não é a primeira vez que nos surge esta questão: é possível poupar tempo sacrificando espaço? Não é raro existir uma opção entre algoritmos mais rápidos que gastam mais memória e algoritmos mais lentos que gastam menos memória. Neste caso em particular, estando disposto a gastar mais memória obtém-se algoritmos de busca e inserção que, no caso médio, são de complexidade temporal constante, i.e.,  $O(1)$ .

## 18.1 Especificação da Tabela

A representação de uma tabela é definida recursivamente a partir de duas operações construtoras: `empty` que representa a tabela vazia e `insert` que representa uma tabela constituída por um registo (uma chave do tipo `Key` e o elemento do tipo `Element`) e a restante tabela.

As outras operações definidas sobre o TDA são:

- `isEmpty` – verifica se a tabela está vazia.
- `contains` – verifica se um elemento representado pela chave está na tabela.
- `retrieve` – devolve o elemento associado à chave dada.
- `remove` – remove da tabela o elemento associado à chave dada.

Segue o Tipo de Dados Abstracto:

```
especificação Table<Element, Key> =
  importa Boolean
  géneros Table
  operações

    empty : → Table
    insert : Table Element Key → Table
    isEmpty : Table → Boolean
    contains : Table Key → Boolean
    retrieve : Table Key → Element
    remove : Table Key → Table

  axiomas

    isEmpty(empty)           = TRUE
    isEmpty(insert(T, I, K)) = FALSE
```

```
contains(empty, K) = FALSE
contains(insert(T, I, K1), K) =
    equals(K, K1) or contains(T, K)
retrieve(insert(T, I, K1), K) =
    if equals(K, K1)
        then I
    else retrieve(T, K)
remove(empty, K) = empty
remove(insert(T, I, K1), K) =
    if equals(K, K1)
        then T
    else insert(remove(T, K), I, K1)

pré-condições
insert(T, I, K) requer not contains(T, K)
retrieve(T, K) requer contains(T, K)

fim-especificação
```

## 18.2 A Interface Tabela

Segue o interface Java:

```
package dataStructures;

/**
 * <p>Título: A interface do TDA Tabela</p>
 * <p>Descrição: O desenho (Table) com contratos</p>
 */

public interface Table {
    /**
     * Criar uma tabela vazia
     * @return uma Tabela vazia
     */
    //@ ensures \result.isEmpty();
    /*@ pure */ Table empty();

    /**
     * Inserir na tabela o item com chave key
     * @param key A chave do elemento
     * @param item A referência para o elemento a inserir
     */
    //@ requires item != null && key != null;
    //@ requires !contains(key);
    //@ ensures !isEmpty();
```

```
//@ ensures    contains(key);
//@ ensures    retrieve(key).equals(item);
void insert(Object item, Object key);
```

As três pós-condições do método `insert()` referem-se respectivamente aos axiomas das operações `isEmpty`, `contains` e `retrieve`.

```
/**
 * A tabela está vazia?
 * @return TRUE se a estrutura está vazia, FALSE c.c.
 */
/*@ pure */ boolean isEmpty();

/**
 * Esta chave pertence à tabela?
 * @param key A chave a procurar
 * @return TRUE se pertence, FALSE c.c.
 */
//@ requires key != null;
//@ ensures \old(isEmpty()) ==> !\result;
/*@ pure */ boolean contains(Object key);

/**
 * Devolver o elemento com chave key
 * @param key A chave a procurar
 * @return A referência do objecto com a chave
 */
//@ requires key != null;
//@ requires contains(key);
/*@ pure */ Object retrieve(Object key);

/**
 * Remover o elemento com chave key
 * @param key A chave a procurar
 */
//@ requires key != null;
//@ ensures \old(isEmpty()) ==> isEmpty();
//@ ensures !contains(key);
void remove(Object key);

/**
 * A tabela 't' é igual a esta?
 * @param t A tabela a ser comparada
 * @return TRUE se são iguais, FALSE c.c.
 */
//@ also
//@ requires t != null;
/*@ pure */ boolean equals(Object t);
```

```
/**  
 * @return Um iterator para os elementos do conjunto  
 */  
/*@ pure @*/ Iterator iterator();  
  
/**  
 * Devolver uma cópia da estrutura  
 * @return devolve uma referência para a cópia  
 */  
//@ also  
//@ ensures (\result != this) && equals(\result);  
/*@ pure @*/ Object clone();  
}  
} //endInterface Table
```

De seguida vamos discutir uma implementação das tabelas onde se procura minimizar a complexidade temporal de acesso à custa de um maior gasto de memória.

### 18.3 Tabelas de Dispersão

A **tabela de dispersão** (do inglês, *hash table*) é uma implementação do TDA tabela em que o valor da chave associado ao elemento é usado para calcular a posição onde esse elemento deve ser inserido e procurado. Este tipo de implementação pressupõe a existência de uma função  $h()$  que relaciona o domínio do tipo da chave (designado por K) e o contradomínio definido pelo conjunto de endereços da estrutura de dados (designado por A). Esta função  $h$  é designada por **função de dispersão** (do inglês, *hash function*):

$$h : K \rightarrow A$$

O problema principal quando se pretende encontrar esta função de dispersão é que o espaço disponível para armazenar as chaves é muito menor que o número de combinações possíveis de valores da chave, i.e.,  $|K| \gg |A|$ . Ou seja, na maioria dos casos, a função de dispersão não é injectiva. Existem diversas chaves que quando aplicada  $h$  vão resultar no mesmo endereço. Este tipo de problema é designado por **colisão**.

Se a função de dispersão fosse bijectiva representaria uma dispersão perfeita sem a existência de colisões. Nos casos em que  $|K| \gg |A|$  essa função perfeita não é possível. E de facto, uma análise combinatória revela que as probabilidades de colisão são bastante altas mesmo para um reduzido conjunto de elementos inseridos. Como exemplo a probabilidade de, num conjunto de 23 pessoas, encontrarmos duas com a mesma data de aniversário é maior que 50%!

Encontramos assim, dois problemas a resolver:

- Que função  $h$  usar?
- Como fazer para resolver potenciais colisões?

Para a primeira questão é preciso conhecer os detalhes específicos do problema (qual o tipo da chave? Qual a memória disponível, i.e., qual a cardinalidade de A?). Consideramos que o vector onde é armazenada a informação possui uma dimensão N (desde o índice 0 até ao índice N–1). Admitimos igualmente a existência de uma função (designada por *ord*) que transforma a chave (de um tipo arbitrário) num valor inteiro. A forma de definir *h*:

$$h(k) = \text{ord}(k) \bmod N$$

A operação *mod* é o resto da divisão inteira (aritmética módulo N). Esta função foi utilizada na implementação estática das filas de espera para criar um vector circular. Deve-se escolher um número primo para o valor de N devido ao seu pequeno número de divisores (ver a sequência de pesquisa descrita adiante).

Existem técnicas de processamento inicial da chave que se reflectem na função *ord*():

- Se partes da chave possuem pouca diversidade, convém removê-las antes de aplicar a função de dispersão. Por exemplo, se a chave é o número do BI e o universo dos elementos a inserir são pessoas com sensivelmente a mesma idade, a variação dos dígitos na ordem dos milhões é muito pequena.
- Dividir a chave em partes e agrupá-las de alguma forma. Esta forma pode melhorar a variabilidade da chave inicial. Por exemplo, sendo uma chave uma palavra (i.e., uma sequência de caracteres) atribui-se a cada caracter um valor e adicionam-se para criar um número. Por exemplo,  $\text{ord}(\text{"zebra"}) = \text{ord}(\text{"z"}) * 3 + \text{ord}(\text{"e"}) * 3^2 + \text{ord}(\text{"b"}) * 3^3 + \text{ord}(\text{"r"}) * 3^4 + \text{ord}(\text{"a"}) * 3^5$ , onde o valor de uma letra é igual à posição no alfabeto (i.e.,  $\text{ord}(\text{"a"}) = 1$ ,  $\text{ord}(\text{"b"}) = 2$ , etc.).

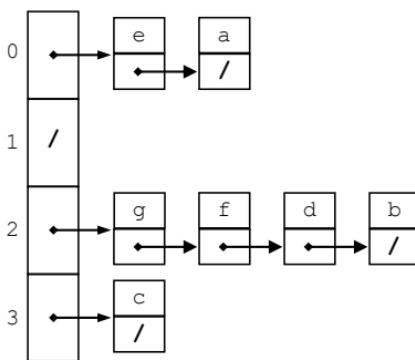
Outro método de dispersão, designado por método da multiplicação, é descrito pela seguinte função (os símbolos  $\lfloor \cdot \rfloor$  representam a parte inteira de um número):

$$h(k) = \lfloor N \cdot (A \cdot \text{ord}(k) - \lfloor A \cdot \text{ord}(k) \rfloor) \rfloor, A \in [0,1]$$

Um valor comum para A é o número de ouro  $\varphi = (\sqrt{5}-1)/2 \approx 0,6180339887\ldots$ . A escolha de N não é tão crítica como no método do módulo (costuma ser uma potência de 2 para acelerar as multiplicações).

A segunda questão – como se resolvem colisões? – depende essencialmente da estrutura de dados utilizada. Se a opção focar um **endereçamento fechado** (do inglês, *closed addressing* ou *direct chaining*) resolve-se a questão armazenando todos os elementos cuja chave devolve o mesmo valor da função de dispersão numa lista.

Neste exemplo, numa tabela com quatro posições, foram inseridos elementos com chaves ‘a’ a ‘g’ (omite-se a informação associada à chave por motivos de clareza já que não é relevante para a explicação) usando uma função de dispersão que nas chaves em questão devolveu  $h(a)=h(e)=0$ ,  $h(g)=h(f)=h(d)=h(b)=2$  e  $h(c)=3$ .



No endereçamento fechado, a colisão é resolvida inserindo o novo elemento à cabeça do objecto lista. Todos os elementos cuja chave aplicada sobre a função de dispersão devolve o mesmo valor encontram-se na mesma lista. A solução apresentada por este método para resolver as colisões é bastante directa mas possui uma potencial desvantagem: apesar do algoritmo de inserção (inserir na cabeça da lista) ser  $O(1)$ , se a função de dispersão não realiza o trabalho de forma efectiva (i.e., os elementos vão-se acumulando em poucas listas), a complexidade da busca pode degenerar numa busca linear, dado ser necessário procurar dentro de uma lista progressivamente maior pelo elemento pretendido. No exemplo acima, poder-se-ia suspeitar da existência de uma assimetria observando a posição de valor 1 (com zero elementos) e a de valor 2 (com quatro elementos).

O **endereçamento aberto** (do inglês, *open addressing*) representa a opção por uma estrutura de dados vectorial, em que cada posição armazena, não uma lista, mas apenas um e um único registo. Quando uma posição está ocupada é preciso procurar outra posição. Qual? É necessário fornecer uma segunda função que permita encontrar uma ou mais localizações alternativas para a chave que provocou a colisão. Esta nova função designa-se por **função de pesquisa** e permite resolver as eventuais colisões que surjam (do inglês, *collision handling*). A função de pesquisa recebe dois argumentos: a chave do elemento a procurar/inserir e um valor inteiro  $n$  que representa a  $n$ -ésima tentativa.

$$h_i : K \times \mathbb{N}_0 \rightarrow A$$

É conveniente que esta função garanta as condições seguintes:

- $h_i(k,0) = h(k)$ , i.e., a primeira tentativa procura na posição indicada pela função de dispersão.
- Dado o vector (que guarda a informação) de dimensão  $N$ , a sequência de valores  $h_i(k,0), h_i(k,1), h_i(k,2), \dots, h_i(k,N-1)$  deve passar por todos os valores possíveis do vector. Esta sequência designa-se por **sequência de pesquisa** (do inglês, *probing sequence*).

A segunda condição é necessária porque desde que a tabela tenha pelo menos uma posição livre, o algoritmo de pesquisa deve ser capaz de encontrá-la. Idealmente, a função de pesquisa deve satisfazer:  $h_i(k,0) = h_i(k,N)$ . Assim, se não existirem posições livres, a função irá chegar ao local de partida: a memória está cheia.

O método mais simples é designado por **pesquisa linear** (do inglês, *linear probing*):

$$h_i(k,i) = (h(k) + i) \bmod N$$

Este método, apesar da simplicidade, tem a desvantagem dos elementos se agruparem em aglomerados (do inglês, *clusters*) à volta de uma ou mais chaves relativamente próximas. Este detalhe diminui o desempenho da busca, degenerando em pesquisas lineares quando ocorrem sobre aglomerados. Uma forma mais geral desta pesquisa:

$$h_i(k,i) = (h(k) + a.i) \bmod N$$

O valor constante  $a$  deve ser primo em relação a  $N$  (i.e., o máximo divisor comum de  $a$  e  $N$  é 1). Se o m.d.c fosse  $x > 1$ , apenas  $1/x$  da tabela seria visitada pela sequência de pesquisa.

Este problema pode ser minimizado (mas não eliminado) com a **pesquisa quadrática**:

$$h_i(k,i) = (h(k) + i^2) \bmod N$$

No entanto, esta função viola a segunda condição referida atrás: nem todos os elementos da tabela são visitados. De facto, se  $N$  for primo, pelo menos metade das posições são visitadas, mas não se garante mais. É possível que este método de pesquisa não encontre uma das posições livres. Na prática, pelo mesmo motivo que é fácil ocorrerem colisões, é fácil encontrar uma posição vaga. Logo, este tipo de problema só ocorre quando a tabela está praticamente cheia.

Dado o **fator de saturação** (do inglês, *load factor*)  $\alpha$  (igual ao número de posições ocupadas a dividir pelo número total de posições) é possível, através da análise estatística do caso médio, determinar que o número esperado de colisões é igual a:

$$C = \frac{1-\alpha/2}{1-\alpha}$$

Por exemplo, mesmo para a pesquisa linear, se  $\alpha=0.5$  (i.e., metade da tabela está ocupada),  $C=1.5$ . Se  $\alpha=0.95$  então  $C=10.5$ : mesmo com 95% da tabela ocupada, o método de pesquisa mais fraco procura, em média, apenas 10 ou 11 posições.

Uma das formas mais eficazes de distribuir os elementos com a mesma chave é através de uma **pesquisa com dispersão dupla**. Neste método é necessário uma segunda função de dispersão (em inglês, *rehashing*) para ser usada na sequência de pesquisa (designada por  $h_1$ ):

$$h_i(k,i) = (h(k) + i.h_1(k)) \bmod N$$

Mesmo que as chaves de dois elementos produzam o mesmo valor quando aplicada a função  $h$ , os valores já serão diferentes para a função  $h_1$  diminuindo assim a criação de aglomerados. Este método é considerado como a melhor forma para criar uma sequência de pesquisa.

Um método final é a **pesquisa aleatória**. É usado um gerador de números pseudo-aleatórios para evitar a criação de aglomerados:

$$h_i(k,i) = (h(k) + i \cdot \text{rand}()) \bmod N$$

O gerador deve ser uniforme no intervalo  $[0, N-1]$  de modo a todas as posições terem a mesma probabilidade de serem escolhidas. O gerador deve ser previsível, i.e., tem de possuir um mecanismo de semente inicializado com a chave do elemento a inserir. Doutra forma, não seria possível pesquisar eficientemente um elemento, porque não haveria nenhuma forma de saber onde tinha sido inserido. Apesar da distribuição dos elementos para evitar colisões ser muito eficiente, o gerador pode demorar demasiado tempo a encontrar uma célula vaga se a tabela for grande e estiver quase cheia.

Até agora foram referidas técnicas para inserção e pesquisa sobre uma tabela de dispersão. Como se efectua a remoção? No caso do endereçamento fechado, a solução é trivial: remover o elemento da lista. Para o endereçamento aberto a solução não é tão directa. Não se pode remover simplesmente o elemento da tabela. Considere o exemplo seguinte (usou-se uma sequência com pesquisa linear):

				a	b	d	c			f		g	h		
--	--	--	--	---	---	---	---	--	--	---	--	---	---	--	--

Do aglomerado com quatro elementos ( $a, b, c, d$ ), o elemento  $d$  está na posição que a função de dispersão lhe atribuiu inicialmente, enquanto  $a, b$  e  $c$  possuem chaves com o mesmo índice, razão pela qual  $b$  e  $c$  estão em posições resultantes de uma e três colisões. Resumindo,  $h(a)=h(b)=h(c)=h(d)-2$ .

Se removermos o elemento  $b$  sem mais nenhuma alteração, essa posição é considerada vazia. O que acontece se procurarmos  $c$ ?

				a		d	c			f		g	h		
--	--	--	--	---	--	---	---	--	--	---	--	---	---	--	--

A função de dispersão, dada a chave de  $c$ , devolve o índice onde se situa  $a$ . Como não encontra  $c$ , a sequência de pesquisa passa para a próxima posição onde encontra uma posição vazia e, por isso, pára. A resposta seria que  $c$  não existe na tabela. O que aconteceu? A posição de onde  $b$  foi removido não se encontra verdadeiramente vazia, pois ainda faz parte de um aglomerado (não o dividiu em dois). A posição está vaga, não vazia. É necessário substituir o elemento removido por um símbolo alternativo que represente esse facto (designemos por  $\rightarrow$ ).



