



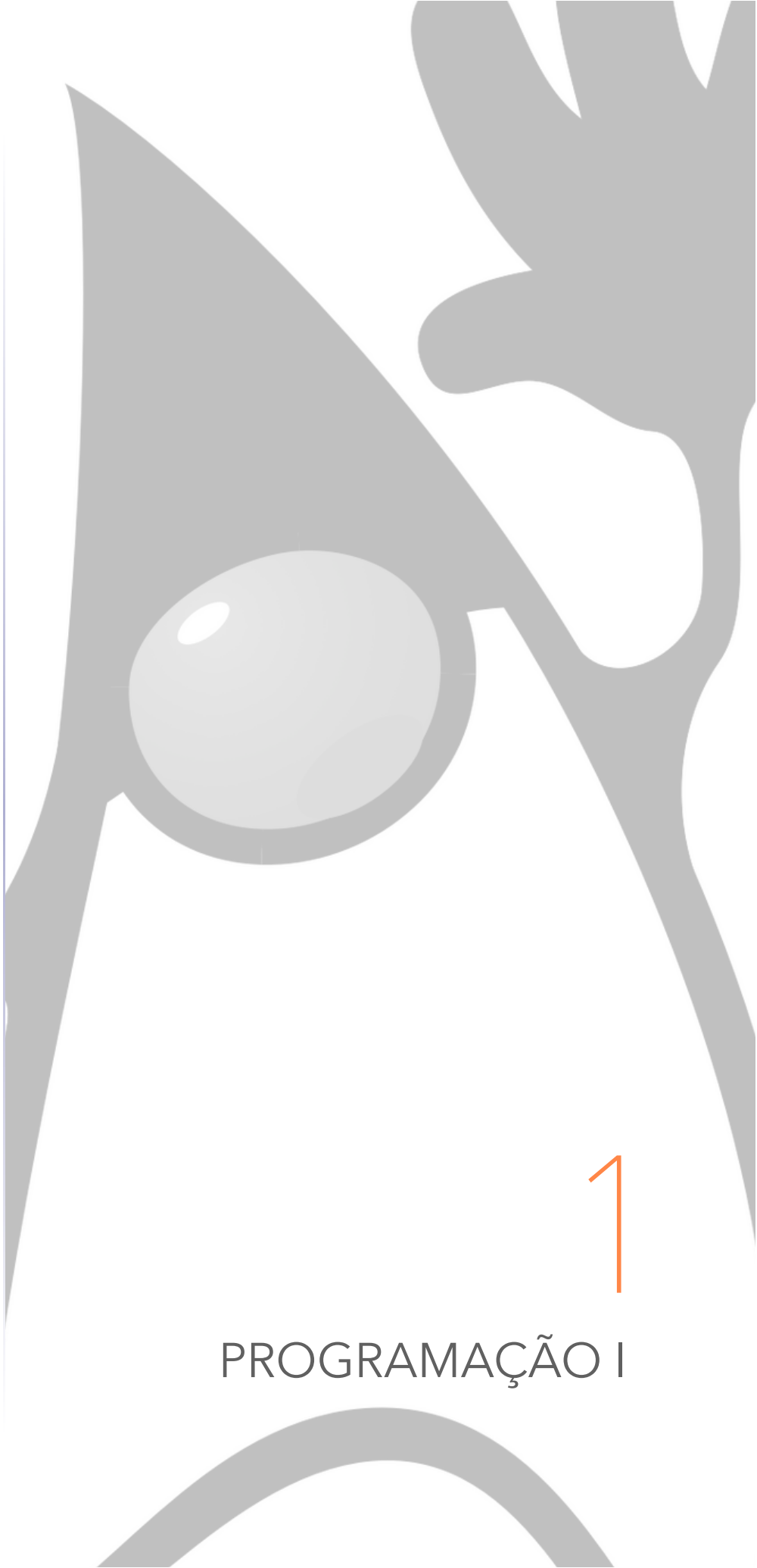
universidade de aveiro  
theoria poiesis praxis

*“I think it's fair to say that personal computers have become the most empowering tool we've ever created. They're tools of communication, they're tools of creativity, and they can be shaped by their user. “*

*Bill Gates*

# PROGRAMAÇÃO I

1



# Atenção!

Todo o conteúdo deste documento pode conter alguns erros de sintaxe, científicos, entre outros... Não estudes apenas a partir desta fonte. Este documento apenas serve de apoio à leitura de outros livros, tendo nele contido todo o programa da disciplina de Programação I, tal como foi lecionada, no ano letivo de 2013/2014, na Universidade de Aveiro. Este documento foi realizado por Rui Lopes.



Os computadores apenas fazem duas coisas: calculam e gravam os resultados desses cálculos, e fá-los especialmente bem. O computador mais normal de todos, o que permanece quieto numa mesa de secretária ou que, pelo contrário, anda sempre de sítio em sítio, por questões pessoais ou de trabalho, faz milhões de cálculos por segundo. É, na verdade, muito difícil conseguir imaginar todos esses cálculos a serem realizados. Mas sempre podemos pensar numa bola a um metro do chão, e que no tempo que a bola cai e toca no chão, o computador faz um bilião de cálculos, ou executa um bilião de instruções.

Em grande parte da história mundial, a computação estava limitada à velocidade de cálculo do cérebro humano e à capacidade de gravação de uma mão a escrever. Felizmente houve desenvolvimentos, mas mesmo assim certos limites ainda permanecem ativos, como o de não conseguir calcular o estado do tempo para amanhã ou depois.

## 1. Introdução aos Computadores

Mas afinal o que é um **computador**? Um computador é um aparelho eletrónico que armazena e processa dados em informação. Um computador contém duas partes essenciais: o **software** e o **hardware**. De uma forma lata, o hardware compreende a parte visível e física de um computador, enquanto que o software compreende as instruções invisíveis que controlam o hardware e que fazem com que este produza tarefas específicas. Um computador deve conter as seguintes estruturas:

**computador**

**software, hardware**

- ▶ Unidade de Processamento Central (CPU)
- ▶ Memória (*main memory*)
- ▶ Dispositivos de Armazenamento (discos e CD's)
- ▶ Dispositivos *input* (como rato ou teclado)
- ▶ Dispositivos *output* (como monitor ou impressora)
- ▶ Dispositivos de Comunicação (como *modems*)

As estruturas de um computador estão interligadas por um subsistema denominado de **barramento** (*bus*). Pode-se pensar no barramento como um conjunto de estradas traçadas entre todas as estruturas de um computador. Nela, transportam-se dados e energia. Num PC (*personal computer*) o barramento está integrado na ***motherboard*** (placa-mãe), estrutura à qual todas as partes de um computador estão anexas.