Assim, quando o algoritmo procura *c*, vai passar pela posição vaga e continua até encontrar o elemento ou uma verdadeira posição vazia.

## A interface *HashFunction*

Primeiro definimos a interface da função de dispersão. Esta interface exige a concretização de três métodos:

- `int hashCode(Object key)` – a função de dispersão *h()*.
- `boolean isHashCode(Object key)` – um método que verifica se o objecto referenciado por *key* pertence ao domínio da função de dispersão.
- `int nextValue()` – a função de pesquisa que devolve o próximo valor. É assumido que pesquisa todas as posições da tabela e que ao fim de uma pesquisa completa volta ao valor inicial da função de dispersão.

```
package dataStructures;

/**
 * <p>Titulo: A interface do TDA Hashable</p>
 * <p>Descrição: Para uso de tabelas de dispersão</p>
 */

public interface HashFunction {

    /**
     * Calcular o valor de dispersão de uma dada chave
     * @param key A chave a ser calculada
     * @return O valor dessa chave
     */
    //@@ requires isHashCode(key);
    int hashCode(Object key);

    /**
     * Este objecto tem valor de dispersão?
     * @param key A chave a verificar
     * @return TRUE se tem valor de dispersão, FALSE c.c.
     */
    /*@ pure */ boolean isHashCode(Object key);

    /**
     * Calcular o n-ésimo valor depois de (n-1) colisões,
     * em relação à última chave inquirida.
     * @return O próximo valor
     */
    int nextValue();

} //endInterface HashFunction
```

Esta interface define-se o tipo das funções de dispersão que podem ser usadas nas classes que implementam a interface `Table`. Isto é necessário dado que no momento da implementação dessas classes não ser possível conhecer qual a função de dispersão (isso vai depender de cada problema específico). De facto, a interface disponibiliza o serviço da função de dispersão (através do método `hashValue()`) bem como da função de pesquisa (através do método `nextValue()`).

Outra opção seria definir duas interfaces separadas, uma para a função de dispersão outra para a função de pesquisa.

## A classe `HashRegister`

Define-se igualmente a classe do tipo registo. Esta classe possui dois atributos (a chave e a informação associada) e os métodos `toString()` e `clone()`. Esta classe não é pública pois é somente usada pela classe `HashTable`.

```
package dataStructures;      // ficheiro hashTable.java
/** Titulo: O tipo dos registos a guardar na Tabela */
class HashRegister implements Cloneable {
    public Object item;
    public Object key;
    public HashRegister(Object o, Object k) {
        item = o;
        key = k;
    }
    public String toString() {
        return "(" + item + ":" + key + ")";
    }
    public Object clone() {
        return new HashRegister(item, key);
    }
} //endClass HashRegister
```

## A classe `HashTable`

```
// ... continuação do ficheiro HashTable.java
/** Titulo: Uma implementação vectorial da Tabela */
public class HashTable implements Table,Cloneable {
    private final int DELTA = 1001;
    protected final double LOAD_FACTOR = 0.75;
```

Definiu-se duas constantes, a primeira refere a dimensão inicial da tabela e a segunda determina qual o factor de saturação para o qual a tabela deve aumentar a sua estrutura de armazenamento. Optou-se por não esperar que a tabela se encha, de modo a minimizar o número de colisões para cada inserção.

```
protected static final HashRegister availableCell  
= new HashRegister(null,null);
```

O atributo `availableCell` é um atributo de classe e uma referência constante. Este objecto (com referências nulas) é o símbolo especial que representa a posição vaga depois da remoção de um registo (a seta do último diagrama).

```
private int nElems;           // Quantos elementos existem  
private HashRegister[] table; // O vector da tabela  
private HashFunction h;      // A função de dispersão
```

Na implementação seguinte optou-se por um endereçamento aberto. Para além dos atributos que armazenam a dimensão da tabela e do número de elementos actuais, existe um vector de objectos `HashRegister` (um vector de registos) e ainda um objecto `h` que referencia a função de dispersão a ser usada. Este objecto é inicializado no construtor:

```
public HashTable(HashFunction h) {  
    this.h = h;  
    table = new HashRegister[DELTA];  
    nElems = 0;  
}
```

O vector de referências é criado. Por convenção, uma referência nula representa uma posição vazia. O cliente que utiliza a tabela de dispersão necessita de fornecer uma referência para um objecto `HashFunction`. Não existe uma função de dispersão padrão porque o tipo da chave é totalmente arbitrário, tem de ser o cliente a definir a função apropriada.

```
public Table empty() {  
    return new HashTable(h);  
}  
  
public boolean isEmpty() {  
    return nElems == 0;  
}  
  
private boolean isCellFree(int n) {  
    return table[n] == null;  
}  
  
private boolean isCellAvailable(int n) {  
    return table[n] == availableCell;  
}
```

Os métodos `isCellFree()` e `isCellAvailable()` verificam respectivamente se uma posição está vazia e se está vaga. Como referimos, uma posição vaga não separa os elementos vizinhos, mantendo-os no mesmo aglomerado.

```
/**  
 * Devolver o índice onde se encontra a chave  
 * @param key A chave a procurar  
 * @return O índice ou -1 se a chave não existir  
 */  
  
private int searchPos(Object key) {  
    int pos = h.hashValue(key) % table.length;  
    int back = pos;  
  
    do {  
        if (isCellFree(pos))  
            return -1;  
        if (isCellAvailable(pos))  
            pos = h.nextValue() % table.length;  
        else if (key.equals(table[pos].key))  
            return pos;  
        else  
            pos = h.nextValue() % table.length;  
    } while (back!=pos);  
  
    return -1;  
}
```

Reparar no uso da sequência de pesquisa se existirem colisões. O método auxiliar `searchPos()` devolve a posição (i.e., o índice do vector) onde o elemento (com a chave dada) se encontra. Se o elemento não existir, devolve -1. Esta funcionalidade é usada para simplificar os três métodos seguintes:

```
public boolean contains(Object key) {  
    return searchPos(key) >= 0;  
}  
  
public Object retrieve(Object key) {  
    return table[searchPos(key)].item;  
}  
  
public void remove(Object key) {  
    int n = searchPos(key);  
  
    if (n!=-1) {  
        table[n] = availableCell;  
        nElems--;  
    }  
}
```

No método `remove()` é necessário conferir se o elemento existe na tabela, dado que esta operação não possui a pré-condição da chave pertencer à tabela. Não se trata de programação defensiva mas apenas de garantir o comportamento especificado: nada ocorre na remoção de uma chave não existente.

A questão do crescimento da estrutura através do método `grow()` é mais delicada para as tabelas de dispersão, pois não basta aumentar a dimensão do vector. A posição de cada item baseia-se num cálculo onde essa dimensão entra em conta. Logo, quando se aumenta o vector é necessário introduzir de novo todos os elementos.

```
private void grow() {
    HashRegister[] oldTable = table;
    int newLength = getNextPrime(table.length+DELTA);
    table        = new HashRegister[newLength];
    nElems       = 0;
    for(int i=0;i<oldTable.length;i++)
        if (oldTable[i] != availableCell &&
            oldTable[i] != null)
            insert(oldTable[i].item, oldTable[i].key);
}
```

A utilização do método `getNextPrime()` (que definimos a seguir) tem a ver com a preocupação de criar vectores com dimensões apropriadas (como referido, os números primos são as melhores escolhas). Por exemplo, com `DELTA` igual a 1001 e sendo essa a dimensão actual do vector, para crescer, adiciona-se outro `DELTA` ( $1001+1001=2002$ ) e procura-se o número primo seguinte (2003). A tabela ficaria com dimensão 2003. Este cálculo é efectuado pelos dois métodos seguintes:

```
//@ requires n>1;
private boolean isPrime(int n) {
    if (n%2 == 0)
        return n==2;
    int limit = (int)Math.round(Math.sqrt(n));
    for(int i=3;i<=limit;i+=2)
        if (n%i == 0)
            return false;
    return true;
}
```

```
private int getNextPrime(int n) {
    while (!isPrime(n))
        n++;
    return n;
}
```

Podemos, finalmente, implementar o método de inserção:

```
public void insert(Object item, Object key) {
    if ((double)nElems/table.length > LOAD_FACTOR)
        grow();                                // encontrar a posição inicial
    int pos = h.hashValue(key) % table.length;
                                                // enquanto estiver num aglomerado...
    while (!isCellFree(pos) && !isCellAvailable(pos))
        pos = h.nextValue() % table.length;      // criar registo
    table[pos] = new HashRegister(item, key);
    nElems++;
}
```

No método `insert()` o vector é aumentado quando se ultrapassa um determinado factor de saturação dado pela constante `LOAD_FACTOR`. É conveniente não deixar que a tabela se encha demasiado para evitar a degradação no tempo de acesso aos registos (por excesso de colisões).

Seguem os métodos habituais e a definição da classe local para o iterador:

```
/***
 * Traduzir a tabela, eg: [| (key1:item1), (key2:item2) |]
 * @return descrição da tabela numa string.
 */
public String toString() {
    StringBuffer result = new StringBuffer("[| ");
    for(int i=0;iif (!isCellFree(i) && !isCellAvailable(i))
            result.append(table[i]+",");
    return result.substring(0,result.length()-1) + " |]";
}

public Object clone() {
    HashTable cp = new HashTable(h);
    cp.table     = new HashRegister[table.length];
    cp.nElems    = nElems;
```

```
cp.h          = h;
for(int i=0;i<table.length;i++) {
    if (isCellFree(i))
        continue;
    if (isCellAvailable(i))
        cp.table[i] = availableCell;
    else
        cp.table[i] = (HashRegister)(table[i].clone());
}
return cp;
}

public boolean equals(Table t) {
    if (!(t instanceof Table))
        return false;
    Iterator it = ((Table)t).iterator();
    while (it.hasNext())
        if (!contains(((HashRegister)it.next()).key))
            return false;
    it = iterator();
    while (it.hasNext())
        if (!((Table)t).contains(
                ((HashRegister)it.next()).key))
            return false;
    return true;
}
```

Considera-se que duas tabelas são iguais se cada tabela conter todas as chaves da outra.

```
public Iterator iterator() {
    return new TableIterator();
}

</** Iterador **
private class TableIterator implements Iterator {
    private int nextElem;
    private TableIterator() {
        nextElem = isEmpty() ? -1 : 0;
    }
```

```
public Object next() {
    if (nextElem >= 0)
        for (int i=nextElem; i<table.length; i++)
            if (!isCellAvailable(i) && !isCellFree(i)) {
                nextElem = i+1;
                return table[i];
            }
    nextElem = -1;
    return null;
}

public boolean hasNext() {
    if (nextElem != -1)
        for (int i=nextElem; i<table.length; i++)
            if (!isCellAvailable(i) && !isCellFree(i))
                return true;
    return false;
}

public void remove() {
    throw new UnsupportedOperationException();
}

} //endLocalClass TableIterator
} //endClass HashTable
```

## Usar a classe *HashTable*

Para usar esta classe é preciso definir uma função de dispersão conveniente no contexto do problema. Considere um registo constituído por uma chave do tipo *String* que identifica uma informação igualmente do tipo *String*. Vamos implementar uma classe *HashFunction* para uma função de dispersão que soma todos os caracteres da chave (uma *string*). A sequência de pesquisa é uma pesquisa linear.

```
class HashString implements HashFunction {

    private int actualValue;

    public HashString() {
        actualValue = 0;
    }

    public int hashCode(Object key) {
        actualValue=0;

        for(int i=0;i<((String)key).length();i++)
            actualValue += (int)((String)key).charAt(i);
        return actualValue;
    }
}
```

```
public boolean isHashable(Object key) {
    return key instanceof String;
}

public int nextValue() {
    actualValue += 1;
    return actualValue;
}
}
```

Definida esta classe:

```
public static void main(String[] args) {
    HashTable ht = new HashTable(new HashString());
    ht.insert("elefante","03e");
    ht.insert("tigre","12t");
    ht.insert("zebra","03z");
    ht.insert("gnu","54p");
    ht.remove("12t");
    ht.insert("foca","02f");
    HashTable cp = (HashTable)(ht.clone());
    ht.insert("tigre2","13t");
    System.out.println(ht);
    System.out.println(cp);
}

□ [ | (03z:zebra),(02f:foca),(03e:elefante),(12t:tigre2),(54p:gnu) | ] ←
    [ | (03z:zebra),(02f:foca),(03e:elefante),(54p:gnu) | ] ←
```

---

## Exercícios

1. Implementar o TDA tabela numa tabela de dispersão com endereçamento fechado.
2. Criar para o exemplo da secção anterior, as implementações seguintes do tipo HashFunction: (i) com uma função de pesquisa quadrática, (ii) com uma função de pesquisa aleatória, (iii) com uma função de pesquisa com dispersão dupla.
3. Implementar para o exemplo da secção anterior, outras funções de dispersão que manipulem chaves do tipo `String`.
4. Um **vector flexível** (do inglês, *flexible array*) possui o acesso e actualização a qualquer elemento através do seu índice `e`, além disso, permite a inserção e remoção de elementos na sua primeira ou última posição (ou seja, a sua dimensão total é dinâmica). Apresente a especificação do vector flexível (defina o TDA `flexArray`) que inclui as seguintes

funções: (i) `empty`, para criar um novo vector, (ii) `insertBg`, para inserir um novo elemento (do tipo `Element`) no início do vector, (iii) `removeBg`, para remover o elemento do início do vector, (iv) `insertEnd`, para inserir um novo elemento (do tipo `Element`) no fim do vector, (v) `removeEnd`, para remover o elemento do fim do vector, (vi) `acess`, que dado um vector e um índice, devolve o elemento nesse índice do vector, (vii) `update`, que dado um vector, um índice e um elemento, colocar esse elemento nesse índice do vector, (viii) `isEmpty`, que verifica se o vector está vazio, (ix) `length`, que devolve o número de elementos e (x) `equals`, que compara se dois vectores são iguais.

5. Crie a interface `FlexArray` e realize uma implementação dinâmica `DFlexArray` tendo em conta o exercício anterior.

6. Uma tabela múltipla é uma variante do tipo tabela onde os elementos podem partilhar “chaves”, ou seja, é possível armazenar elementos com a mesma chave. Apresente a especificação da tabela múltipla (defina o TDA `MTable`) que inclui as seguintes funções: (i) `empty`, para criar uma nova tabela, (ii) `insert`, para inserir um novo elemento (do tipo `Element`), (iii) `remove`, para remover o(s) elemento(s) com uma dada chave, (iv) `retrieve`, para devolver uma lista com o(s) elemento(s) de uma dada chave, (v) `isEmpty`, que verifica se a tabela está vazia e (vi) `length`, que devolve o número de elementos.

7. Crie a interface `MTable` e realize uma implementação vectorial `VMTTable` tendo em conta o exercício anterior.



# 19 – Ordenação

---

Este capítulo centra-se na seguinte pergunta: *como se ordena um conjunto de objectos?* Primeiro, para ordenar objectos estes têm de ser comparáveis<sup>33</sup>. Se objectos de um dado tipo são comparáveis implica que existe um critério que para um par de objectos determina se o primeiro objecto é maior, menor, ou igual ao segundo. Esse critério de comparação, seja  $\leq$ , para funcionar adequadamente necessita de ser reflexivo ( $X \leq X$ ), anti-simétrico (se  $X \leq Y$  é verdadeiro então  $Y \leq X$  é falso) e transitivo (se  $X \leq Y$  e  $Y \leq Z$  então  $X \leq Z$ ). Para além destas três propriedades (que definem uma ordem parcial) todos os elementos têm de ser comparáveis entre si (i.e., para todo  $X$  e  $Y$ , ou  $X \leq Y$  ou  $Y \leq X$ ). Veremos alguns algoritmos que providos deste critério de comparação, recebem um conjunto de elementos desordenados e devolvem o conjunto ordenado.

## 19.1 Algoritmos de Ordenação

Cada um dos algoritmos seguintes propõe uma estratégia de ordenação diferente. Alguns são mais eficientes que outros e, como veremos, pertencem a uma de duas classes de complexidades temporais.

Existem duas tarefas principais nestes algoritmos de ordenação: a *comparação* entre dois objectos e a *troca* de posição entre dois objectos. Consoante os problemas, o custo

---

<sup>33</sup> É fácil ordenar um conjunto de alunos (alfabeticamente pelos nomes ou numericamente pelos números de aluno) mas como se ordena um conjunto de gatos?

associado à comparação pode ser muito diferente do custo associado à troca (custo, aqui, significa maior tempo de execução). Vejamos duas das possíveis situações:

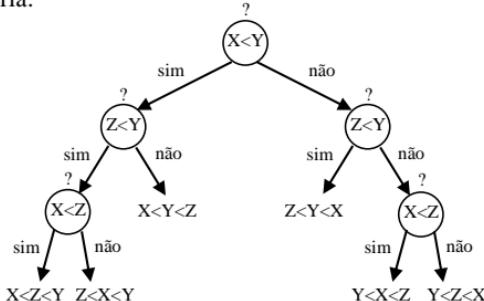
- A comparação custa muito mais que a troca. Por exemplo, os objectos são matrizes, o critério de comparação é baseado no valor do determinante e a troca passa somente por uma mudança de referências. Os algoritmos de ordenação que efectuem poucas comparações são preferíveis nestes casos.
- A troca custa muito mais que a comparação. Por exemplo, os objectos são inteiros armazenados num ficheiro sequencial: a comparação é imediata (comparar dois inteiros) mas a troca é dispendiosa (alterar o conteúdo de um ficheiro sequencial). Os algoritmos de ordenação que efectuem poucas trocas são preferíveis nestes casos.

Considere um algoritmo que compare todos os pares de objectos para que a ordenação seja realizada. Se o conjunto possui  $n$  objectos, a complexidade deste tipo de algoritmo é  $O(n^2)$  dado que cada objecto é comparado com todos os outros. Esta é a maior quantidade de informação que se pode recolher do conjunto, definindo um limite máximo para os algoritmos de ordenação (não existem mais perguntas relevantes a fazer).

Qual será o limite inferior (para o pior caso) de um algoritmo de ordenação baseado na comparação de pares de objectos? A pergunta “X é menor que Y?” possui duas respostas possíveis: sim e não. Se existem  $n$  elementos no conjunto a ordenar, existe um número mínimo de perguntas “X é menor que Y?” que necessitam de resposta para o conjunto ficar ordenado. O limite inferior é proporcional a esse número mínimo de perguntas necessárias. Mas que limite é esse?

Para um conjunto de  $n$  objectos existem  $n!$  permutações válidas. É possível criar uma árvore binária que dada uma pergunta inicial (com um ramo para a resposta afirmativa e outro para a resposta negativa) passe para a pergunta seguinte e assim sucessivamente até atingir as folhas que correspondem às diferentes soluções. Este tipo de árvores designa-se por **árvore de decisão**.

Por exemplo, considere o conjunto {X,Y,Z} a ordenar por ordem crescente. Uma possível árvore de decisão seria:



Considere que existe um método que decide como distribuir as perguntas para criar a árvore com a menor altura possível (designemos essa árvore por T). O número de folhas de T é igual a  $n!$  (i.e., o número de permutações). O menor número de perguntas, no pior caso, é igual à altura de T, limitada inferiormente por  $\log_2(n!) \geq \log_2((n/2)^{n/2}) = (n/2)\log_2(n/2) \in \Omega(n\log(n))$ . Assim, um limite inferior para algoritmos de ordenação baseado em comparações é  $\Omega(n\log(n))$ .

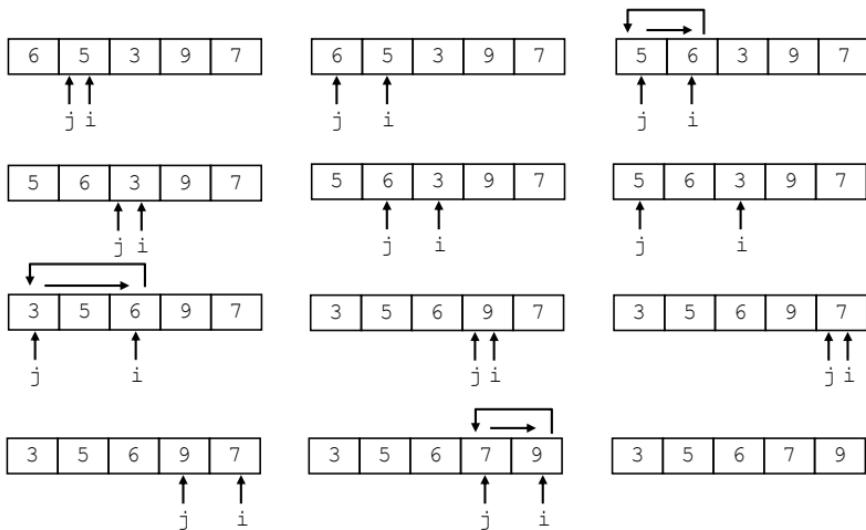
Convencionamos que um algoritmo de ordenação com complexidade  $O(n\log(n))$  é um algoritmo rápido e com complexidade  $O(n^2)$  é um algoritmo lento.

Outro factor a ter em conta é a complexidade de cada algoritmo consoante o melhor caso, o pior caso e o caso médio. Por norma, como medida de segurança para o desempenho do algoritmo, considera-se o pior caso. No entanto, um algoritmo é interessante se o caso médio possuir uma complexidade baixa e o pior caso (mesmo com uma complexidade alta) for suficientemente raro para não ter relevância prática. Esse é o caso do algoritmo *quicksort* que veremos adiante.

Descrevem-se, a seguir, alguns algoritmos de ordenação existentes na literatura. Para simplificar a explicação (não há perda de generalidade) consideramos o exemplo da ordenação de números inteiros de forma crescente. Os programas da secção 19.2 foram desenhados para funcionarem em contextos mais gerais (com vectores de objectos comparáveis).

## O algoritmo Insertsort

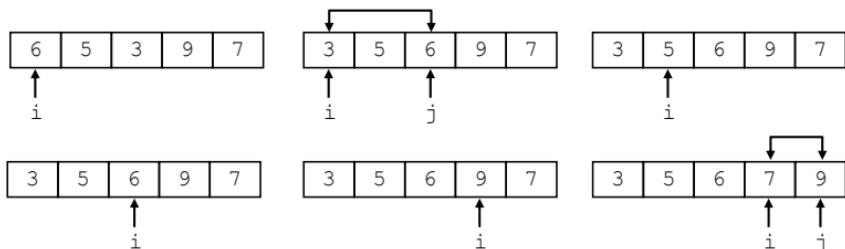
Este algoritmo procura, para cada índice do vector (excluindo o primeiro), quais dos elementos à sua esquerda (i.e., com menor índice) são maiores. Ao deslocar a pesquisa para a esquerda, se atingir o índice zero ou encontrar um elemento menor ou igual que o elemento associado ao índice actual, desloca todos os outros elementos uma casa para a direita e coloca o elemento actual na posição deixada vaga. Um exemplo (neste e nos próximos exemplos,  $i$  e  $j$  referem-se respectivamente às variáveis de progresso do ciclo principal e do ciclo interior):



No melhor caso possível (o algoritmo já está ordenado) este algoritmo de ordenação é  $\Theta(n)$ . No pior caso (o algoritmo está inversamente ordenado) e no caso médio (uma distribuição inicial aleatória dos elementos) a complexidade é  $O(n^2)$ . Tanto o número de comparações como o número de trocas é elevado e são limitados superiormente por  $O(n^2)$ .

## O algoritmo Selectionsort

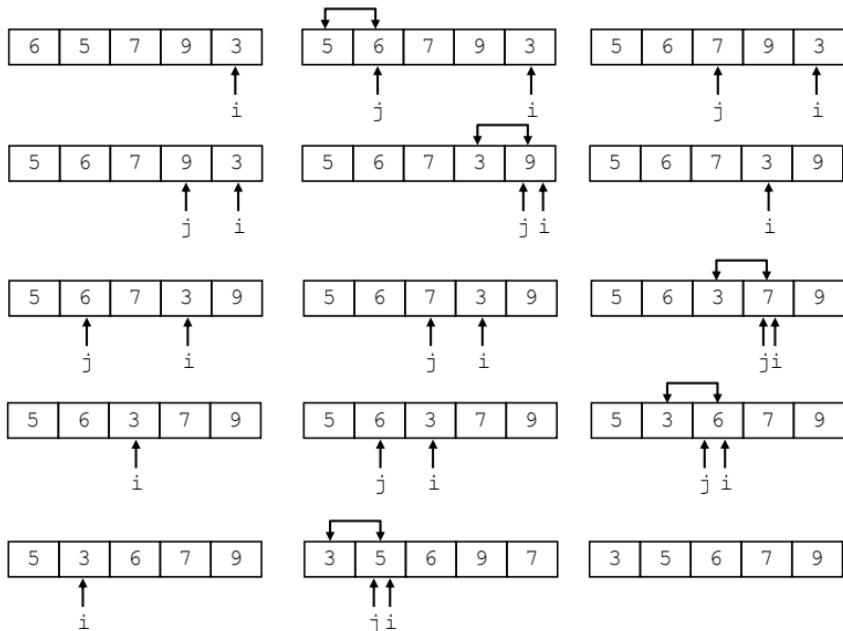
Este algoritmo procede da forma seguinte: para o índice *i* do vector *v*, selecciona o menor valor que se encontra entre o índice *i* e o último índice. Assim, na primeira iteração (*i*=0) encontra o menor valor do vector e troca-o com *v[0]*. Depois encontra o menor valor entre *v[1]* e o ultimo índice e troca-o com *v[1]*. Repete o processo até ao penúltimo índice (não é necessário fazer para o último índice).



Este algoritmo procura sempre o menor elemento para cada posição do vector, independentemente da ordem dos elementos. Possui complexidade  $O(n^2)$  para o melhor caso, o pior caso e o caso médio. Enquanto o número de comparações é alto, i.e., é limitado por  $O(n^2)$ , o número de trocas é baixo sendo limitado por  $O(n)$  para o pior caso (uma troca por cada iteração).

## O algoritmo Bubblesort

Este algoritmo efectua uma série de passagens pelo vector, comparando cada par de valores adjacentes. Para cada par na ordem errada, troca as posições dos dois elementos. No fim da primeira passagem, o maior valor encontra-se na última posição. Na segunda passagem são vistos os valores do vector exceptuando o último e assim sucessivamente até à última passagem que apenas compara os dois primeiros elementos (trocando-os se for o caso).



A complexidade do algoritmo é  $O(n^2)$  em todos os casos. O número de comparações em todos os casos é limitado por  $O(n^2)$ . Para o pior caso e para o caso médio, o número de trocas é limitado por  $O(n^2)$ . No melhor caso (o algoritmo já está ordenado) nenhuma troca é executada.

## O algoritmo Quicksort

Os algoritmos anteriores são considerados lentos. As complexidades nos piores casos e nos casos médios são de  $O(n^2)$ . O algoritmo *quicksort*, inventado por C. Hoare e apresentado em 1962, possui a complexidade óptima  $O(n \cdot \log(n))$  para o caso médio. O algoritmo é recursivo, dividindo a tarefa de ordenar o vector em duas ordenações de vectores menores (a técnica de dividir para conquistar). Este processo recursivo continua até se chegar a um caso de ordenação trivial (por exemplo, vectores de dimensão um ou dois).

Como em qualquer algoritmo recursivo é necessário compor a resposta do problema a partir das respostas dos subproblemas. Para minimizar esta composição os subvectores a ordenar são processados de modo a distribuir os menores valores para um vector e os maiores para outro vector. Desse modo, a solução do problema passa por anexar as soluções dos dois subproblemas.

Por exemplo, para ordenar este vector:

3	9	6	7	5	1	4	2	8	0
---	---	---	---	---	---	---	---	---	---

Colocar-se-ia os menores valores na metade esquerda e os maiores na metade direita:

3	2	4	1	0	7	6	5	8	9
---	---	---	---	---	---	---	---	---	---

Dividir-se-ia os dois vectores para serem ordenados:

3	2	4	1	0
---	---	---	---	---

7	6	5	8	9
---	---	---	---	---

Quando as soluções estiverem disponíveis (através da invocação recursiva do algoritmo)...

0	1	2	3	4
---	---	---	---	---

5	6	7	8	9
---	---	---	---	---

...unem-se os dois vectores produzidos pelos subproblemas de modo a compor a resposta do problema inicial.

0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---

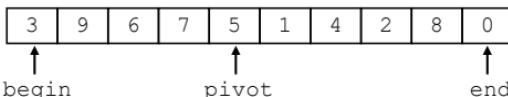
Este é o funcionamento básico do *quicksort*. Falta uma pergunta (e uma resposta): como se separam os menores valores para a esquerda e os maiores para a direita de uma forma eficiente?

O algoritmo escolhe um valor arbitrário no vector (normalmente no meio do vector) designado por *pivot*. O *pivot* é utilizado para comparar os valores do vector, os valores menores que o *pivot* são armazenados na esquerda, os maiores na direita. Em cada invocação do algoritmo existem duas variáveis, *begin* e *end* (com valores inicialmente

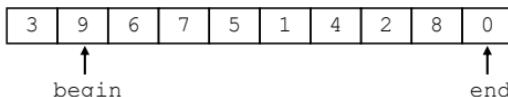
iguais aos índices do início e do fim do vector) que avançam em direcção ao *pivot*. Cada par de valores nas metades incorrectas é trocado. Usando o exemplo anterior:



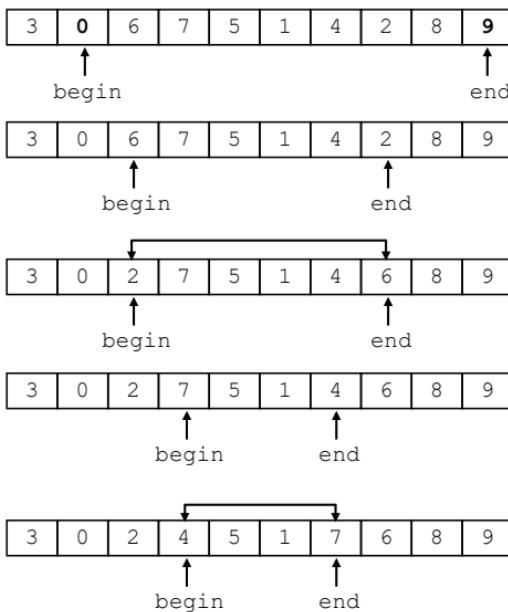
O *pivot* armazena o valor que se encontra na metade do vector. Neste exemplo, o *pivot* é igual a 5. Os valores de *begin* e *end* encontram-se no primeiro e no último índice:

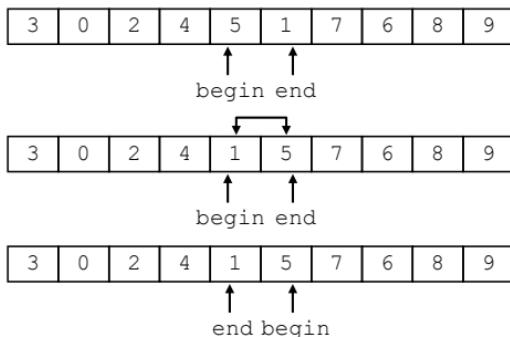


O índice de *begin* avança para a direita até encontrar um elemento maior que o *pivot*, enquanto *end* avança para a esquerda até encontrar um valor menor que o *pivot*:



Nesse momento, o par de valores é trocado e o processo recomeça:





Quando o valor do *end* é menor que o valor do *begin* o processo de separação terminou e inicia-se a resolução dos subproblemas.



O valor inicial do *pivot* é determinante para a dimensão dos dois vectores resultantes. Um valor demasiado grande ou demasiado pequeno produz subvectores muito diferentes, o que resulta numa perda de desempenho. Porém, não existe um método rápido para determinar o melhor valor para o *pivot* sem prejudicar o desempenho geral do algoritmo. O valor no meio do vector é um bom candidato porque em muitas situações o vector a ordenar já se encontra parcialmente ordenado, o que significa que o valor que se encontra a meio do vector é, provavelmente, próximo da média. Outro método consiste em considerar três valores (normalmente, o primeiro, o último e o do meio) e escolher a mediana dos três.

A complexidade no melhor caso (os dois subvectores são separados exactamente ao meio) é dada pela função:

$$T(n) = \begin{cases} \Theta(1) & , n \leq 1 \\ 2.T(n/2) + \Theta(n) & , n > 1 \end{cases}$$

Pelo método mestre,  $T(n) \in O(n \log(n))$ .

No caso médio, a separação tende a ser mais desequilibrada. Por exemplo, para  $n=4$  (o vector tem quatro elementos e assumindo que o *pivot* fica na primeira posição) pode ocorrer entre o primeiro e o segundo, entre o segundo e o terceiro, entre o terceiro e o quarto ou depois do quarto, cada separação com probabilidade  $1/4$ . Assim,  $T(4) = 1/4.(T(0)+T(3)) + 1/4.(T(1)+T(2)) + 1/4.(T(2)+T(1)) + 1/4.(T(3)+T(0)) + \Theta(n)$ . Em termos gerais e simplificando a expressão anterior:

$$T(n) = \begin{cases} \Theta(1) & , n \leq 1 \\ \frac{1}{n} \left( \sum_{i=0}^{n-1} 2.T(i) \right) + \Theta(n) , & n > 1 \end{cases}$$

Obtemos  $T(n) \in O(n \cdot \log(n))$ .