**barramento**

**motherboard**

### Unidade de Processamento Central (CPU)

A **unidade de processamento central** (*central processing unit*) é essencial a um computador. É esta estrutura que recebe dados da memória e os processa em informação. Usualmente, o CPU tem duas partes principais: uma unidade de controlo (*control unit*) e uma unidade lógica/aritmética (*arithmetic/logic unit*). A unidade de controlo controla e coordena as ações dos outros componentes. A unidade lógica/aritmética produz operações matemáticas (adição, subtração, multiplicação, divisão) e operações lógicas (comparações).

**CPU**

Os CPU's dos dias de hoje são feitos em pequenos chips semicondutores de sílica que contêm milhões de interruptores, estes chamados de **transístores**, com o objetivo de processar dados.

**transístores**

### 3 PROGRAMAÇÃO I

Todos os computadores têm um relógio interno (*clock*) que emite pulsos eletrônicos a um ritmo constante. Estes pulsos são usados para controlar e sincronizar o andamento de operações. Uma maior **velocidade** do relógio faz com que sejam geradas mais intruções a serem executadas num dado intervalo de tempo. A unidade de medida da velocidade do relógio é o **hertz (Hz)**, sendo que 1 Hz equivale a 1 pulso por segundo. Nos computadores da década de '90 a unidade usada era o **megahertz (MHz)**, mas a tecnologia inova constantemente e, atualmente, a velocidade é medida em **gigahertz (GHz)**. Existem, hoje, em 2013, processadores que trabalham à velocidade de 3.7 GHz.

**velocidade**

**hertz (Hz)**

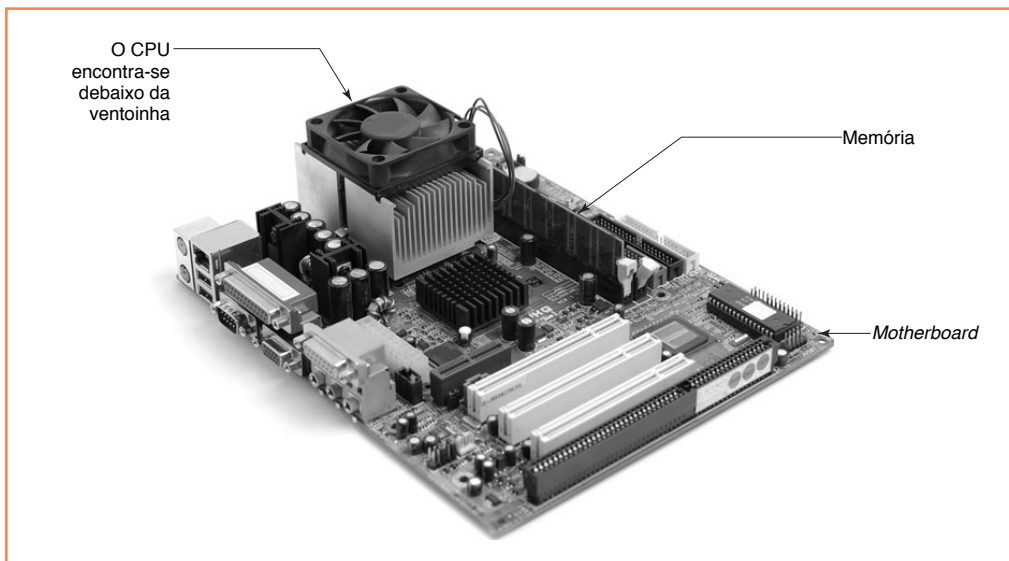
**megahertz (MHz)**

**gigahertz (GHz)**

Originalmente, os processadores eram desenvolvidos apenas com um **core**. O core é a parte do processador que lê e executa as instruções. De modo a aumentar a capacidade de processamento de um CPU, as empresas criadoras desenvolvem, cada vez mais, processadores com múltiplos cores. Um processador multi-core é um processador que contém dois ou mais cores. Hoje os computadores já contém processadores *dual-core* (com dois cores) ou até mesmo *quad-core* (com quatro cores).

**figura 1**

**a motherboard liga todas as estruturas de um computador**



## Bytes

Antes de estudar a memória de um computador, verifiquemos como é que a informação pós-processamento é armazenada.

Tal como referido anteriormente, um computador é um largo conjunto de interruptores. Tal afirmação é válida tanto para o hardware como para o software. Para o software, cada interruptor pode ter duas posições: *on* ou *off*. Guardar informação é tão simples quanto criar uma sequência desses interruptores, cada um, com um sinal. Se um deles estiver ligado, então o valor dele é 1; se estiver desligado, o valor dele é 0. Como já foi visto, estes 0s e 1s são dígitos binários e são chamados de bits.

A unidade mínima de armazenamento num computador é o **byte**. Um byte é um conjunto de 8 bits. Um número pequeno, como o número 3, pode ser armazenado num só byte. Para gravar um número que não caiba num só byte, o computador cria vários bytes.

**byte**

Dados de vários tipos, entre números e caracteres, são encriptados em séries de bytes. As ações de encriptação e desencriptação são desenvolvidas automaticamente,

pela máquina, independentemente do **esquema de codificação** (*encoding scheme*). O esquema de codificação é um conjunto de regras às quais o computador tem de se reger de modo a traduzir caracteres, números e símbolos para dados com os quais ele consiga trabalhar. A maior parte destes esquemas traduz caracteres para sequências numéricas. O esquema mais conhecido é o **ASCII** (*American Standard Code for Information Interchange*). Neste, por exemplo, o carácter *C* é representado por 01000011 num só byte.

**esquema de codificação**

**ASCII**

A capacidade de armazenamento de um computador é medido em bytes e nos seus múltiplos:

- ▶ **Kilobyte (KB)**, aproximadamente igual a 1000 bytes (1024 bytes);
- ▶ **Megabyte (MB)**, aproximadamente igual a 1 milhão de bytes;
- ▶ **Gigabyte (GB)**, aproximadamente igual a 1 bilião de bytes;
- ▶ **Terabyte (TB)**, aproximadamente igual a 1 trilião de bytes.

**kilobyte (Kb)**  
**megabyte (Mb)**  
**gigabyte (Gb)**  
**terabyte (Tb)**

Um típico documento com uma página de processamento de texto tem cerca de 20 KB. Já com 50 páginas pode ter 1 MB, e 1 GB pode conter 50000 páginas.