O pior caso (um dos subvectores tem uma dimensão igual à do vector inicial menos um) é dado pela função:

$$T(n) = \begin{cases} \Theta(1) & , n \leq 1 \\ T(n-1) + \Theta(n) , & n > 1 \end{cases}$$

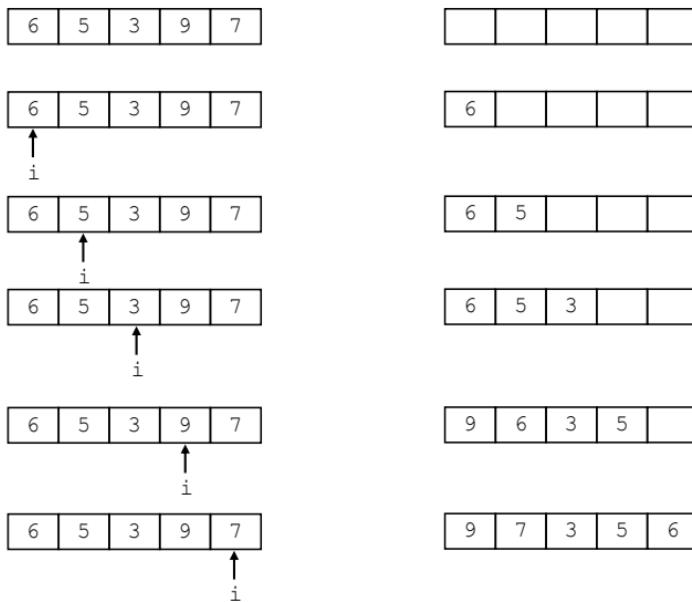
Cuja complexidade é  $O(n^2)$ . Porém, se os valores iniciais do vector forem distribuídos aleatoriamente, a probabilidade de ocorrer o pior caso decresce exponencialmente com a dimensão do vector. Na prática, o algoritmo *quicksort* é normalmente mais rápido que outros algoritmos de complexidades  $O(n \cdot \log(n))$  no pior caso.

A estrutura recursiva do *quicksort* é um pouco pesada se os vectores a ordenar são pequenos (cerca de trinta elementos ou menos). Nesses casos, mesmo algoritmos considerados lentos, como o *insertsort*, são mais rápidos. É possível aproveitar este conhecimento para combinar os dois algoritmos: quando se inicia uma nova ordenação (ou uma das invocações recursivas do *quicksort*), determinar se o vector tem menos de 30 elementos, se sim executa o *insertsort*, caso contrário executa o *quicksort*. Testes com vectores de dezenas de milhares de elementos apontam para ganhos de desempenho na ordem dos 20% em relação ao *quicksort* padrão.

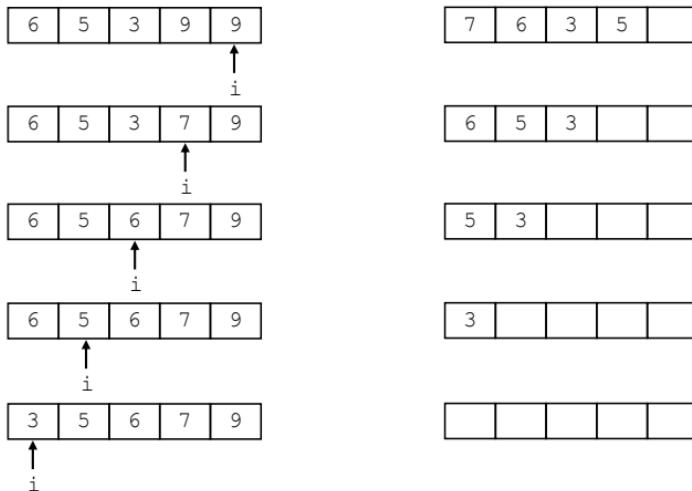
## O algoritmo Heapsort

A estrutura de dados amontoado, descrita na secção 17.4, possui operações de inserção e remoção de complexidade  $O(\log(n))$ . No exemplo seguinte, usámos um amontoado (com representação estática).

Inicialmente, cada valor do vector inicial é inserido no amontoado:



Quando o amontoado contém todos os elementos do vector removem-se as sucessivas raízes do amontoado (i.e., os maiores valores) e colocam-se os valores da última para a primeira posição:



Utilizar um amontoado para armazenar os valores do vector (de dimensão  $n$ ) requer  $n$  inserções (cuja complexidade total é  $O(n \cdot \log(n))$ ) seguido de  $n$  remoções (dos elementos progressivamente maiores) para preencher o vector ordenado (novamente com complexidade total de  $O(n \cdot \log(n))$ ). Assim, a complexidade do *heapsort* é  $O(n \cdot \log(n))$  para todos os casos.

## Escolher o algoritmo certo

Na prática, o *heapsort* é normalmente mais lento que o *quicksort* mas tem a grande vantagem de possuir a complexidade teórica ideal. Em problemas onde um comportamento médio rápido é suficiente é preferível escolher o *quicksort*. Em problemas de tempo real, onde é necessário garantir resultados em qualquer situação é preferível o *heapsort*. Em problemas onde o conjunto de elementos a ordenar é pequeno (menos de 50) convém escolher o *bubblesort* ou o *insertsort* que, pela sua simplicidade, são muito rápidos. Se for necessário ordenar grandes quantidades de informação (armazenada em memória secundária) dispondendo apenas de pouco memória primária, deve-se optar pelo algoritmo de ordenação *mergesort* (ver exercício 4) que necessita de pouca memória a cada instante.

## 19.2 A Classe Sort

Os algoritmos explicados na secção anterior foram reunidos na classe `Sort` constituída por métodos de classe. Esta classe é um repositório de algoritmos de ordenação, da mesma forma que a classe `Math` é um repositório de constantes e funções matemáticas comuns.

```
package dataStructures;  
/**  
 * <p>Titolo: A classe utilitária para ordenação</p>  
 * <p>Descrição: Possui vários algoritmos de ordenação sobre  
 *               vectores de objectos comparáveis.</p>  
 */  
  
public class Sort {  
    // Não se pode construir objectos desta classe!  
    private Sort() {}  
}
```

O construtor privado impede a criação de objectos.

```
private static void swap(Object[] v, int i, int j){  
    Object tmp = v[i];  
    v[i] = v[j];  
    v[j] = tmp;  
}
```

O método `swap()` é utilizado pelos algoritmos de ordenação e corresponde à troca dos dois objectos do vector situados nos índices `i` e `j`.

```
/**  
 * Algoritmo insertion sort, complexidade O(|v|^2)  
 * @param v O vector de elementos a ordenar  
 */  
public static void insert(Comparable[] v) {  
    int i,j;  
    Comparable tmp;  
    for (i=1;i<v.length;i++) {  
        tmp = v[i];  
        for(j=i;j>0 && v[j-1].compareTo(tmp)>0;j--)  
            v[j] = v[j-1];  
        v[j] = tmp;  
    }  
}
```

Assume-se (neste método e nos seguintes) que os objectos a ordenar implementam a interface `Comparable`.

```
/**  
 * Algoritmo selectionsort, complex. O(|v|^2)  
 * @param v O vector de elementos a ordenar  
 */  
public static void selection(Comparable[] v) {  
    int least;  
    for (int i=0;i<v.length-1;i++) {  
        least = i;  
        for(int j=i+1;j<v.length;j++)  
            if (v[j].compareTo(v[least])<0)  
                least=j;  
        swap(v,i,least);  
    }  
}  
/**  
 * Algoritmo bubblesort, complexidade O(|v|^2)  
 * @param v O vector de elementos a ordenar  
 */  
public static void bubble(Comparable[] v) {  
    for (int i=v.length;i>0;i--)  
        for(int j=1;j<i;j++)  
            if (v[j].compareTo(v[j-1]) < 0)  
                swap(v,j,j-1);
```

```

}

/** 
 * Método auxiliar do quicksort
 * @param v O vector de elementos a ordenar
 */
private static void quicksort(Comparable[] v,
                             int begin, int end) {
    int         lower  = begin,
                higher = end;
    Comparable pivot;
    if (end>begin) {
        // O pivot, arbitrariamente, é o do meio
        pivot = v[(begin+end)/2];
        while (lower<=higher) {
            while ((lower<end) &&
                   (v[lower].compareTo(pivot)<0))
                ++lower;
            while ((higher>begin) &&
                   (v[higher].compareTo(pivot)>0))
                --higher;
            // se os indices ainda não se cruzaram...
            if(lower<=higher) // ...trocar os elementos
                swap(v,lower++,higher--);
        }
        if(begin<higher)
            quicksort(v,begin,higher);
        if(lower<end)
            quicksort(v, lower, end);
    }
}
/** 
 * Algoritmo quickSort, complexidade O(|v|*log(|v|))
 *                               no caso médio
 * @param v O vector de elementos a ordenar
 */
public static void quick(Comparable[] v) {
    quicksort(v,0,v.length-1);
}
/** 
 * Algoritmo heapSort, complexidade O(|v|*log(|v|))
 * @param v O vector de elementos a ordenar

```

```
/*
public static void heap(Comparable[] v) {
    VHeap h = new VHeap();
    for(int i=0;i<v.length;i++)
        h.insert(v[i]);           // constrói o amontoado
    for(int i=v.length-1;
        !h.isEmpty();           // os maiores colocam-se
        h.remRoot(),i--)        // no fim do vector
        v[i] = (Comparable)h.root();
    }
} //endClass Sort
```

---

## Exercícios

1. Simular a execução de cada um dos algoritmos apresentados com as sequências: (i) 4 7 5 8 8 2, (ii) 5 4 3 2 1, (iii) 6 7 8 9 10 e (iv) 1 1 1 1 1.
2. Alterar os métodos da classe `Sort` para devolverem o número de trocas e o número de comparações durante a ordenação de um vector. Comparar os resultados de vectores inicializados com valores aleatórios (de inteiros, por exemplo) com dimensões de: (i) 100 elementos, (ii) 1000 elementos, (iii) 10000 elementos.
3. Implementar o método de ordenação *quicksortTurbo* baseado no *quicksort* mas quando a dimensão dos subvectores é menor ou igual a trinta é utilizado o *insertsort*.
4. Existe um algoritmo de ordenação denominado *mergesort*, que usa a técnica de “dividir para conquistar” na ordenação de um vector. O algoritmo separa o vector inicial em duas partes iguais e invoca-se recursivamente para ordenar esses dois subvectores (a base da recursão ocorre quando o vector tem dimensão 1). A construção da solução passa por fundir (do inglês, *merge*) os dois subvectores no vector resultado. Pretende-se que (i) implemente este método em Java, (ii) verifique que a complexidade do algoritmo (para todos os casos) é dada pela seguinte função  $T \in O(n \log(n))$ .

$$T(n) = \begin{cases} \Theta(1) & , n \leq 1 \\ 2.T(n/2) + (n-1) & , n > 1 \end{cases}$$

- (iii) repita os exercícios 1 e 2 para este algoritmo (iv) discuta o custo do *mergesort* em termos da memória necessária, (v) justifique porque o *mergesort* é um bom algoritmo para ordenar valores armazenados em memória secundária quando há pouca memória primária disponível para executar o processo de ordenação.

# **20 – Programação Dinâmica**

---

No capítulo 5 apresentou-se a recursão como mecanismo de resolução de problemas que podem ser decompostos em subproblemas e onde seja possível construir a solução final a partir das soluções desses subproblemas. A estrutura da solução, como referido nesse capítulo, pode ser descrita por:

```
problema(P) :  
    se o caso base B responde a P  
        devolver B  
    senão  
        decompor P em P1, ..., Pn  
        R1 = problema(P1)  
        ...  
        Rn = problema(Pn)  
        R = construir resposta com R1, ..., Rn  
        devolver R
```

Porém, existem casos onde a resolução dos subproblemas envolve a repetição de certos cálculos. Observámos na secção 5.1 que a definição recursiva da função de *Fibonacci* possuía este problema: para calcular  $\text{Fib}(5)$  é necessário calcular o valor de  $\text{Fib}(4)$  e  $\text{Fib}(3)$ . Mas para calcular  $\text{Fib}(4)$  é também necessário calcular  $\text{Fib}(3)$ . Se não tomarmos em conta este problema, o algoritmo recursivo resultante terá complexidade exponencial. Para o problema de *Fibonacci* é possível construir um algoritmo igualmente recursivo mas de complexidade linear. Um enorme ganho de desempenho.

A **programação dinâmica** (do inglês, *dynamic programming*) refere uma técnica de optimização para problemas com soluções recursivas onde os subproblemas possuem cálculos comuns (partilham subsubproblemas). Normalmente a abordagem recursiva directa nestes casos produz um algoritmo de complexidade exponencial.

A técnica da programação dinâmica é criar, a partir dos casos base, um conjunto de soluções usadas para criar novas soluções (de instâncias progressivamente mais complexas do problema) até chegar à solução do problema final. Esta técnica tem um custo em memória: é necessário armazenar as soluções dos subproblemas já resolvidos (na maior parte dos casos este custo é aceitável). As soluções são armazenadas em tabelas construídas dinamicamente (a palavra programação em “programação dinâmica” tem a ver com a construção destas tabelas, não com os respectivos algoritmos).

Ainda o exemplo da função de *Fibonacci*: numa solução baseada nesta abordagem, as soluções são armazenadas num vector de inteiros. Iniciar-se-ia nos casos base: Fib(0) e Fib(1) e progressivamente seriam calculados e armazenados Fib(2), Fib(3), Fib(4) e finalmente Fib(5). Não são necessárias repetições dos cálculos pois cada solução intermédia é calculada uma só vez e armazenada no vector.

Um exemplo que reflecte precisamente esta política na procura das soluções:

```
public long fib(int n) {  
    long[] sols = new long[n+2];  
    sols[0] = 1;  
    sols[1] = 1;  
    for(int i=2;i<=n;i++)  
        sols[i] = sols[i-2] + sols[i-1];  
    return sols[n];  
}  
...  
System.out.print(fib(60));  
□ 2504730781961
```

No caso particular da sequência de *Fibonacci* basta armazenar as duas últimas soluções como se verifica no algoritmo iterativo da página 111 (que necessita, assim, de menos memória).

## Memorização

A **memorização** (do inglês, *memoization*) é uma técnica baseada na programação dinâmica que guarda as soluções já encontradas. Para cada nova instância do problema é verificado primeiro se a solução já existe, evitando, assim, a repetição dos cálculos.

Esta técnica inicializa o vector (ou matriz) de subsoluções a um valor especial (aqui denominado UNKNOWN) que representa o desconhecimento da resposta para um dado problema. Ainda no exemplo anterior:

```
private final int UNKNOWN = -1;  
  
public long fib(int n) {  
    long[] sols = new long[n+1];  
    for(int i=0;i<=n;i++) // no início, não se conhece  
        sols[i] = UNKNOWN; // qualquer solução  
    return fib(sols,n);  
}  
  
public long fib(long[] sols, int n) {  
    if (sols[n] != UNKNOWN) // já foi calculado?  
        return sols[n];  
    if (n==0 || n==1) // base da recursão  
        return sols[n] = 1;  
    return sols[n] = fib(sols,n-1) + fib(sols,n-2);  
}
```

A programação dinâmica aplica-se a problemas com as seguintes características:

- O problema pode ser decomposto em subproblemas;
- Os subproblemas partilham subsubproblemas; e
- A solução óptima pode ser composta a partir de subsoluções óptimas.

O primeiro ponto refere que deve ser possível definir recursivamente o problema. O segundo ponto restringe-se aos problemas onde a definição recursiva produz um algoritmo com complexidade exponencial. O terceiro e último ponto descreve um ingrediente necessário para que a programação dinâmica funcione: a capacidade de construir um conjunto de soluções óptimas do mais simples para o mais complexo *utilizando somente as subsoluções óptimas*. Caso contrário será necessário explorar todas as possibilidades não óptimas, o que resulta num algoritmo de complexidade exponencial.

Seguem-se outros exemplos de aplicação da programação dinâmica.

## 20.1 Jogo de *Bachet*

No jogo de *Bachet* existem N pedras numa mesa e cada um dos dois jogadores retira, no seu turno, de 1 até K pedras. Ganha o jogador que tiver jogado pela última vez. Não é difícil verificar que as posições em que um jogador perde são aquelas em que na mesa está um número de pedras divisível por K+1. Por exemplo, para N = 20 e K = 6, o primeiro

jogador ganha se retirar 6 pedras, já que sobram 14 pedras (e 14 é divisível por  $6+1=7$ ). Assim, o primeiro jogador ganha se o valor inicial de N não for divisível por K+1.

Consideremos uma variante deste jogo<sup>34</sup> em que existe um conjunto limitado de pedras que se podem retirar (onde se inclui o número 1 para garantir que o jogo termine). No exemplo anterior, em vez de se poder retirar de 1 a 6 pedras, pode-se, por exemplo, retirar somente 1, 3 ou 4 pedras. Com 20 pedras iniciais ganha o primeiro ou o segundo jogador?

O seguinte método mostra uma abordagem que testa todas as possibilidades:

```
/**  
 * Verifica se o proximo jogador ganha  
 * @param N      O numero de pedras inicial da mesa, N>0  
 * @param moves Quais as jogadas possiveis (inclui 1)  
 * @return TRUE se o proximo jogador ganha, FALSE c.c.  
 */  
public boolean bachel(int N, int[] moves) {  
    for(int i=0;i<moves.length;i++)  
        if (N==moves[i] ||  
            (N>=moves[i] && !bachel(N-moves[i],moves)))  
            return true;  
    return false;  
}
```

Para cada jogada possível (armazenadas no vector `moves[]`) verifica-se se o próximo jogador ganha ou perde. Se não existir uma jogada que leve à vitória, então o jogador actual perde.

Um exemplo e o resultado:

```
int[] m = {1, 3, 4};  
System.out.print((bachel(20,m)?"1°":"2°") + " ganha");  
□ 1° ganha
```

Porém, este algoritmo possui complexidade exponencial. Certas combinações de jogadas levam aos mesmos estados do jogo. A maioria dos cálculos são meras repetições do que já foi calculado (como na versão exponencial da função de *Fibonacci*).

---

<sup>34</sup> Apresentado por Piotr Rudnicki e que pertence ao extenso (e extremamente interessante) arquivo de problemas das competições de programação mantido pela Universidade de Valladolid em <http://acm.uva.es>.

Uma abordagem baseada na programação dinâmica parte de instâncias mais simples do problema ( $N=1,2,3,\dots$ ) e compõe as soluções de problemas progressivamente mais complexos até atingir o  $N$  do problema inicial.

Considere  $s[J]$  o vector a construir contendo a informação seguinte: se o primeiro jogador ganha o jogo com  $N=i$  pedras, então  $s[i] = 1$ , caso contrário,  $s[i] = 0$ . Depois de calcular o conteúdo deste vector, a solução do problema inicial encontrar-se-á em  $s[N]$ . Os casos triviais ocorrem quando o valor de  $i$  é igual a um dos valores de  $moves[J]$ : o primeiro jogador retira todas as pedras da mesa ganhando imediatamente.

É possível calcular os restantes valores através do raciocínio seguinte: numa dada posição com  $Q$  pedras na mesa, se o primeiro jogador consegue retirar um número de pedras para que restem  $P$  pedras e o valor  $s[P]$  é 0 (dado considerar-se o subjogo onde o segundo jogador começa com  $P$  peças na mesa) então a posição  $Q$  é uma vitória para o primeiro jogador (logo,  $s[Q] = 1$ ). Caso contrário, (i.e., se  $s[P]$  é 1 para qualquer jogada possível) então a posição  $Q$  é uma derrota para o primeiro jogador.

O algoritmo consistirá em: para cada índice do vector  $s$ , testar todos os movimentos possíveis válidos (não é possível retirar mais pedras que as existentes) para determinar se alguma delas leva à vitória. Obtemos assim, uma nova versão do método:

```
/* Segunda versão do problema */

public boolean bachel(int N, int[] moves) {
    int[] s = new int[N+1];

    for(int i=1;i<=N;i++) { // i: num.de pedras na mesa
        s[i] = 0;
        for(int j=0;j<moves.length;j++)
            if (i>=moves[j] && s[i-moves[j]] == 0) {
                s[i] = 1;
                break;
            }
    }
    return s[N]==1;
}
```

Como a construção do vector depende apenas de um ciclo que itera  $N$  vezes (onde é executado um número fixo de instruções), a complexidade deste algoritmo é  $\Theta(N)$ , o que representa uma melhoria substancial em relação ao primeiro algoritmo.

## Utilizar memorização

Uma outra forma de melhorar o desempenho do método inicial é através da técnica da memorização referida na introdução do capítulo. Antes de calcular uma determinada jogada para  $N$  pedras, verificamos se conhecemos essa resposta. Em caso negativo, faz-se os cálculos devidos e armazena-se essa informação para posterior uso:

```
/* Terceira versão do problema */

public boolean bachet(int N, int[] moves) {
    int[] sols = new int[N+1];
    for(int i=0;i<=N;i++) // inicialização do vector
        sols[i] = UNKNOWN; // de soluções
    return b3(N,moves,sols);
}
```

Este primeiro método inicializa o vector de soluções e executa o método `b3()` que utiliza memorização para calcular a resposta desejada sem repetição de cálculos.

```
public boolean b3(int N, int[] moves, int[] sols) {
    if (sols[N] != UNKNOWN) // se conhecemos a resposta
        return sols[N]==1; // ...devolvê-mo-la
    for(int i=0;i<moves.length;i++)
        if (N==moves[i] ||
            (N>=moves[i] && !b3(N-moves[i],moves,sols))) {
            sols[N] = 1; // guardar para uso futuro
            return true;
        }
    sols[N] = 0; // guardar para uso futuro
    return false;
}
```

## 20.2 A maior subsequência comum

Seja  $X = \langle x_1, x_2, \dots, x_n \rangle$  uma sequência de objectos do mesmo tipo. Uma sequência  $Y = \langle y_1, y_2, \dots, y_m \rangle$  é uma **subsequência** de  $X$  se existe uma sequência de índices crescentes  $x_{i1}, x_{i2}, \dots, x_{im}$  tal que  $y_j = x_{ij}$ , para  $1 \leq j \leq m$ . Por exemplo, para  $X = \langle 1,2,3,4,5 \rangle$  e  $Y = \langle 2,3,5 \rangle$ ,  $Y$  é uma subsequência de  $X$ .

Dados duas sequências  $X$  e  $Y$ , a sequência  $Z$  é uma **subsequência comum** (doravante designada por **ssc**) de  $X$  e  $Y$ , se e só se  $Z$  for subsequência de  $X$  e de  $Y$ . Por exemplo, para  $X = \langle 1,2,3,4,5,6,7 \rangle$  e  $Y = \langle 3,5,7,9,11 \rangle$  a sequência  $Z = \langle 3,7 \rangle$  é uma ssc de  $X$  e  $Y$ .

O objectivo desta secção é apresentar um algoritmo que encontra uma das maiores subsequências comuns entre duas sequências iniciais  $X$  e  $Y$ . Para o exemplo anterior, com  $X = \langle 1,2,3,4,5,6,7 \rangle$  e  $Y = \langle 3,5,7,9,11 \rangle$  a maior ssc é  $\langle 3,5,7 \rangle$ .

A forma directa de resolver este problema é calcular todas as subsequências de  $X$  e verificar se são subsequências de  $Y$ . A maior dessas subsequências é a resposta. A dificuldade é que se  $X$  é uma sequência de  $N$  elementos, o número de subsequências de  $X$

é igual a  $2^N$ . Qualquer algoritmo que utilize esta estratégia possui complexidade exponencial. A abordagem da programação dinâmica permite encontrar, como veremos, um algoritmo de complexidade polinomial.

Considere as sequências  $X = \langle x_1, x_2, \dots, x_n \rangle$  e  $Y = \langle y_1, y_2, \dots, y_m \rangle$ . Designemos o comprimento da maior ssc por  $s[n,m]$ .

- Se os últimos elementos de ambas as sequências são iguais, i.e.,  $x_n = y_m$ , então esse elemento comum pertence à maior ssc de X e Y. Porquê? Vamos assumir que não: a maior ssc seria então  $\langle x_{i1}, x_{i2}, \dots, x_{ik} \rangle = \langle y_{j1}, y_{j2}, \dots, y_{jk} \rangle$ . Se  $x_{ik} = x_n$  ou  $y_{jk} = y_m$  obtemos o mesmo resultado substituindo  $x_{ik}$  por  $x_n$  ou  $y_{jk}$  por  $y_m$ . Se  $x_{ik} \neq x_n$  (ou  $y_{jk} \neq y_m$ ) então existiria uma ssc maior  $\langle x_{i1}, x_{i2}, \dots, x_{ik}, x_n \rangle$  (uma contradição). Assim, se  $x_n = y_m$  o valor de  $s[n,m]$  é igual ao valor de  $s[n-1,m-1]$  (i.e., procurar uma das maiores ssc nas duas sequências sem os últimos elementos) mais um:

$$s[n,m] = s[n-1,m-1] + 1 \quad , \text{ se } x_n = y_m$$

- Se os últimos elementos de ambas as sequências são diferentes, i.e.,  $x_n \neq y_m$ , então a maior sequência poderá terminar com  $x_n$  ou com  $y_m$ , mas não com os dois:

$$s[n,m] = \max( s[n,m-1], s[n-1,m] ) \quad , \text{ se } x_n \neq y_m$$

Esta é a definição recursiva de  $s[m,n]$  com a base da recursão  $s[0,b] = s[a,0] = 0$ . A definição recursiva de  $s[n,m]$  para o caso  $x_n \neq y_m$  é não linear: existem subproblemas que partilham subsubproblemas, por exemplo, o valor de  $s[1,1]$  pode ser usado por  $s[2,2]$ ,  $s[2,1]$  e  $s[1,2]$ . Como não é possível obter a maior ssc sem conhecer as maiores ssc dos subproblemas, a propriedade da solução óptima ser composta a partir de subsoluções óptimas é satisfeita. Com esta informação é possível construir um algoritmo baseado nos pressupostos da programação dinâmica.

Por motivos de clareza, consideraremos apenas objectos da classe `String` (i.e., procuramos a maior subsequência comum entre duas frases). Esta não é uma restrição grave pois os algoritmos apresentados são independentes do tipo de informação utilizado. A construção de  $s[m,n]$  é dada pelo método seguinte:

```
private int[][] lcsMatrix(String a, String b) {
    int[][] s = new int[a.length() + 1][b.length() + 1];
    for(int i=1; i<=a.length(); i++)
        for(int j=1; j<=b.length(); j++)
            if (a.charAt(i-1) == b.charAt(j-1))
                s[i][j] = s[i-1][j-1] + 1;
            else
                s[i][j] = s[i-1][j] >= s[i][j-1] ?
                    s[i-1][j] : s[i][j-1];
    return s;
}
```

Este método, de complexidade  $O(n.m)$ , devolve a matriz com todos os valores  $s[m,n]$  construindo as soluções mais complexas à custa das soluções mais simples.

Por exemplo:

```
String a = "abcdefg";
String b = "efafebg";
int s[][] = lcsMatrix(a,b);
System.out.print(s[a.length()][b.length()]);
```

□ 4

A matriz construída no exemplo anterior:

	e	f	a	e	f	g
a	0	0	0	0	0	0
b	0	0	0	1	1	1
c	0	0	0	1	1	1
d	0	0	0	1	1	1
e	0	1	1	1	2	2
f	0	1	2	2	2	3
g	0	1	2	2	3	4

Mas, até agora, apenas conhecemos a dimensão da maior ssc entre as duas sequências. Como determinar uma solução? Começando no canto inferior direito da tabela e verificando se o valor é maior que os valores imediatamente à direita e acima. Se assim for, concluímos que os elementos dessa linha e coluna pertencem à maior ssc das duas sequências iniciais (o caso  $x_n = y_m$ ). Se não ocorrer, passa-se para a célula à direita ou acima com o maior valor (se ambas as células guardarem valores iguais, é indiferente) e repete-se o procedimento até retroceder à linha ou coluna inicial (i.e., a base da recursão) onde o processo de construção termina.

Segue um método que determina a solução:

```
public String getLcs(String a, String b) {
    return getLcsAux(a,
                      lcsMatrix(a,b),
                      a.length(),
                      b.length());
}

private String getLcsAux(String a, int[][] s,
                        int i, int j) {
    if (i==0 || j==0)
        return "";
    if ((s[i][j] == s[i-1][j-1] + 1) &&
```

```

(s[i][j] > s[i-1][j]) &&
(s[i][j] > s[i][j-1]))
return getLcsAux(a,m,i-1,j-1)+a.charAt(i-1);
else
    if (s[i-1][j] >= s[i][j-1])
        return getLcsAux(a,m,i-1,j);
    else
        return getLcsAux(a,m,i,j-1);
}

```

Para o mesmo exemplo, o caminho percorrido é visualizado na tabela seguinte:

	e	f	a	e	f	g
e	0	0	<b>0</b>	0	0	0
f	0	0	0	<b>1</b>	1	1
a	0	0	0	<b>1</b>	1	1
b	0	0	0	<b>1</b>	1	1
c	0	0	0	<b>1</b>	1	1
d	0	0	0	<b>1</b>	1	1
e	0	1	1	1	<b>2</b>	2
f	0	1	2	2	2	<b>3</b>
g	0	1	2	2	2	<b>4</b>

As células mais escuras correspondem à linha/coluna dos elementos que pertencem à maior ssc.

```

String a = "abcdefg";
String b = "efafefg";
System.out.print(getLcs(a,b));
□ aefg

```

## Utilizar memorização

A próxima resolução substitui o método `lcsMatrix()` por outro que utiliza memorização, sendo este mais próximo da definição recursiva do problema:

```

private int[][] lcsMatrix(String a, String b) {
    int[][] sols = new int[a.length()+1][b.length()+1];
    for(int i=0;i<=a.length();i++)
        for(int j=0;j<=b.length();j++)
            sols[i][j] = UNKNOWN;
    lcsAux(sols,a,b,a.length(),b.length());
    return sols;
}

```

Este primeiro método inicializa a matriz das subsoluções (com o valor UNKNOWN) e invoca o método `lcsAux()` para calcular a solução desejada:

```
private int lcsAux(int[][] sols, String a, String b,
                    int n, int m) {
    if (sols[n][m] != UNKNOWN) // já foi calculado?
        return sols[n][m];
    if (n==0 || m==0)           // base da recursão
        return sols[n][m] = 0;
    if (a.charAt(n-1)==b.charAt(m-1))
        sols[n][m] = 1 + lcsAux(sols,a,b,n-1,m-1);
    else {
        int n1 = lcsAux(sols,a,b,n,m-1),
            m1 = lcsAux(sols,a,b,n-1,m);
        sols[n][m] = n1>m1 ? n1 : m1;
    }
    return sols[n][m];
}
```

Qual das soluções é a melhor? A segunda solução é recursiva enquanto a primeira não é. Isso indica uma estrutura mais leve e aparentemente mais rápida de executar. Porém há dois pontos em favor desta segunda versão: (i) o algoritmo descrito é igual à solução proposta, sendo por isso mais fácil de implementar e testar e (ii) nem todas as subsoluções são calculadas, apenas as necessárias. No exemplo utilizado anteriormente (com as frases “*abcdefg*” e “*efaeefg*”) obtemos a seguinte matriz (os valores desconhecidos são representados pelo ponto de interrogação):

	e	f	a	e	f	g
e	?	0	0	?	?	?
f	?	?	?	?	?	?
a	0	0	0	1	?	?
b	0	0	0	1	?	?
c	0	0	0	1	?	?
d	0	0	0	1	?	?
e	?	?	?	?	2	?
f	?	?	?	?	?	3
g	?	?	?	?	?	4

Cerca de metade das subsoluções possíveis não foram sequer calculadas, o que indica um potencial ganho no desempenho do algoritmo.

## 20.3 Multiplicação de Matrizes

Este problema é descrito da forma seguinte:

*Dadas N matrizes de inteiros,  $A_0, A_1, \dots, A_{N-1}$ , efectuar o produto  $A_0 \times A_1 \times \dots \times A_{N-1}$  com o menor número de multiplicações possível.*

Assume-se que as multiplicações entre as matrizes são válidas, i.e., o número de colunas da matriz  $A_{i-1}$  é igual ao número de linhas da matriz  $A_i$  (para  $1 \leq i \leq N-1$ ).

Dada uma matriz A com dimensão  $m$  linhas por  $n$  colunas (ou seja,  $m \times n$ ) e uma matriz B com dimensão  $n \times p$ , a multiplicação destas matrizes produz uma matriz  $m \times p$ . O algoritmo que calcula a multiplicação executa  $m \cdot n \cdot p$  multiplicações inteiras.