## Memória

A **memória** (*main memory*) de um computador consiste numa sequência ordenada de bytes para armazenamento de programas e dados com os quais o programa trabalha. Numa imagem, a memória de um computador é, no fundo, a área de trabalho do computador, o escritório, para executar um determinado programa. Um programa e os seus dados devem ser movidos para a memória antes de serem executados pelo CPU.

**memória**

Cada byte entra numa só **célula** e tem um e um só **endereço** (*unique address*). Os endereços servem para localizar os bytes, de modo a que se possa armazenar e receber dados. Sendo que os dados podem ser acedidos por qualquer ordem, a memória também é referida, muitas vezes como **RAM** (*random-access memory*).

**célula, endereço**

**RAM**

Embora não haja direito nem esquerdo num interior de um computador, normalmente, por questões de simplicidade e raciocínio, vêm-se os bits ordenados numa linha. Ao primeiro bit dá-se o nome de **extremo de ordem superior** (*high-order end*) e ao último o nome de **extremo de ordem inferior** (*low-order end*). A atribuição destes nomes justifica-se: no extremo de ordem superior ou bit com maior significado porque se o conteúdo de uma célula for interpretado como uma representação de um valor numérico, este bit será o dígito mais importante no número.

**high-order end**  
**low-order end**

Dado a RAM, o uso de flip-flops acaba por ser ultrapassado, dado que este utiliza novas tecnologias que permitem um armazenamento mais duradouro e seguro. Muitas destas tecnologias necessitam apenas de uma pequena porção de energia para a gravação de bits e dissipam-se muito mais fácil e eficientemente. Em reconhecimento da sua volatilidade, memórias construídas a partir dessa tecnologia são chamadas de **memórias dinâmicas** (*dynamic memory*), que nos ligam aos termos **DRAM** (*Dynamic RAM*) ou **SDRAM** (*Synchronous DRAM*), este último, usado em referência à DRAM que utiliza ainda menos tempo para a troca de dados.

**DRAM**  
**SDRAM**

Os computadores dos dias de hoje conseguem armazenar, no mínimo, 1 GB de RAM, embora, o mais comum seja ter 2 ou 4 GB instalados. Em suma, quanto mais RAM tiver, mais rápido o computador é, mas há limites.

Um byte de memória nunca está vazio, mas o seu conteúdo é passível de ser ignorado pelo programa. A partir do momento em que nova informação entra na memória, a antiga apaga-se.

Tal como os processadores, as memórias são feitas de *chips* semicondutores de sílica, com milhares de transístores instalados na sua superfície. Em comparação com eles, as memórias são menos complexas, mais lentas e menos caras.

## Armazenamento em Massa

Devido à volatilidade e ao espaço limitado de uma memória RAM, a maior parte dos computadores têm memórias adicionais de **armazenamento em massa** (*mass storage*) - ou armazenamento secundário - como discos magnéticos, CD's, DVD's, *flash drives* e cassetes magnéticas. Estes, em relação à *mass memory* têm mais vantagens, sendo menos voláteis, tendo maiores capacidades, baixos custos, e na maior parte dos casos podendo ser retirados (desligados) por razões de arquivo. A grande desvantagem destas tecnologias, é que estas necessitam de movimento (trabalho mecânico), por conseguinte, precisam de mais tempo para escreverem e lerem informação do que a memória principal, onde todas as tarefas são feitas automaticamente.

Durante muitos anos, a tecnologia do **magnetismo** dominou a área do armazenamento em massa. O exemplo mais comum, que ainda hoje se usa, são os **discos magnéticos**, nos quais existe um disco de superfície magnética que gira e guarda informação. Cabeças de escrita e leitura estão posicionadas por cima e/ou por baixo do disco, para quando este girar, cada cabeça faça um círculo, este, chamado de **faixa** (*track*). Ao reposicionar as cabeças de escrita/leitura diferentes faixas concêntricas são passíveis de serem acedidas. Em muitos casos, sistemas de armazenamento em disco consistem em vários discos montados num mesmo eixo, uns por cima dos outros, com determinados espaços entre eles, de modo a que caibam as cabeças de escrita/leitura, entre as superfícies. Em certos casos as cabeças movem-se em uníssonos. Cada vez que estas são reposicionadas, um novo conjunto de faixas - chamado de **cilindro** - torna-se acessível.

armazenamento em massa

magnetismo

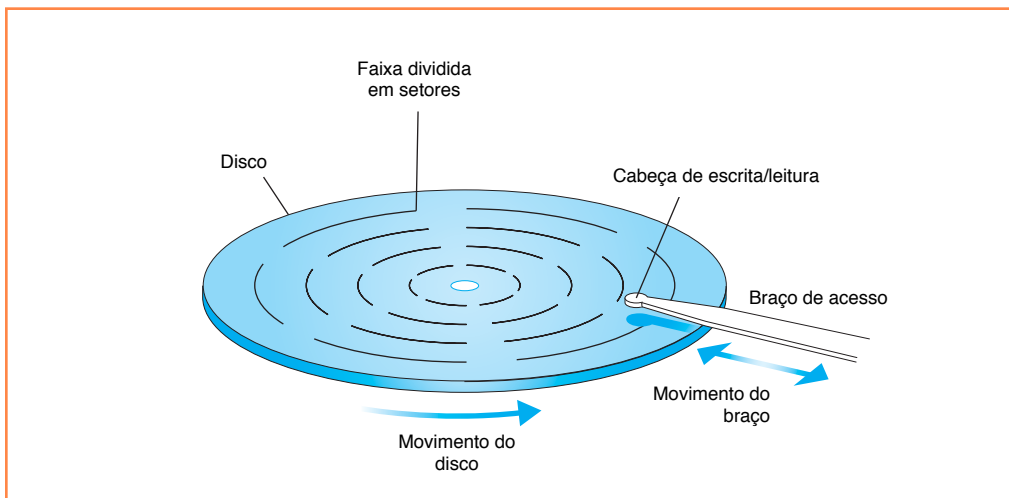
discos magnéticos

faixa

cilindro

figura 2

sistema de armazenamento de disco



Sendo que uma faixa pode conter muito mais informação do que a que nós pretendemos manipular em simultâneo, cada faixa é dividida em pequenos arcos chamados **setores** onde a informação é gravada numa contínua sequência de bits. Todos os setores num disco contêm o mesmo número de bits (as capacidades típicas são entre 512 bytes a uns poucos kilobytes), e no mais simples disco, todas as faixas

setores

apresentam o mesmo número de setores. Por conseguinte, nesse disco, as informações gravadas na faixa mais exterior estão menos compactadas do que na mais interior, sendo que as primeiras são maiores que as últimas. Em discos de alta capacidade de armazenamento, as faixas mais afastadas do centro são as com maior probabilidade de ter mais setores que as interiores, capacidade adquirida pelo uso da técnica de **gravação de bit localizada** (*zoned-bit recording*). Usando esta técnica, várias faixas adjacentes são coletivamente conhecidas como zonas, sendo que cada disco contém cerca de dez zonas. Aqui, todas as faixas de uma dada zona têm o mesmo número de setores e cada zona tem mais setores por zona que as suas inferiores. Desta forma consegue-se maior rendimento no que toca ao aproveitamento de todo o espaço do disco. Independentemente dos detalhes, um sistema de armazenamento em disco consiste em vários setores individuais, cujos podem ser acedidos como sequências independentes de bits.