O método seguinte calcula a multiplicação de duas matrizes (assume-se que a matriz resultado c já tem a dimensão correcta e foi inicializada a zeros):

```
public void multMatrix(int[][] a, int[][] b, int[][] c) {  
    for(int i=0;i<a.length;i++)  
        for(int j=0;j<b[1].length;j++)  
            for(int k=0;k<a[1].length;k++)  
                c[i][j] += a[i][k]*b[k][j];  
}
```

Um exemplo de uso:

```
int[][] A = { {1,2,3}, {4,5,6} }; //2x3  
int[][] B = { {2,2,2}, {3,3,3}, {4,4,4} }; //3x3  
if (A[0].length!=B.length)  
    throw new Exception("Dimensões Incompatíveis");  
int[][] C = new int[A.length][B[0].length];  
multMatrix(A,B,C);
```

Existem várias formas de multiplicar N matrizes e o número de operações total depende da ordem como o fazemos. Por exemplo, para  $N=4$ ,  $A_0, A_1, A_2$  e  $A_3$  com dimensões  $5 \times 10$ ,  $10 \times 20$ ,  $20 \times 4$  e  $4 \times 6$  obtemos as soluções seguintes:

$(A_0(A_1(A_2A_3)))$  implica 1980 multiplicações  
 $(A_0((A_1A_2)A_3))$  implica 1340 multiplicações  
 $((A_0A_1)(A_2A_3))$  implica 2080 multiplicações  
 $((A_0(A_1A_2))A_3)$  implica 1120 multiplicações  
 $((A_0A_1)A_2)A_3)$  implica 1520 multiplicações

A solução que desejávamos para este problema particular seria a solução com o menor número de multiplicações:  $((A_0(A_1A_2))A_3)$ .

É possível demonstrar que o número de formas diferentes para multiplicar N matrizes é exponencial em relação a N. Um algoritmo que teste todas as possibilidades possui complexidade exponencial. A programação dinâmica oferece um método alternativo que diminui a complexidade do algoritmo para  $O(N^3)$ .

Para a sequência de matrizes  $A_0, A_1, \dots, A_{N-1}$  designamos os valores dos respectivos números de linhas e colunas por  $r_0 \times r_1, r_1 \times r_2, \dots, r_{N-1} \times r_N$ .

Considere  $m_{i,j}$  o custo mínimo de multiplicar as matrizes  $A_i, A_{i+1}, \dots, A_j$ , com  $0 \leq i \leq j < N$ . O problema mais simples é quando  $i = j$ , i.e., quando só existe uma matriz. Neste caso, o custo é zero (não existem multiplicações a efectuar):  $m_{i,i} = 0$ . Para o caso seguinte (com duas matrizes)  $m_{i,i+1} = r_i \cdot r_{i+1} \cdot r_{i+2}$ . Também aqui não há alternativas pois só existe uma forma de as multiplicar.

Já o caso de três matrizes  $m_{i,i+2}$  é igual ao valor mínimo entre  $m_{i,i+1}$  (multiplica-se a primeira e a segunda) e  $m_{i+1,i+2}$  (multiplica-se a segunda e a terceira) mais o valor da multiplicação final, respectivamente,  $r_i \cdot r_{i+2} \cdot r_{i+3}$  e  $r_i \cdot r_{i+1} \cdot r_{i+3}$ .

Para N matrizes, existem  $N-1$  formas de separar a multiplicação em duas multiplicações de matrizes, para depois efectuar a multiplicação final. Por exemplo, para  $N=4$  o valor de  $m_{1,3}$  é o mínimo entre:

$$\begin{aligned} (A_0).(A_1A_2A_3) \text{ i.e., } m_{0,0} + m_{1,3} + r_0 \cdot r_1 \cdot r_4 \\ (A_0A_1).(A_2A_3) \text{ i.e., } m_{0,1} + m_{2,3} + r_0 \cdot r_2 \cdot r_4 \\ (A_0A_1A_2).(A_3) \text{ i.e., } m_{0,2} + m_{3,3} + r_0 \cdot r_3 \cdot r_4 \end{aligned}$$

Ou seja,  $m_{i,j} = \min_{i \leq k < j} \{ m_{i,k} + m_{k+1,j} + r_i \cdot r_{k+1} \cdot r_{j+1} \}$  para  $i < j$ .

Calcula-se o valor de  $k$  começando pelas soluções mais simples e armazenando-as para serem usadas nas soluções seguintes. Uma vez mais, calculamos as respostas óptimas dos problemas mais básicos para construir soluções progressivamente mais complexas. Quando soubermos o valor de  $m_{0,N-1}$  conhecemos o número mínimo de multiplicações desejado.

O exemplo das matrizes  $A_0, A_1, A_2$  e  $A_3$  com dimensões  $5 \times 10, 10 \times 20, 20 \times 4$  e  $4 \times 6$  seria executado da forma seguinte:

De início, todos os  $m_{i,i}$  são zero:

$i \backslash j$	0	1	2	3
0	0			
1		0		
2			0	
3				0

Na primeira iteração, por exemplo, o valor de  $m_{0,1}$  é igual a  $r_0 \cdot r_1 \cdot r_2 = 5 \cdot 10 \cdot 20 = 1000$ :

i \ j	0	1	2	3
0	0	<b>1000</b>		
1		0	<b>800</b>	
2			0	<b>480</b>
3				0

Na segunda iteração, por exemplo, o valor de  $m_{0,2}$  é igual ao mínimo entre:

$$m_{0,0} + m_{1,2} + r_0 \cdot r_1 \cdot r_3 = 0 + 800 + 5 \cdot 10 \cdot 4 = 1000$$

$$m_{0,1} + m_{2,2} + r_0 \cdot r_2 \cdot r_3 = 1000 + 0 + 5 \cdot 20 \cdot 4 = 1400$$

i \ j	0	1	2	3
0	0	1000	<b>1000</b>	
1		0	800	<b>920</b>
2			0	480
3				0

Finalmente, o valor de  $m_{0,3}$  é igual ao mínimo entre:

$$m_{0,0} + m_{1,3} + r_0 \cdot r_1 \cdot r_4 = 0 + 920 + 5 \cdot 10 \cdot 6 = 1220$$

$$m_{0,1} + m_{2,3} + r_0 \cdot r_2 \cdot r_4 = 1000 + 480 + 5 \cdot 20 \cdot 6 = 2080$$

$$m_{0,2} + m_{3,3} + r_0 \cdot r_3 \cdot r_4 = 1000 + 0 + 5 \cdot 4 \cdot 6 = 1120$$

i \ j	0	1	2	3
0	0	1000	1000	<b>1120</b>
1		0	800	920
2			0	480
3				0

O algoritmo seguinte recebe um vector com os valores  $r_0, r_1, \dots, r_N$  e devolve o número mínimo de multiplicações (no exemplo anterior, 1120) e ainda uma matriz  $s[i][j]$  com a ordem de multiplicação das matrizes. O valor  $s[i][j]$  corresponde ao último produto a ser executado para multiplicar as matrizes  $A_i, \dots, A_j$ . A complexidade deste método é  $O(N^3)$  como se verifica pelos três ciclos encadeados.

```
public long ordMatrix(int[] r, int[][] s) {
    long[][] m = new long[r.length-1][r.length-1];
    for(int DELTA=1; DELTA<r.length-1; DELTA++)
        for(int i=0; i<r.length-1-DELTA; i++) {
            int j = i + DELTA;
            m[i][j] = Long.MAX_VALUE;
```

```
    for(int k=i;k<j;k++) {
        long q = m[i][k] + m[k+1][j] +
                 r[i]*r[k+1]*r[j+1];
        if (q<m[i][j]) {
            m[i][j] = q;
            s[i][j] = k;
        }
    }
}
return m[0][r.length-2];
}
...
int[] r = {5, 10, 20, 4, 6};
int[] s = new int[r.length-1][r.length-1];
System.out.print(ordMatrix(r,s));
□ 1120
```

O método devolve o número de multiplicações da melhor escolha. Para determinar a ordem das multiplicações é necessário consultar a matriz no segundo argumento (inicialmente uma matriz nula). A melhor forma de multiplicar  $A_i, A_{i+1}, \dots, A_j$  consiste em multiplicar  $A_i \times \dots \times A_{s[i][j]}$  por  $A_{s[i][j]+1} \times \dots \times A_j$ . Para resolver o problema repete-se recursivamente este processo até chegar aos casos base (as próprias matrizes) sendo então executada a multiplicação das matrizes para os casos progressivamente mais complexos.

O método seguinte executa este processo recursivo, recebendo um vector  $A[ ]$  com todas as matrizes, a matriz  $s[ ]$  e os índices  $i, j$  que determinam a sequência  $A_i \times A_{i+1} \times \dots \times A_j$  a calcular.

```
public int[][] ordMultMatrix(int[][][] A, int[][] s,
                               int i, int j) {
    if (j>i) { // passo da recursão
        int [][] S = ordMultMatrix(A,s,i,s[i][j]);
        int [][] T = ordMultMatrix(A,s,s[i][j]+1,j);
        int [][] R = new int[X.length][Y[0].length];
        multMatrix(S,T,R);
        return R;
    }
    return A[i]; // caso base
}
```

Um exemplo completo:

```
public static void main(String[] args) {  
    int[][] A0 = { {0,1,2}, {1,2,3} }; //2x3  
    int[][] A1 = { {0,1,2}, {1,2,3}, {2,3,4} }; //3x3  
    int[][] A2 = { {0,1,2,3}, {1,2,3,4},  
                    {2,3,4,5} }; //3x4  
    int[][] A3 = { {1,2,3}, {2,3,4}, {3,4,5},  
                    {4,5,6} }; //4x3  
    int[][] A4 = { {1,2}, {2,3}, {3,4} }; //3x2  
  
    int[] r = {2,3,3,4,3,2}; // 2x3 3x3 3x4 4x3 3x2  
    int[][] s = new int[r.length-1][r.length-1];  
  
    System.out.println(ordMatrix(r,s));  
  
    int[][][] A = { A0, A1, A2, A3, A4 };  
    int[][] R = ordMultMatrix(A,s,0,3);  
  
    System.out.print("{ "); //imprimir matriz resultado  
    for (int i=0;i<R.length;i++) {  
        System.out.print("{ ");  
        for(int j=0;j<R.length;j++)  
            System.out.print(R[i][j] + " ");  
        System.out.print("} ");  
    }  
    System.out.print("}");  
}  
  
□ 78 ↴  
{ { 780 1044 } { 1380 1848 } }
```

---

## Exercícios

1. Se no jogo de *Bachet* quisermos apenas saber quem ganha, sem conhecer a sequência das jogadas vitoriosas, como se pode optimizar a solução apresentada em termos da memória necessária?
2. Dados N tipos de moedas diferentes e um dado valor de troco, determinar se esse troco é possível e (se for) qual o número mínimo de moedas necessário. Por exemplo, com moedas de 2, 5 e 20, o troco 37 é possível com o mínimo de cinco moedas (uma de 20, três de 5 e uma de 2).||
3. Calcule a maior ssc das sequências “0010100101110101” e “000000”.
4. Calcule a maior ssc das sequências “TATACGAAATTG” e “TAAACG”. Quantas soluções possíveis existem?

5. Qual é a melhor forma de calcular o produto das matrizes com as dimensões:  $35 \times 40$ ,  $40 \times 10$ ,  $10 \times 50$ ,  $50 \times 100$  e  $100 \times 1$ ?
6. Se fosse necessário saber qual a pior forma de multiplicar uma sequência de matrizes, o que deveria ser modificado no algoritmo apresentado?
7. A multiplicação de matrizes apenas necessita de uma matriz triangular para armazenar as soluções. Como modificaria o programa para diminuir a memória utilizada?
8. Descreva um algoritmo de complexidade  $O(n^2)$  que determine a maior sequência crescente de números numa sequência de  $n$  números.
9. Dado um inteiro positivo  $n$ , calcule os coeficientes do polinómio  $(x+1)^n$  (*pista*: relembrar-se dos coeficientes binomiais e do triângulo de Pascal).
10. O problema seguinte é designado por Problema do Saco/Mochila (do inglês, *Knapsack Problem*): Um ladrão entra numa loja para roubar material. Na loja existem vários objectos do tipo  $T_1, T_2, \dots, T_n$ , tendo  $T_i$  associado um peso em quilos e um valor em euros  $\langle P_i, V_i \rangle$ . Sabendo que o ladrão carrega no máximo  $k$  quilos, quantas unidades de cada objecto deve roubar para maximizar o valor total? Implemente um algoritmo que, recebendo os dados iniciais, devolva a resposta correcta.

11. Uma encyclopédia de  $n$  tomos tem de ser arrumada em  $k$  prateleiras. Sabendo que  $i$ -ésimo tomo possui  $p_i$  páginas, como distribuí-los pelas prateleiras (sem alterar a ordem dos mesmos) de modo a minimizar o número máximo de páginas da prateleira com maior volume?

Por exemplo, dado 3 prateleiras para guardar 9 tomos com os respectivos números de páginas:

100 200 300 400 500 600 700 800 900

Uma possível forma de os separar em três partes seria:

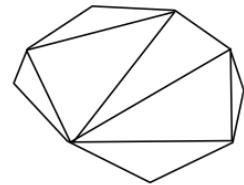
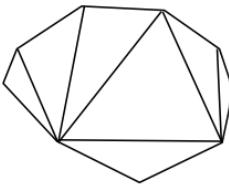
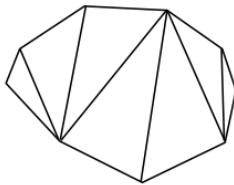
100 200 300 | 400 500 600 | 700 800 900

Com a maior prateleira a guardar 2400 páginas. Porém, a melhor solução é:

100 200 300 400 500 | 600 700 | 800 900

Com o máximo de 1700 páginas na terceira prateleira.

12. Dado um polígono convexo  $P$ , a triangulação de  $P$  é um conjunto de diagonais que não se intersectam que divide  $P$  em triângulos. Seguem três exemplos de triangulação de um mesmo polígono:



O comprimento de uma triangulação é a soma das dimensões dessas diagonais. Encontrar um algoritmo que, dado um polígono convexo, calcule a menor triangulação possível.



# Anexo A – Programas Java

---

Este anexo apresenta parte das funcionalidades disponibilizadas pelo SDK (*Software Development Kit*) o pacote de desenvolvimento fornecido pela empresa *Sun*, de modo que o leitor possa compilar e usar os programas desenvolvidos ao longo do processo de aprendizagem.

## A.1 A máquina virtual

Uma das principais razões para o sucesso do Java é o facto da linguagem ter sido construída para ser independente do sistema operativo usado. Um programa Java compilado num ambiente *Windows* pode funcionar sem problemas num ambiente *Linux* ou *Macintosh* entre outros.

Na abordagem convencional, o programa é compilado para o ambiente (o sistema operativo e o processador usado) onde o programa foi desenvolvido. Por exemplo, uma aplicação compilada em *Windows* não funciona em *Linux* e vice-versa. Para funcionar num ambiente diferente tem de se compilar o código fonte noutro compilador. Também no mundo da informática as linguagens possuem dialectos e mesmo os compiladores assumem abordagens diferentes, provocando mais atrasos na construção de um segundo programa executável, implicando maiores custos e menor compatibilidade.

No Java este problema foi eliminado através da introdução de um passo intermédio. Todos os compiladores Java (designados `javac`) traduzem o programa para uma linguagem de baixo nível designada linguagem de *bytecodes*. Esta linguagem é reconhecida por uma máquina virtual (de nome `java`) incluída em todos os pacotes de desenvolvimento que executa programas em *bytecodes* independentemente do sistema operativo. O ambiente de

programação deixa de ser um parâmetro no processo de desenvolvimento das aplicações. Como a Sun disponibiliza as ferramentas necessárias para a maioria dos sistemas operativos existentes é promovida a compatibilidade e a migração dos programas entre os diversos ambientes.

## A.2 O SDK

O SDK existe em três versões<sup>35</sup>:

- J2SE (*Java 2 Standard Edition*) – onde são incluídas todas as ferramentas e bibliotecas de classes para desenvolver aplicações normais e *applets* (os objectos executados dentro dos *browsers*).
- J2EE (*Java 2 Enterprise Edition*) – para desenvolver aplicações maiores (como aplicações para servidores). Esta versão é executada por cima do J2SE.
- J2ME (*Java 2 Micro Edition*) – para desenvolver aplicações em sistemas menores como os computadores de mão ou aplicações *wireless*.

Entre as ferramentas que o SDK disponibiliza, algumas das mais comuns:

- javac – o compilador de Java. Converte os ficheiros de texto com código fonte Java num programa Java (constituído por *byteCodes*).
- java – a máquina virtual que executa um programa Java (i.e., interpreta e executa os *bytecodes* que definem o programa).
- appletviewer – a aplicação que executa um *applet* Java.
- javadoc – a ferramenta de documentação. Constrói ficheiros HTML a partir de determinadas directivas (ver a secção neste capítulo sobre comentários).
- jar – uma ferramenta que reúne e comprime um conjunto de ficheiros e directórios que definem uma aplicação num arquivo único.