**gravação de bit localizada**

Diversas medidas são usadas para avaliar o desempenho de um disco:

- ▶ **Tempo de procura** (*seek time*): o tempo necessário para mover uma cabeça de escrita/leitura de uma faixa para outra; **seek time**
- ▶ **Atraso de rotação** ou **Tempo latente** (*rotation delay*): metade do tempo necessário para que o disco faça uma rotação completa, a qual é a média do tempo total para que uns dados desejados girem até às cabeças de escrita/leitura, desde o momento em que estas já se localizam por cima da faixa pretendida; **rotation delay**
- ▶ **Tempo de acesso** (*access time*): a soma do tempo de procura e do atraso de rotação; **access time**
- ▶ **Taxa de transferência** (*transfer rate*): a taxa de velocidade a qual a informação pode ser transportada para ou do disco. **transfer rate**

**note-se** que no caso da gravação de bit localizada, o total de dados que passam as cabeças de escrita/leitura numa única rotação do disco é maior nas faixas exteriores que nas faixas interiores, por conseguinte a taxa de transferência também diferirá consoante a faixa em questão.

**nota**

Um fator que limita o tempo de acesso e a taxa de transferência é a velocidade a que o disco roda. De modo a facilitar essa velocidade as cabeças de escrita/leitura não tocam nunca no disco, mas antes “flutuam” sobre a sua superfície. O espaço entre a cabeça e o disco é tão pequeno que um simples grão de pó pode travar o sistema, destruindo ambos - um fenómeno chamado de **head crash**. Tipicamente, estes discos vêm, de fábrica, selados, dadas essas possibilidades. Só assim é que os discos podem girar a velocidades de várias centenas por segundo, alcançando taxas de transferência que são medidas em MB por segundo.

**head crash**

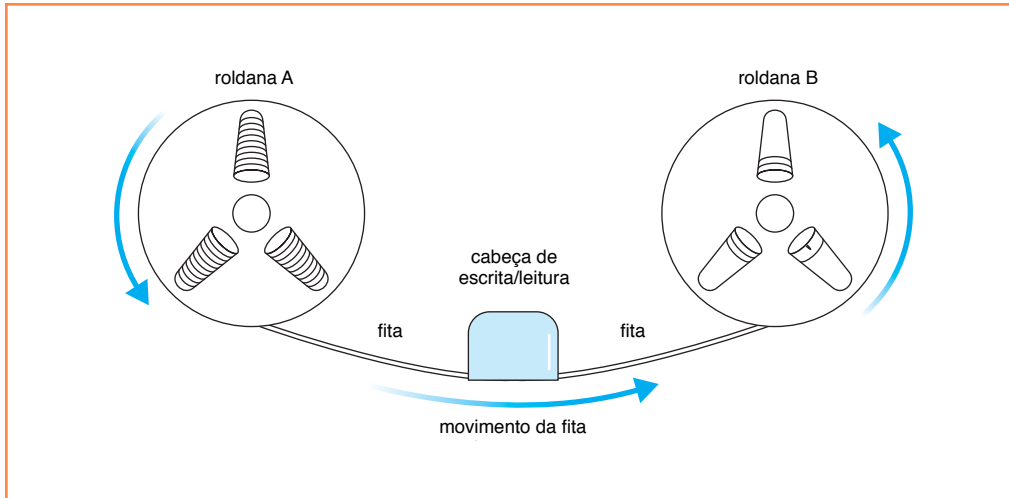
Sendo que os sistemas de armazenamento requerem movimento para as suas operações, estes ficam a perder quando comparados a circuitos eletrónicos. Tempos de atraso, num circuito eletrónico, são medidos em nanossegundos ou menos, enquanto que tempos de procura, de latência e de acesso são medidos em milissegundos. Assim, o tempo necessário para receber algum dado de um sistema de armazenamento em disco pode parecer um “eternidade”, em comparação a um circuito eletrónico que aguarda uma resposta.

Mas os sistemas de armazenamento em disco não são os únicos *mass storage* onde se aplica o magnetismo. Um sistema mais antigo e que o usava era a **cassete magnética**. Neste tipo de sistema, a informação era gravada sobre a superfície de uma

**cassete magnética**

fina tira de plástico enrolado em duas roldanas. Para ler essa informação, estas roldanas eram colocadas sobre um sistema de leitura de cassetes, que incluía uma cabeça de escrita/leitura que passava (por contacto) na fita. Estas fitas, dependendo do seu tamanho poderiam chegar a gravar vários GB.

As grandes desvantagens deste sistema traduzem-se pela sua mecânica. O facto da fita ser transportado de uma roldana até outra é *time-consuming*. É por isso que as cassetes têm um tempo de acesso muito maior que um outro disco, no qual as cabeças de escrita/leitura apenas precisam de se mover um pouco para conseguirem ler/escrever. Por conseguinte, as cassetes não se tornaram um método muito popular entre vários utilizadores.



**figura 3**  
sistema de armazenamento  
em cassette magnética

## Sistemas Óticos

Uma outra categoria de *mass storage* usa a tecnologia **ótica**. Um exemplo mais popular é o **compact disk (CD)**. Estes discos têm 12 centímetros de diâmetro e consistem numa fina camada refletora e uma protetora, na superfície do disco. As informações são gravadas, criando diferentes variações na camada refletora. A sua leitura é feita por via de um feixe de luz (*laser*) que deteta irregularidades na superfície refletora do disco, enquanto este gira.

A tecnologia do CD foi originalmente criada para gravações áudio, num formato conhecido como **CD-DA (compact disk-digital audio)**, e os CD's usados hoje em dia para fins de gravação de dados têm praticamente o mesmo formato. Em particular, a informação nesses CD's está sobre uma única faixa que gira como um antigo disco de vinil. No entanto, contrariamente aos discos de vinil, a faixa gira de dentro para fora. Este disco está dividido em unidades, chamadas de setores, cada um com uma identidade própria e uma capacidade de cerca 2 KB de dados, o que equivale a 1/75 (um, setenta e cinco avos) de um segundo de uma música.