Para compilar um ficheiro executa-se o comando:

```
javac Start.java
```

Quaisquer erros serão indicados pelo programa e o ficheiro executável não é construído. Se nada mais for dito, assume-se que os ficheiros necessários estão na directória actual. O compilador conhece a localização das classes incluídas no SDK. Porém, se existirem ficheiros em directórias diferentes é necessário incluir na variável de sistema CLASSPATH quais as directórias onde o Java deve procurar. Por exemplo:

```
set CLASSPATH = C:\work\test; .           Para o Windows  
setenv CLASSPATH /work/test: .           Para o Linux
```

---

<sup>35</sup> Estes pacotes podem ser obtidos gratuitamente em <http://java.sun.com>

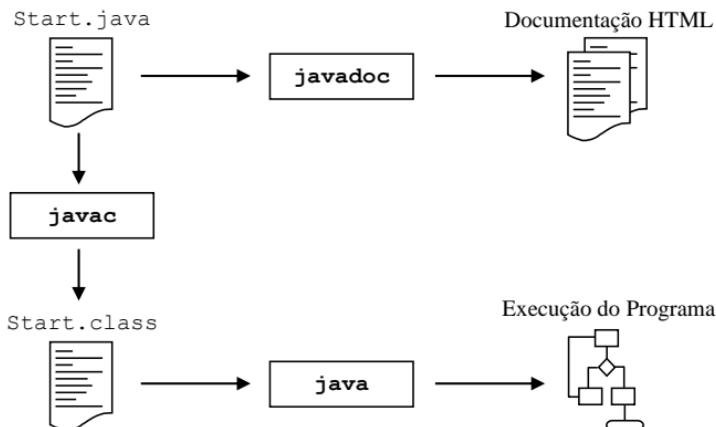
Um exemplo do conteúdo do ficheiro Start.java:

```
class UmaClasse {  
    public void oMeuMetodo(String s) {  
        System.out.println("olá " + s);  
    }  
}  
  
public class Start {  
    public static void main(String[] args) {  
        UmaClasse umObjecto = new UmaClasse();  
        umObjecto.oMeuMetodo("mundo!");  
    }  
}
```

O ficheiro é sempre guardado com o nome da sua (única) classe pública com extensão .java. Depois da compilação é criado o ficheiro com o mesmo nome mas com extensão .class passível de ser executado pela máquina virtual incluída no JDK:

```
javac Start.java  
java Start  
olá mundo!
```

Graficamente, o ciclo de uma aplicação é:



Qual o primeiro método executado pela máquina virtual? Uma aplicação Java é uma colecção de classes (e interfaces) interligadas por relações de herança ou de cliente. Nenhuma classe é especial em relação às outras. Mas é necessário definir um ponto de

partida para a aplicação se iniciar. A classe inicial é dada na linha de comando do programa `java` (no exemplo, a classe `Start`). Esta classe tem de conter um método de classe designado `main()` a partir do qual a aplicação se inicia.

Por exemplo, a linha de comando seguinte inicia-se a partir da classe `Start` com três argumentos iniciais:

```
java Start arg1 arg2 arg3
```

Esta invocação equivale à execução das seguintes instruções:

```
String[] s = { "arg1", "arg2", "arg3" };
Start.main(s);
```

Esta noção de executar um método inicial `main()` possui mais razões históricas que teóricas. Os programas da linguagem C iniciam-se num método com o mesmo nome. Este método de classe é um corpo estranho às aplicações centradas em objectos e pode facilmente resultar numa fonte de confusão. Convencionámos que cada aplicação possui uma classe `Start` cuja única finalidade é conter `main()` e dentro do qual criamos os objectos iniciais da aplicação. Assim, o método `main()` fará parte do seguinte código fonte padrão:

```
public class Start {
    private Start() {} // não se criam instâncias de Start
    public static void main(String[] args) {
        ...
    }
}
```

O construtor privado impede a criação de objectos da classe `Start`. Esta classe apenas serve para iniciar a aplicação.

Um exemplo que utiliza um objecto do tipo lista dentro do pacote `dataStructures` (construído ao longo da 3<sup>a</sup> parte):

```
package dataStructures;
public class Start {
    private Start() {}
    public static void main(String[] args) {
        DList l = new DList();
        l.addBegin(new Integer(2));
        l.addBegin(new Integer(1));
        System.out.print(l);
    }
}
```

```
    }  
}
```

Para visualizar o resultado da execução:

```
☒ javac Start.java  
☒ java Start  
☐ [1,2]
```

Existem aplicações que facilitam a construção de programas através de **ambientes integrados de desenvolvimento** ou IDEs (do inglês, *Integrated Development Environment*). Para a implementação dos algoritmos apresentados ao longo do manual foi utilizado o IDE da Borland *JBuilder* cuja versão pessoal é de utilização gratuita<sup>36</sup>.

### A.3 Pacotes

As classes (e outros componentes) reúnem-se em unidades de *software* denominadas por **pacotes**. Um pacote deve conter somente classes relacionadas a um determinado assunto. Por exemplo, o pacote `java.io` referido no capítulo 7 contém todas as classes que lidam com ficheiros. Para além da vantagem da modularização, um pacote é uma unidade de encapsulamento de informação, dado que as classes de acesso *package* são vistas internamente mas não para o exterior. Os pacotes fornecem um contexto às classes que incluem evitando conflitos entre classes distintas (pertencendo a pacotes diferentes) com nomes iguais.

Para indicar que uma classe pertence a um pacote X coloca-se na primeira linha do ficheiro da classe a declaração seguinte:

```
package X;
```

Em cada ficheiro (e logo, para cada classe pública) só pode existir uma declaração de pacote. Uma classe pertence a um e um só pacote.

É preciso algum cuidado com o nome do pacote para evitar repetições com pacotes já existentes. Um bom método é associar ao pacote o nome de um endereço da Internet que pertença à instituição responsável. Por exemplo, se uma empresa com o *website* comercial `www.empresax.pt` definiu um pacote `utilitarios`, o nome completo do pacote poderia ser:

```
package pt.empresax.utilitarios;
```

Para executar uma classe C do pacote P incluído nas directórias (ou arquivos `jar`) D1, D2 ou D3:

---

<sup>36</sup> [http://www.borland.com/products/downloads/download\\_jbuilder.html](http://www.borland.com/products/downloads/download_jbuilder.html)

```
java -cp "D1;D2;D3" P.C
```

Um pacote pode importar funcionalidades de um outro pacote. Observámos no capítulo 7 que para trabalhar com fluxos de informação é necessário importar o pacote `java.io`. A declaração seguinte permite utilizar a totalidade das classes que o pacote disponibiliza (porém, não inclui eventuais subpacotes):

```
import java.io.*;
```

A partir da declaração é possível usar as classes definidas em `java.io` sem quaisquer prefixos.

Os ficheiros que contêm componentes de um pacote estão organizados num mesmo directório cujo nome deve ser igual ao nome do pacote. Os pacotes podem conter outros pacotes (colocados em subdirectorias).

É igualmente possível descrever totalmente a classe a importar:

```
import java.util.Date;  
  
public class Start {  
    public static void main(String[] args) {  
        Date agora = new Date();  
        System.out.println(agora);  
    }  
}
```

```
Wed Nov 13 16:50:11 GMT 2002
```

Pode-se usar uma classe sem declarar um `import`, mas nesse caso é necessário descrever o pacote a que pertence em cada utilização:

```
public class Start {  
  
    public static void main(String[] args) {  
        java.util.Date agora = new java.util.Date();  
        System.out.println(agora);  
    }  
}
```

O pacote `java.lang` é considerado tão importante que não é necessário escrever o nome completo para aceder às suas classes. Por exemplo, quando se usa a classe `String` não é obrigatório escrever `java.lang.String`.

Existe um conjunto de classes predefinidas organizadas por pacotes. Entre os pacotes mais importantes encontramos:

- `java.lang` – contém as classes relevantes para a linguagem: `Object`, `Math`, `String`, `Number` (onde derivam as classes numéricas como `Integer`, `Double`, `Byte...`), `System`, `Thread...`
- `java.io` – contém as classes que tratam da entrada e saída de dados, muitas das quais foram vistas no capítulo 7.
- `java.net` – contém as classes referentes à comunicação por redes de dados.
- `java.security` – contém as classes que implementam a estrutura de segurança de dados fornecida pelo Java.
- `java.util` – contém múltiplas classes úteis para diferentes tarefas. Existem classes que lidam com colecções de objectos: `Collection`, `Map`, `Set`, `List`, `Vector`, `Stack...`. Existem ainda classes que lidam com datas e tempo: `Date`; com despertadores de processos: `Timer`; com números aleatórios: `Random`; entre outras.
- `java.beans` – contém classes para criar e manipular componentes *JavaBeans*, uma arquitectura de componentes onde é possível exportar configurações, comportamentos e conjuntos de eventos capazes de se interligarem para criar sistemas mais complexos.
- `javax.crypto` – contém classes com rotinas de encriptação de dados.

É aconselhável que o programador tenha uma visão geral das funcionalidades disponibilizadas pelas classes Java. Devido ao grande número de classes disponíveis (mais de mil!) é provável que certas rotinas necessárias possam já estar implementadas.

Com estes conceitos descreve-se um ficheiro e um programa Java:

- Um **ficheiro Java** é um ficheiro de texto contendo código fonte Java com as restrições seguintes: (i) uma directiva `package` opcional; (ii) zero ou mais directivas `import`; (iii) uma ou mais definições de classes, sendo que uma é uma só é pública. O nome do ficheiro deve ser igual ao nome da classe pública com extensão `.java`.
- Um **programa Java** é um conjunto de um ou mais pacotes, em que uma das classes públicas possui o método de classe designado `main()`, implementado com o objectivo de ser o primeiro método executado.

## A.4 Comentários e Documentação

Um programador experiente sabe que passa mais tempo a ler código fonte do que a escrevê-lo. Se adicionarmos o facto de um programa ser constituído por milhares de linhas escritas durante meses ou anos, algum do código acabará por se tornar obscuro e de difícil

interpretação. É conveniente escrever comentários sobre o código fonte de modo a facilitar uma posterior leitura. Em Java existem três tipos de comentário:

- De linha – colocam-se os símbolos `//` com o comentário na mesma linha.
- De bloco – colocam-se os comentários entre os símbolos `/*` e `*/`.
- De documentação – colocam-se os comentários de documentação entre os símbolos `/**` e `*/`.

Os comentários de linha servem para frases curtas e objectivas que referem algum ponto específico do código fonte.

```
if (Math.sqrt(x) > 2)    // x deve ser positivo!
y *= x%2;
```

Os comentários de bloco servem para guardar textos maiores ou para “neutralizar” um determinado conjunto de instruções sem o apagar.

```
/* NOTA! Este código não está a funcionar
if (Math.sqrt(x) > 2)    // x é sempre positivo!
y *= x%2;
*/
```

Não se pode colocar comentários de bloco dentro de outros comentários de bloco. O exemplo seguinte não é aceite pelo compilador Java:

```
/* NOTA! Este código não está a funcionar
if (Math.sqrt(x) > 2) /* x é sempre positivo! */
y *= x%2;
*/
```

Os comentários de documentação servem para documentar classes e métodos. Esta documentação é criada automaticamente quando o programa `javadoc` é executado. Existe um conjunto de parâmetros para estes comentários. Cada um dos comandos é prefixado pelo símbolo `@`. Apresentamos a lista dos comandos mais usados:

- `@param nome texto` – descreve o parâmetro `nome` do método associado.

```
/** @param a coordenada no eixo xx
 *  @param b coordenada no eixo yy
 *
public void Ponto2D(double a, double b) {
    x=a; y=b;
}
```

- `@return` *texto* – descreve o resultado do método.

```
/** @return Devolve verdadeiro se o ponto p está contido no
 *         círculo de raio DELTA e centro neste objecto;
 *         caso contrário, devolve falso.
 */
public boolean equals(Ponto2D p) {
    return Math.abs(x-p.x)<DELTA &&
           Math.abs(y-p.y)<DELTA;
}
```

- `@throws` *nome texto* – descreve uma exceção que o método pode lançar.

```
/** @throws IOException O ficheiro não pôde ser aberto
 */
public void abrir(String s) throws IOException {
    FileOutputStream ficheiroEscrita =
        new FileOutputStream(s);
}
```

- `@author` *nome* – especifica o autor do código fonte.

```
/** @author Elvis Presley
 */
public void cantar(String s) {
    System.out.print("Eu canto " + s);
}
```

- `@version` *número* – especifica o versão actual do código fonte.
- `@since` *versão* – indica quando o método foi implementado pela primeira vez.
- `@see` *marcador* – cria uma referência para outro comentário.

```
/** @see #newDELTA(double D)
 */
public boolean equals(Ponto2D p) {
    return Math.abs(x-p.x)<DELTA &&
           Math.abs(y-p.y)<DELTA;
}
```

- `{@link marcador}` – igual ao `@see` mas a referência fica no meio do texto.

```
/** @param D O novo DELTA da igualdade entre pontos.  
           Consultar {@link #equals(Ponto2D p)}.  
 */  
public void newDELTA(double D) {  
    DELTA = D;  
}
```

É possível usar etiquetas HTML dentro dos comentários de documentação:

```
/** A classe <code>Ponto2D</code> usa o método de  
 * comparação de pontos 2D dentro de uma vizinhança  
 * <b>DELTa</b>  
 */
```

Para criar os ficheiros HTML de documentação:

☒ javadoc ficheiro.java

Considere o exemplo:

```
import java.io.*;  
public class Start {  
    /**  
     * @param args Recebe o nome de um Ficheiro para leitura  
     * @throws IOException Se o Ficheiro não existe!  
     */  
    public static void main(String[] args)  
        throws IOException {  
        long nBytes = 0;  
        FileInputStream f = new FileInputStream(args[0]);  
        while (f.read() != -1) nBytes++;  
        System.out.println("Número = " + nBytes);  
    }  
}
```

Ao executar o comando `javadoc Start.java` são criados documentos HTML com a informação sobre o pacote, num formato padrão comum a todas as documentações Java (a consulta inicia-se no ficheiro `index.html`). Esta padronização, para além de automatizar o processo de documentação, facilita a interpretação a quem ter de ler código fonte de vários programas Java.

Para criar documentação só com componentes públicas:

☒ javadoc -public ficheiro.java

## Contratos e Comentários

Os contratos são colocados em comentários especiais. Se o javadoc for executado, o contrato não aparece dado a aplicação não conhecer etiquetas JML. Para incluir os contratos na documentação deve-se escrever explicitamente (através de etiquetas HTML). Por exemplo, para o método `top()` da interface `Stack`:

```
/**  
 * <b>top()</b>: Devolver o elemento do topo  
 * <p>Contrato:  
 *   <u>Pré-Condições:</u>  
 *   <code>!isEmpty()</code>  
 * </p>  
 * @return uma referência para o elemento do topo da Pilha  
 */  
  
//@ requires !isEmpty()  
Object top();
```

## A.5 Arquivos JAR

É possível que uma aplicação seja constituída por dezenas ou mesmo centenas de classes. No momento da compilação criam-se um igual número de ficheiros `.class`, o que é inconveniente para a distribuição do produto final. A ferramenta `jar` reúne e compacta esses ficheiros (incluído ficheiros de recursos diversos, como imagens ou sons) num único ficheiro de arquivo (em inglês, *jar file*).

Para compactar um conjunto de classes no mesmo directório num único ficheiro de arquivo executamos o programa da forma seguinte (a palavra `cfv` representa um conjunto de opções que auxiliam na criação do ficheiro de arquivo, apresentando a descrição textual do processo):

```
█ jar cfv programa.jar *.class
```

Pode-se igualmente aceder a subdirectorias. No exemplo seguinte adicionam-se todos os ficheiros da directória `imagens`:

```
█ jar cfv programa.jar *.class imagens
```

Um ficheiro arquivo pode ser executado. No exemplo seguinte executa-se o método `main()` da classe `Start` que pertence à aplicação guardada no ficheiro de arquivo `programa.jar`:

```
█ jar -classpath programa.jar Start
```

Um ficheiro de arquivo (que represente um *applet*) pode ser executado a partir de uma etiqueta HTML:

```
<applet archive = "programa.jar"
        code      = "Start.class"
        width     = 200
        height    = 200>
</applet>
```

# Bibliografia Seleccionada

---

Existem múltiplas referências sobre as questões fundamentais da programação. Esta bibliografia (onde se incluem as principais referências da concepção deste texto) é constituída por uma pequena selecção de obras (em inglês) aconselhadas para desenvolver o conhecimento do leitor.