Para maximizar a capacidade de um CD, a informação é gravada numa densidade linear uniforme por toda a faixa, o que significa que mais informação é digitada no *loop* do lado mais exterior da faixa do que no *loop* do lado mais interior da faixa. Assim, mais setores são lidos numa só volta do disco, quando o *laser* está a percorrer o lado exterior da faixa do que quando o *laser* está a ler o lado interior da faixa. Para obter uma taxa de transferência de dados uniforme, os leitores de CD-DA variam a velocidade consoante a zona de leitura do feixe de luz. No entanto, certos

**ótica**  
**compact disk (CD)**

**CD-DA**



CD's de dados giram a velocidades muito altas e constantes, o que permite ajustar uma taxa de transferência, inferior, mas constante.

Como consequência desses factos, os sistemas de armazenamento em CD funcionam melhor quando trabalham com contínuas e longas sequências de dados, o que acontece quando ouvimos música.

Os CD's tradicionais têm capacidades que se compreendem entre 600 e 700 MB. No entanto, os **DVD** (*digital versatile disk*), os quais são construídos de camadas múltiplas e semitransparentes, traduzidas como diferentes superfícies por um feixe de luz de alta precisão, conseguindo capacidades de vários GB. Assim, estes discos são capazes de gravar multimédia diversa.

DVD

Finalmente, os **blu-ray disc (BD)**, usam um laser de cor azul-violeta (contrariamente ao vulgar laser vermelho) para a leitura e escrita de informação com muito mais precisão. Estes discos podem superar um DVD cerca de 5 vezes, sendo estes os portadores de vídeos de alta-definição.

blu-ray disc (BD)

## Dispositivos amovíveis (Flash Drives)

Uma coisa que até agora todos os dispositivos de armazenamento em massa tinham em comum, é que todos eles precisavam de trabalho mecânico para ler ou escrever informação. Isso significa que a velocidade de funcionamento de um trabalho mecânico é inferior à velocidade de funcionamento de um sistema eletrónico. Assim criaram-se as **memórias flash** (*flash memory technology*), nos quais se grava informação simplesmente através de pequenos sinais eletrónicos diretamente para o meio de armazenamento, o que provoca que eletrões sejam movidos para pequenas caixas de dióxido de sílica, alterando as características dos circuitos. Como estas caixas têm a habilidade de preservar os eletrões durante muitos anos, este sistema de *mass storage* pode ser usado para fins de arquivo.

memórias flash

Uma desvantagem deste tipo de sistema é que apagar os dados, repetidamente, faz com que os eletrões tenham que se mover constantemente e o circuito começa a perder as suas qualidades e a danificar-se dado que o dióxido de carbono passa a assumir um carácter polarizante.

Os dispositivos onde esta tecnologia é aplicada têm capacidades que podem rondar, no máximo, três centenas de GB. Também pode ser encontrada em **cartões de memória SD** (*Secure Digital*) ou apenas **cartão SD** e podem ter até uma dezena de GB. Outros espécimes destes cartões podem ser o **SDHC** (*Secure Digital with High Capacity*), que pode atingir os 32 GB, e o **SDXC** (*Secure Digital with Extended Capacity*), que pode atingir 1 TB (1024 GB). Dado os seus tamanhos estes cartões são muito usados em pequenos aparelhos como máquinas fotográficas e telemóveis.

cartão SD

cartão SDHC

cartão SDXC

## Escrita e leitura de ficheiros

A informação que é escrita num sistema de armazenamento em massa é concetualmente agrupada em grandes unidades chamadas **ficheiros**. Um ficheiro típico consiste num documento escrito completo, uma fotografia, um programa, uma gravação áudio. Nós já estudámos que os dispositivos *mass storage* obrigam os ficheiros a serem gravados em muitas e pequenas unidades de bytes. A um bloco de dados conforme às características específicas de um dispositivo dá-se o nome de **gravação física** (*physical record*). Assim, um ficheiro de largas dimensões gravado na nossa unidade de armazenamento consistirá em várias gravações físicas.

ficheiros

gravação lógica

Em contraste à divisão em gravações físicas, um ficheiro muitas vezes já possui divisões naturais determinadas pela informação que contém. Por exemplo, um ficheiro

que contenha todas as informações sobre os empregados de uma empresa, iria ter múltiplas divisões - várias informações por empregado. A estas divisões naturais dá-se o nome de **gravação lógica** (*logical record*).

Gravação lógica muitas vezes consiste em várias unidades chamados de **campos** (*fields*). Por exemplo a informação sobre um empregado em específico pode ter muitos campos: entre eles o nome, a morada, o cargo, ... etc. Também pode ter um campo de identificação, como um número atribuído ou outro carater - a este campo dá-se o nome de **campo-chave** (*key field*). O valor nele inserido chama-se **chave** (*key*).

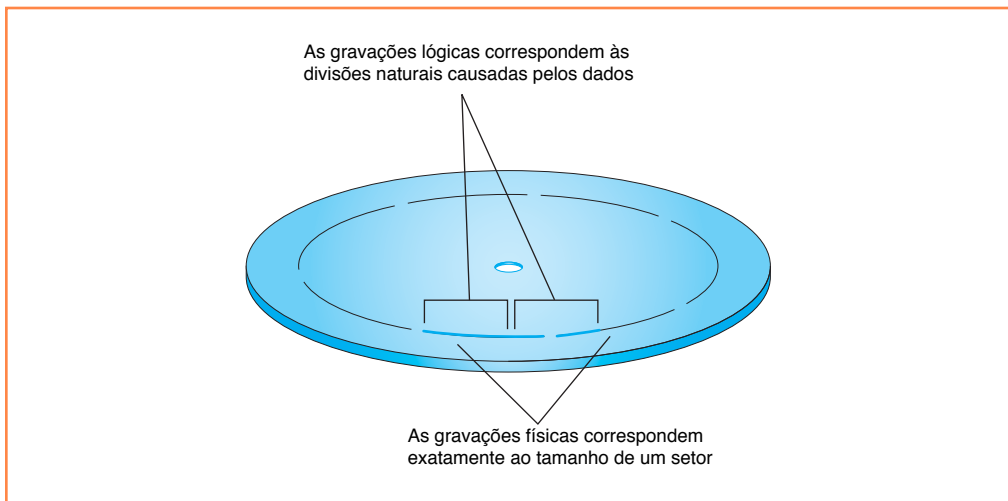
O tamanho de uma gravação lógica quase nunca coincide com o de uma gravação física. Por conseguinte, num caso pode acontecer haver várias gravações lógicas numa física ou uma lógica dividida entre várias físicas. O que acontece após isso é uma fragmentação de informação associada à leitura dos ficheiros. Uma solução muito comum para este problema é deixar de parte a área da *main memory* que é maior para guardar várias gravações físicas e usar a outra parte da memória para reagrupar as gravações. A esta parte da memória que é usada para reagrupar as gravações dá-se o nome de **buffer** (retentor).

**gravação lógica**  
**campos**

**campo-chave, chave**

**buffer**

**figura 4**  
**gravações físicas versus**  
**gravações lógicas num disco**



## Dispositivos de comunicação

Os computadores podem ser ligados a **redes** (*networks*) - meios de comunicação integrados numa máquina de computação. De modo a estarem conectados só alguns dispositivos poderão efetuar a ligação: entre eles os **modems** (*modulator/demodulator*), um **cabo DSL** ou cabo modem, um **NIC** ou um **adaptador wireless**:

**redes**

**modem, NIC, wireless**

- ▶ Um **dial-up modem** usa uma linha de telefone e pode transferir dados até 56000 **bps** (*bits per second*);
- ▶ O **DSL** (*digital subscriber line*) também se conecta através de uma linha telefónica, mas pode transferir cerca de vinte vezes mais rápido que uma conexão *dial-up*;
- ▶ Um **cabo modem** usa a ligação TV e é geralmente mais rápida que a DSL;
- ▶ Um **network interface card** (**NIC**) é um cartão que liga o computador a uma **ligação de área local** (*local area network* - LAN). Estes são mais utilizados nas escolas, universidades e empresas. Por

**dial-up modem**  
**bps (bits per second)**  
**DSL**

**cabo modem**

**NIC**  
**LAN**

exemplo, o modelo *1000BaseT* consegue transferir dados à velocidade de 1000 **mbps** (**milhões de bits por segundo**);

► As ligações **wireless** ou **sem-fios** já são muito vulgares em casas e em empresas. Todos os computadores vendidos atualmente são equipados com um adaptador wireless de níveis de abrangência b, b/g ou b/g/n. Este serviço liga-se a uma ligação de área local, podendo servir de meio de partilha de ficheiros numa **intranet**.

**mbps** (megabits per second)

**wireless**

**intranet**

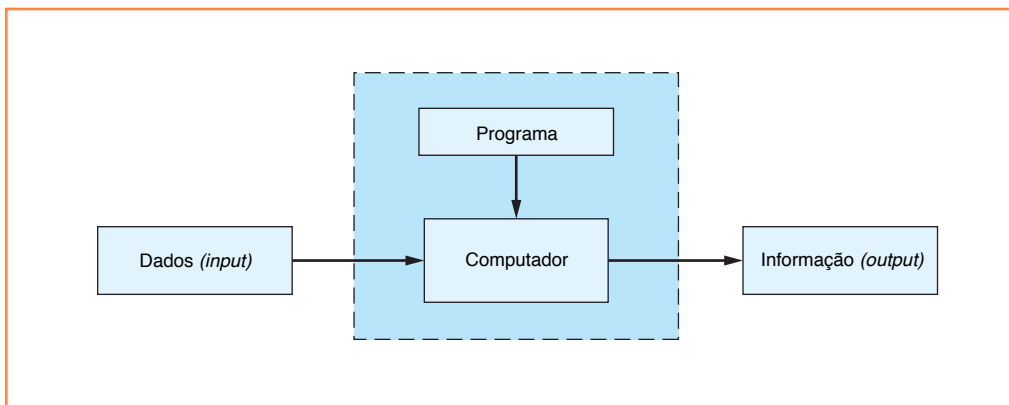
## 2. Introdução à Programação

### Conceito de programa

Muito provavelmente, uma pequena ideia do que é um **programa**, já estará assimilada pela maior parte das pessoas. Todos os utilizadores de dispositivos eletrónicos usam programas. Por exemplo, editores e processadores de texto são dois programas, dos mais usados. Basicamente, um programa é um conjunto de instruções que são dadas ao computador para as seguir. Quando se dá um programa junto de alguns dados, passíveis de serem sintetizados em informação, ao computador, este **corre** (*runs*) ou **executa** o programa.

**programa**

**correr, executar**



**figura 5**

**execução de um programa  
(dupla perspetiva)**

A figura acima mostra duas maneiras de interpretar a execução de um programa. Para conseguir ver a primeira forma, ignoremos a linha a traço interrompido e o sombreado azul que formam um retângulo. O que sobra é o que realmente acontece quando se corre um programa. Nesta perspetiva o computador tem dois tipos de *inputs*: o programa - contém as instruções para execução - e os dados - executandos do programa, a serem processados. Por exemplo, num programa que se baseie simplesmente em correção gramatical, os dados seriam as palavras que se inserissem. Assim, só nos resta o *output*, que será a informação. Lembra-se que a definição de **informação** é diferente de **dados**, pois a primeira traduz-se no processamento da segunda.

**informação, dados**

Uma segunda perspetiva dir-nos-ia que o computador e o programa são vistos com uma só unidade e que só os dados eram *input*, perspetiva esta, refletida na figura atrás, não fazendo qualquer tipo de alteração. Neste caso, o programa é visto como um membro que assiste o computador. As pessoas que escrevem estes membros - chamados de **programadores** - acham esta perspetiva muito mais útil e fácil de usar.

**programadores**

O nosso computador tem mais programas que àqueles que possamos pensar que existem. Grande parte do que nós chamamos de “computador” é realmente um programa - isto é, software. Mal iniciamos um computador entramos num programa,

esse chamado de **sistema operativo, SO** (*operative system, OS*). O sistema operativo funciona como um programa supervisor de todos os outros que correm num computador. Por exemplo, quando queremos abrir um programa, nós, sem sabermos, estamos a informar o sistema operativo de que queremos efetivamente abrir um determinado programa. Logo de seguida, o SO responde e efetua o nosso pedido. Um programa passível de ser aberto pode ser de qualquer tipo, desde um editor de texto a um *browser*, ou até mesmo um programa baseado em Java. Alguns dos mais conhecidos sistemas operativos são o Microsoft Windows, Apple Mac OS, Linux e UNIX.