## PARTE I

- *David Watt – Programming Language Concept and Paradigms.* Prentice Hall, 1990. Onde se estuda com pormenor os aspectos mais relevantes referentes a tipos e valores, sistemas de tipos, os paradigmas imperativo, centrado em objectos, funcional e lógico.
- *John Mitchell – Concepts in Programming Languages.* Cambridge University Press, 2003. Um texto que foca o mesmo assunto da referência anterior, i.e., os conceitos centrais da programação, mas com um detalhe e desenvolvimento bastante mais actual.
- *Ken Arnold, James Gosling, David Holmes – The Java Programming Language,* 3<sup>rd</sup> Ed. Sun MicroSystems, 2000. Uma introdução muito pormenorizada da linguagem Java pelos próprios arquitectos da mesma.
- *David Flanagan – Java in a Nutshell,* 3<sup>rd</sup> Ed. O'Reilly, 1999. Uma referência exaustiva da linguagem Java e das bibliotecas padrão.

- *Walter Savitch – Java – An Introduction to Computer Science and Programming*, 2<sup>nd</sup> Ed. Prentice-Hall, 1999. Introdução aos conceitos da programação através do uso da linguagem Java.

## PARTE II

- *Bertrand Meyer – Object-Oriented Software Construction*, 2<sup>nd</sup> Ed. Prentice Hall, 1997. Talvez a referência mais importante e exaustiva sobre o modelo de construção de aplicações centrado em objectos. A 2<sup>a</sup> parte deste manual baseia-se, essencialmente, nos conceitos defendidos por Meyer.
- *David William Brown – Object-Oriented Analysis*, 2<sup>nd</sup> Ed. John Wiley, 2002. Uma análise sobre os principais conceitos da programação centrada em objectos, através da terminologia e conceitos UML.
- *Gary Leavens, Yoonsik Cheon – Design by Contract with JML*, 2003. Um relatório onde os autores apresentam a estrutura e funcionalidade desta ferramenta de desenho por contrato para a linguagem Java.

## PARTE III

- *Thomas Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein – Introduction to Algorithms*, 2<sup>nd</sup> Ed. MIT Press, 2001 – Fornece uma introdução muito completa e um desenvolvimento bastante cuidado sobre o estudo da algoritmia e das estruturas de dados. A abordagem ao tema é mais formal não usando nenhuma linguagem de programação em particular.
- *Adam Drozdek – Data Structures and Algorithms in Java*. Brooks/Cole, 2001. Apresenta um conjunto variado de algoritmos e estruturas de dados baseado na linguagem Java.
- *Kenneth Lambert, Martin Osborne – Java, A Framework for Program Design and Data Structures*, Brooks/Cole, 2000. Apresenta as estruturas de dados mais comuns numa abordagem centrada a objectos e de desenho estruturado.

# Índice Remissivo

---

## A

ábaco, 21  
abstracção, 17, 164  
*abstract*, 174  
acção, 213  
acesso  
    dinâmico, 149  
    qualificado, 76  
    sequencial, 149  
agregação, 170  
álgebra, 61  
algoritmo, 21, 43  
    de Euclides, 103  
    eficiência, 252  
*aliasing*, 82  
ambiguidade, 215  
âmbito, 46  
    contratos, 219  
    excepção, 128  
amontoado, 366, 411  
análise, 161  
apontador, 124

*applet*, 232  
arbitriedade, 188  
argumento, 77  
arquivo JAR, 445  
*array*, 61  
árvore, 327  
    (a,b), 362  
altura, 328  
aridade, 328  
AVL, 359  
B/B+, 365  
de decisão, 404  
de recursão, 109  
Fibonacci, 360  
genérica, 362  
ordenada, 347  
percurso, 329  
profundidade, 328  
asserção, 131, 212  
    ligar e desligar, 133  
    não verificável, 217  
*assert*, 131

assinatura, 78  
assíncrona, 232  
associatividade, 33  
atómico, 239  
atribuição, 30, 45  
atributo, 73  
    de classe, 91  
período de vida, 76,  
    91  
temporário, 182  
*autoboxing*, 100  
AWT, 233

## B

*Bachet*, jogo de, 419  
*backtracking*, 114  
base da recursão, 107  
binária, pesquisa, 346  
*binding*, 198  
bit, 17  
bloco, 46  
*boolean*  
    definição, 23

- operações, 23  
*Boolean*, 97, 98  
*break*, 48, 52  
*bubble sort*, 407  
*buffer*, 136  
    *overflow*, 125  
*Buffered*, classes, 146  
*bugs*, 122  
*byte*, 17  
    definição, 23  
    operações, 25  
*bytecode*, 435
- C**
- C*, linguagem, 35, 86, 89  
*C++*, 35, 86, 88, 124, 204  
cabeçalho, 78  
*callback*, 235  
caminho, 328  
caracteres especiais, 24  
*casting*, 31  
*catch*, 127  
chamada, 232  
*char*  
    definição, 23  
    operações, 24  
chave, 383  
*checked exceptions*, 126  
ciclo infinito, 52  
*class*, 74  
classe, 34  
    *abstracta*, 170, 173  
    atributo de, 91  
    componente de, 91  
    concreta, 170  
    de complexidade, 250  
    definição, 173
- destruição, 87  
envolvente, 98  
estática, 201  
final, 193  
genérica, 38  
interior, 92  
ligação dinâmica, 198  
local, 93, 240  
mensagem, 78  
método de, 91  
nível de acesso, 175  
ocultar, 197  
operador de  
    pertença, 196  
rescrever, 197  
subclasse, 191  
superclasse, 191  
    *this*, 81  
classificação, 187  
cliente, 159  
clonagem  
    profunda, 179  
    superficial, 179  
*clone*, 179  
*Cloneable*, 178  
código  
    binário, 16  
    fonte, 17  
coerção, 31  
*coercion*, 31  
colisão, 387  
comando, 44, 74, 223  
comentário Java, 441  
*Comparable*, 178, 414  
comparação, 403  
compilador  
    erros, 121  
componente, 73  
comportamento, 74
- composição, 170, 202  
condição, 44  
    universal, 211  
configuração, 74  
conjunção, 23  
conjunto, 311  
construtor, 79, 165  
    inicialização, 194  
contador, 51  
*continue*, 53  
contrato, 159, 170, 176, 214  
    âmbito, 219  
    comentário, 445  
    herança, 226  
conversão, 31, 195  
correcção, 122, 193, 210, 216  
    parcial, 211  
    total, 211  
crescimento, 252
- D**
- dangling else*, 48  
*DataInputStream*, 143  
*DataOutputStream*, 143  
*Date*, 102  
declaração, 45  
defeito, 122  
delegação, 235  
*delegation*, 235  
delimitadores, 151  
desenho, 169  
deserialização, 182  
destrutores, 195  
dicionário, 383  
dicotómica, pesquisa, 346  
Dijkstra, Edsger, 212

- direitos, 215  
 disjunção, 23  
 dividir para conquistar, 108, 408, 416  
 documentação, 216, 441  
 domínio, 209  
**double**  
 definição, 23  
 igualdade entre, 28  
 infinitos, 28  
 precisão, 27  
**Double**, 97, 98  
*downcasting*, 195  
*dynamic binding*, 198
- E**
- efeito secundário, 31, 132, 163  
 benevolente, 319  
**Eiffel**, 204, 217  
 encapsulamento, 175  
 endereçamento  
     aberto, 389  
     fechado, 388  
 equivalência, 84  
 erro, 122  
     de compilação, 121  
     de execução, 121  
 espaço, 247  
     de estados, 209  
 especificação, 159  
     assinatura, 176  
     definição, 168  
     descrição implícita, 166  
     desempenho, 161  
     desenho, 161  
     incoerente, 168  
     instrução, 210  
     padrão, 167
- qualidade, 161  
**TDA**, 164, 174  
 estado, 74, 231, 238, 240  
     de programa, 44  
     estável, 220  
     mudança, 231  
 estrutura  
     de controlo, 45  
 estruturas de dados  
     polimórficas, 200  
 evento, 231  
     rato, 236  
 exceção, 125  
     apanhar, 127  
     criar, 203  
     lançar, 126  
     relançar, 130  
     verificável, 126  
**Exception**, 126  
 expressão, 30  
*extends*, 189, 193  
 extensibilidade, 188
- F**
- factorial, 51, 108, 255, 257  
 falha, 122  
 fecho, 94  
**Fibonacci**, 109, 418  
 ficheiro, 16  
**FIFO**, 293  
 fila, 293  
     prioridade, 309  
**File**, 149  
**FileInputStream**, 138  
**FileOutputStream**, 139  
**FileReader**, 140
- FileWriter**, 141  
**final**, 45  
**finalize()**, 89  
**flexible array**, 400, 401  
**float**. ver **double**  
**Floyd**, Robert, 210  
 fluxo, 135  
     binário, 185  
     de objectos, 185  
 folha, 328  
**for**, 51  
 fornecedor, 159  
     exigente, 222  
     tolerante, 222  
 função  
     auxiliar, 167  
     dispersão, 387  
     parcial, 167  
     pesquisa, 389  
 funcionalidade, 168
- G**
- garbage collection*, 88  
 gramática, 164  
 guarda, 44, 49, 212
- H**
- Hánoi  
     torres de, 112  
**hash table**, 387  
**Haskell**, 302  
**heap**, 366  
*heap sort*, 411  
 herança, 29, 170, 173, 188, 201, 202, 226  
     diagrama, 192  
     múltipla, 204  
     simples, 205  
 hexadecimal, base, 25  
**Hoare**

triplo de, 210  
Hoare, Charles, 164, 210  
homeostase, 210  
HTML, 234, 266

**I**

IDE, 439  
identidade, 84  
`if`, 47  
implementação, 169,  
  174  
incoerente,  
  especificação, 168  
infinito, 99  
  ciclo, 52, 269  
  conjunto, 166, 311  
*information hiding*, 175  
initialização, 194  
`InputStream`, 136  
*insert sort*, 405  
`instanceof`, 196  
instância, 74  
instrução, 44  
  âmbito, 46  
  atribuição, 45  
  bloco, 46  
  condicional, 47  
  declaração, 45  
  iteração, 49  
  salto, 52  
  vazia, 53  
  vector, 61  
`int`  
  aritmética módulo,  
    27  
  definição, 23  
  deslocamentos `>>`,  
    `>>>, <<`, 26  
  operações, 25  
  quociente `/`, 26

resto `%`, 26  
`Integer`, 97, 98  
interface, 169, 223  
  capacidade, 178  
  definição, 176  
  sinalização, 179  
interrogação, 74, 223  
  básica, 221  
  derivada, 221  
intervalo, 231  
invariante, 212, 215  
invocação, 232  
iteração, 44  
  guarda, 44  
iterador, 314

**J**

`JAR`, 232, 445  
`JML`, 217, 267  
  `==>`, 218  
  `ensures`, 218  
  `\exists`, 219  
  `\forall`, 219  
  invariant, 218  
  `\old`, 218  
  `requires`, 218  
  `\result`, 218

**L**

`LIFO`, 293  
ligação dinâmica, 198  
linguagem  
  contratos, 162  
  máquina, 15  
  programação, 43  
lista, 36, 37, 261  
  circular, 290  
  duplamente ligada,  
    289

ligada, 278  
literal, 29  
local, 231  
`long`  
  definição, 23  
operações, 25

**M**

máquina de estados, 238  
máquina virtual, 17, 435  
*marker interfaces*, 179  
`Math`, 100  
matriz, 61, 427  
*memoization*, 418  
memória, 21  
  fixa, 16  
  primária, 16, 136  
  secundária, 16, 136  
  tampão, 136  
memorização, 418  
*memory leak*, 125  
mensagem, 78  
*merge sort*, 413, 416  
*method overloading*, 87  
*method overriding*, 197  
método, 76  
  coesão, 86  
  de classe, 91  
  recursivo, 107  
método mestre, 256  
Meyer, Bertrand, 217  
ML, 36, 37, 302  
modulação, 188  
monitor, escrever para,  
  148  
Moore, Gordon, 16  
Murphy, lei de, 121

**N**

negação, 23

- new, 62, 79  
 nó, 275, 327  
 notação  
      $\Omega$ , 253  
     o, 254  
     O, 250  
     polaca, 309  
     prefixa,infixa,sufixa  
         , 330  
 null, 65  
 número  
     aleatórios, 101  
     expansão de Cantor,  
         104  
     expansão prima,  
         104  
     perfeito, 103  
     primo, 103  
     triplo pitagórico,  
         102
- O**
- Object, 179, 203  
 ObjectInputStream  
     , 182  
 objeto  
     ciclo de vida, 239  
     clonagem, 83  
     imutável, 86  
     mutável, 86  
     ouvinte, 235  
 ObjectOutputStream  
     , 182  
 octal, base, 25  
 octeto, 17  
 operação, 73  
 operador ternário ?, 31  
 operator, 332  
 ordenação, 403  
 ou exclusivo, 23  
 OutputStream, 136  
 overflow, 27
- P**
- pacote, 266, 439  
 parâmetro, 77  
 PASCAL, 28, 35  
 passagem  
     referência, 85  
     valor, 85  
 passo da recursão, 107  
 periféricos, 16  
 pesquisa, 346, 390  
 pilha, 293  
 polígono, 173, 191, 226  
     triangulação, 433  
 polymorphism, 198  
 pós-condição, 210, 215  
 power set, 36  
 precedência, 32  
 pré-condição, 210, 215  
 pré-processador, 218  
 princípio da  
     substituição, 226  
 private, 175  
 problema  
     intratável, 252  
 procedimento, 77  
 programa, 16, 43  
     estado, 44  
 programação dinâmica,  
     418  
 progresso, 213  
 projeto, 169  
 PROLOG, 114  
 propriedade, 73  
 protected, 175  
 public, 175
- Q**
- quick sort, 408
- R**
- raiz, 327  
 random access, 149  
 random(), 101  
 RandomAccessFile,  
     149, 150  
 Reader, 136  
 rechamada, 235  
 recolha de lixo, 88  
 recursão, 107, 255  
     árvore de, 109  
     infinita, 108  
     linear, não-linear,  
         110  
 retrocesso, 114  
 terminal, 112  
 referência, 34, 61  
     classe, 73  
 refinamento, 169  
 reflexão, 185  
 registo, 383  
 rehashing, 396  
 reificação, 185  
 relação  
     agregação, 170  
     composição, 170  
     has-a, 170, 202  
     herança, 170  
     is-a, 170, 202  
 representação  
     abstracta, 319  
     concreta, 319  
     dinâmica, 274  
     estática, 274  
     explícita, 174  
 requisitos, 159

responsabilidade, 160  
restrição, 209  
*return*, 53  
reutilização, 188  
robustez, 122, 161  
*Runnable*, 179

## S

saco, 325  
recibos, 325  
saturação, 390  
*Scanner*, 151  
*scope*, 46  
*selection sort*, 406  
semântica, 43, 223  
sequência, 43  
  despesquisa, 389  
serialização, 182  
*serialization*, 182  
*Serializable*, 179  
*short*  
  definição, 23  
sinal, 232  
síncrona, 232  
sintaxe, 43, 164  
sistema, 170, 231  
  operativo, 16  
sobrecarregar, 87  
sobre-especificação, 168  
*Sort*, 413  
*source code*, 17  
*state machine*, 238  
*streams*, 135  
*String*, 94  
subclasse, 191  
sub-especificação, 168  
subsequência, 422  
*super*, 191  
superclasse, 191  
suporte  
  físico, 15

lógico, 15  
*System*, 148

## T

tabela, 383  
  de verdade, 24  
dispersão, 387  
*tail recursive call*, 112  
taxonomia, 187  
**TDA**  
  *binTree*, 331  
  *heap*, 368  
  *list*, 262  
  *queue*, 301  
  *SearchBinTree*,  
    349  
  *set*, 312  
  *stack*, 294  
  *table*, 384  
teclado, ler do, 148  
*templates*, 38  
tempo, 247  
*this*, 81  
*throw*, 126  
*Throwable*, 126  
tipo, 22  
  atómico, 22  
  composto, 34  
  de referência, 34  
enumerado, 28, 315  
  genérico, 167  
  monomórfico, 198  
  paramétrico, 37  
  polimórfico, 198  
  *power set*, 36  
  primitivo, 22  
  recursivo, 36  
  subtipo, 28

tuplo, 34  
união disjunta, 35

**Tipos de Dados**  
  *Abstracto*, 164  
*tokens*, 151  
*toString()*, 96  
*transient*, 182  
triângulo de Pascal, 68  
troca, 403  
*try*, 127  
tuplo, 34

## U

*underflow*, 27  
união disjunta, 35  
*Unicode*, 24  
unidade central de  
  processamento, 15  
*Unit*, 35  
*upcasting*, 195

## V

valor, 22  
  atómico, 22  
  composto, 22  
variável, 29, 167  
  convenção, 29  
  de progresso, 51  
  declaração, 29  
*vector*, 34  
  acesso, 63  
  anónimos, 65  
  circular, 305, 388  
  criação, 62  
  flexível, 400  
  *length*, 64  
  literais, 64  
  multidimensional,  
    66

## ÍNDICE REMISSIVO

---

void, 35

### W

Writer, 136

while, 49, 50

*wrapper classes*, 98