**sistema operativo (SO)**

## O sistema UNIX

O sistema **UNIX** é um sistema *general-purpose*, multifacetado, interativo e intuitivo, criado pela empresa Bell Laboratories. Inicialmente a **interface de utilizador** (*user interface, UI*) era um **terminal** (*shell*) - linha de comandos - onde se poderia visualizar somente texto. A interação com o sistema fazia-se simplesmente através da introdução de comandos escritos no teclado e da observação da resposta produzida na folha (sistemas muito antigos) ou no ecrã, pelos programas executados. Atualmente existem ambientes gráficos que correm sobre o UNIX e que permitem visualizar informação de texto e gráfica, e interagir por manipulação virtual de objetos gráficos recorrendo a um rato ou a um teclado. É o caso dos sistemas operativos de Linux e Apple Mac OS, e dos Sistemas de Janelas X, como o Quartz, para Mac OS, ou simplesmente X.

**UNIX**

**user interface  
terminal (shell)**

Apesar das novas formas de interação proporcionadas pelos ambientes gráficos, continua a ser possível e em certos casos preferível usar a interface do emulador de terminal, um programa que abre uma janela onde se podem introduzir comandos linha-a-linha e observar as respostas geradas tal como num terminal de texto à moda antiga.

Os comandos em UNIX são escritos como uma sequência de “palavras” separadas por espaços. A primeira palavra corresponde a um **comando** enquanto que as seguintes são **argumentos**. Por exemplo:

**comando**

```
ls -l
```

este comando vai mostrar (*print*) - **imprimir** porque antigamente o *output* era imprimido em folhas de papel, pois não havia ecrã - a lista do nome dos ficheiros armazenados na pasta ou **diretório** (*directory*) atual. O argumento *-l* diz ao *ls* para imprimir a data da última abertura, o tamanho e outras informações detalhadas sobre cada ficheiro.

**imprimir (print)**

**diretório (directory)**

```
drwx----- 13 a12345 users 4096 2007-01-26 14:03 .
drwxr-xr-r 3 root root 4096 2007-01-25 10:52 ..
drwx----- 1 a12345 users 4096 2007-01-26 08:00 Documents
lrwxrwxrwx 1 a12345 users 0 2007-01-25 15:34 Examples ->...
```

**comando ls -l**

Os principais atributos mostrados nestas listagens longas são, mais pormenorizadamente:

- **Tipo de ficheiro** - identificado pelo primeiro carater à esquerda, sendo *d* para diretório, - para ficheiro normal, *l* para *soft link*, etc...;
- **Permissões** - representadas por três conjuntos de três carateres. Indicam as permissões de leitura *r*, escrita *w* e execução/pesquisa *x* relativamente ao dono do ficheiro, aos outros elementos do mesmo grupo e aos restantes utilizadores;

- **Propriedade** - indica a que utilizador e a que grupo pertence o ficheiro;
- **Tamanho** - tamanho do ficheiro, em bytes;
- **Data e hora** - data e hora da última modificação;
- **Nome** - nome do ficheiro.

Repare-se que a resposta, após a inserção do comando no **prompt**, de forma concisa, a resposta foi impressa no ecrã. Este comportamento é normal em muitos comandos UNIX e é típico de um certo estilo defendido pelos criadores deste sistema.

Se executarmos o seguinte comando<sup>1</sup>:

```
cal dez 1995
```

comando cal

obtemos a seguinte resposta:

```

      December 1995
Su Mo Tu We Th Fr Sa
 1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31
```

Na linha de comandos é possível recapitular um comando dado anteriormente usando as teclas *drag* (para cima e para baixo). É possível depois editá-lo para produzir um novo comando com argumentos diferentes, por exemplo. Outra funcionalidade muito útil é a possibilidade de o sistema completar automaticamente comandos ou argumentos parcialmente escritos usando a tecla *tab*.

Para executar um comando, o terminal, normalmente, cria um novo processo e aguarda até que este esteja concluído, para iniciar outro. Para executar dois ou mais comandos simultaneamente basta escrever da seguinte forma:

```
command &
```

assim o terminal vai iniciar o comando indicado e entra em *prompt*, deixando, indicado, um número de processo, como o exemplo abaixo:

```
$ command &
[1] 4810
$ _
```

argumento &

Tal como noutros sistemas operativos, e como já foi dito atrás, no UNIX a informação é armazenada numa estrutura hierárquica formada por diretórios, subdiretórios e ficheiros. O diretório-raiz desta árvore é representado simplesmente por uma barra “/”. Cada utilizador possui um diretório próprio nesta árvore, a partir do qual pode criar e gerir toda a sua sub-árvore de diretórios e ficheiros: é o chamado **home directory** (diretório do utilizador). Após a operação de *login* o sistema coloca-se nesse diretório. Quando abre o terminal deve ser esse, também, o seu diretório atual (*current directory*). Para saber ao certo qual o seu diretório atual digite o seguinte comando, e obterá a resposta que pretende:

home directory

```
$ pwd
/Users/"username"
```

comando pwd

<sup>1</sup> Usando um sistema operativo da gama Apple Mac OS X o argumento do mês é um valor numérico entre (1,...,12), dado que o sistema tem por base o UNIX, embora o *kernel* seja diferente (baseado em BSD).

# 13 PROGRAMAÇÃO I

Neste momento, então, encontra-se sobre um subdiretório de *Users*, subdiretório direto da raiz */*. Para listar, sem detalhes, o conteúdo do diretório, escreva o comando *ls* e obterá uma lista dos ficheiros (e subdiretórios) contidos no diretório atual (*current directory*), por exemplo:

```
$ ls
Documents  Examples
```

comando *ls*

Neste caso observa-se um subdiretório e um **soft link** - tipo de ficheiro que serve de atalho para outro ficheiro ou diretório. Dependendo da configuração do sistema, os nomes nesta listagem poderão aparecer a cores diferentes e/ou com uns caracteres especiais (*/*, *@*, *\**) no final, que servem para indicar o tipo de ficheiro, mas de facto não fazem parte do seu nome.

soft link

Ficheiros cujos nomes começam por *."* não são listados por defeito, são **ficheiros ocultos**, usados geralmente para guardar informações de configuração de diversos programas. Para listar todos os ficheiros de um diretório, incluindo os ocultos, deve executar a variante *ls -a* ou *ls -la* caso pretenda uma lista detalhada com os ficheiros ocultos.

ficheiros ocultos

```
$ ls -a
.Android  Documents  Examples
```

comando *ls -a*

Normalmente existe um **alias** *ll* equivalente ao comando *ls -l*.

alias

Além do *ls* e variantes, existem outros comandos importantes para a observação e manipulação de diretórios, entre os quais:

- ▶ o *cd*, de *change directory* (mudança de diretório):

comando *cd*

```
$ pwd
/Users/"username"
$ cd /Users/"username"/Documents
$ pwd
/Users/"username"/Documents
$ cd
$ pwd
/Users/"username"
```

- ▶ *mkdir dir*, que cria um novo diretório chamado *dir*:

comando *mkdir*

```
$ ls
Documents  Examples
$ mkdir Pictures
$ ls
Documents  Examples  Pictures
```

- ▶ *rmdir dir*, que remove o diretório chamado *dir*:

comando *rmdir*

```
$ ls
Documents  Examples  Pictures
$ rmdir Pictures
$ ls
Documents  Examples
```

O argumento *dir* pode ser dado de forma relativa ou absoluta. Na forma absoluta, identifica o caminho (*path*) para o diretório pretendido a partir da raiz de todo o sistema de ficheiros; tem a forma */subdir1/.../subdirN*. Na forma relativa, *dir*, indica o caminho para o diretório pretendido a partir do diretório atual; tem a forma *subdir1/.../subdirN*.

Há dois nomes especiais para diretórios: “.” e “..” que representam, respetivamente, o diretório atual e o diretório pai - o diretório ao qual o atual pertence.

Quanto à manipulação de ficheiros, o Linux (UNIX) oferece diversos comandos. Entre eles:

- ▶ *cat file* que imprime o conteúdo do ficheiro *file*;
- ▶ *rm file* que remove (apaga o ficheiro *file*):

comando rm

```
$ cd /Users/"username"/Documents/Text Files
$ pwd
/Users/"username"/Documents/Text Files
$ ls
Sample1.txt Sample2.txt Sample3.txt Sites
$ rm Sample2.txt
$ ls
Sample1.txt Sample3.txt
```

- ▶ *mv file1 file2* que muda o nome de *file1* para *file2*:

comando mv

```
$ mv Sample1.txt About.txt
$ ls
About.txt Sample3.txt Sites
```

- ▶ *cp file1 file2* que cria uma cópia de *file1* chamada *file2*:

comando cp

```
$ cp Sample3.txt Zebra.txt
$ ls
About.txt Sample3.txt Sites Zebra.txt
```

- ▶ *cp file1 dir1* que cria uma cópia do ficheiro *file1* dentro do diretório *dir1*:

variante cp file1 dir

```
$ cp About.txt Sites
$ cd Sites
$ pwd
/Users/"username"/Documents/Text Files/Sites
$ ls
About.txt
$ cd ..
$ pwd
/Users/"username"/Documents/Text Files
$ ls
About.txt Sample3.txt Sites Zebra.txt
```

- ▶ *head file1* que mostra as primeiras linhas do ficheiro de texto *file1*:

comando head

```
$ head Zebra.txt
The zebra is an animal.
They exist only in America, Africa and Asia.
Their scientific name is Equus quagga.
```

- ▶ *tail file1* que mostra as últimas linhas do ficheiro de texto *file1*:

comando tail

```
$ tail Zebra.txt
Bibliography:
- Churcher, C.S. 1993. Mammalian Species No. 453.
- McClintock, Dorcas. "A Natural History Of Zebras" 1976.
```

- ▶ *more file1* que imprime no ecrã, página a página, o conteúdo de *file1*;
- ▶ *wc file1* que conta o número de linhas, palavras e caracteres do ficheiro *file1*:

comando wc

```
$ wc Zebra.txt
543      3080      85632 Zebra.txt
```

- ▶ *sort file1* que ordena as linhas do ficheiro *file1*;
- ▶ *find dir1 -name file1* que procura um ficheiro com o nome *file1* a partir do diretório *dir1*.

## Linguagens de programação, compiladores e intérpretes

A maior parte das **linguagens de programação** que hoje em dia se usam estão desenhadas para uma melhor compreensão pela parte que a usa para programar. Essas linguagens chamam-se **linguagens de alto nível** (*high-level language*). O Java é uma dessas linguagens, tal como Visual Basic, C++, C#, COBOL, Python ou Ruby. Infelizmente a máquina (computador) não consegue reconhecer tais linguagens. Assim, antes de um programa ser executado, a linguagem específica tem de ser traduzida para que o computador as entenda.

A única linguagem que o computador consegue compreender chama-se **linguagem da máquina** (*machine language*). Mas essa linguagem, em vez de ser traduzida para Java ou outra de alto nível, pode ser traduzida para uma **linguagem de baixo nível** (*low-level language*), mas, na mesma, compreensível pelo homem. A essa linguagem traduzida dá-se o nome de **linguagem de montagem** (*assembly language*).

A tradução de uma linguagem de alto nível numa de baixo nível é feita, totalmente, através de um outro programa. Para algumas linguagens de alto nível esta tradução é feita, passo-a-passo, por um programa chamado **compilador** (*compiler*). Portanto, antes de executar uma linguagem de alto-nível, devemos iniciar um compilador, para compilar o programa. Só depois é que se pode correr o programa, pois, agora, ele já se encontraria em linguagem de máquina.

A terminologia aqui, neste contexto, pode soar um pouco confusa, dado que tanto para *input* como para *output* do compilador se obtém programa. Tudo, agora, é programa, mas para evitar o erro, ao programa *input*, logo, ao escrito em Java chamarmos-lhe-emos de **programa-fonte** (*source program*) ou **código-fonte** (*source code*). Ao *output* do compilador chamarmos-lhe-emos de **objeto de programa** ou **objeto de código**<sup>2</sup>.

Outras linguagens de alto-nível, ao invés de um compilador, utilizam um **intérprete**, que traduz, também, linguagens de alto nível para linguagens de baixo nível. Mas ao contrário de um compilador, um intérprete faz executar parte do programa enquanto traduz, em vez de traduzi-lo todo de uma só vez. Usando um intérprete, o computador iniciará o elemento de execução do programa, alternando com o de tradução. Para além disso, cada vez que se executa um programa, este é traduzido, o que faz com que o programa, através de um compilador se torne muito mais rápido.

Uma grande desvantagem destes processos que foram apresentados, tanto o compilador como o intérprete é que ambos são específicos para uma determinada linguagem. Mais, se uma empresa produzir um novo tipo de computadores, os seus engenheiros devem criar, de imediato, um compilador e um intérprete para esse computador. Isto é um grande problema, dado que estes programas são muito extensos e demoram muito tempo a fazer.

linguagens de programação

linguagens de alto nível

linguagem da máquina

linguagem de baixo nível

linguagem de montagem

compilador

programa-fonte, código-fonte

objeto de programa

objeto de código

intérprete

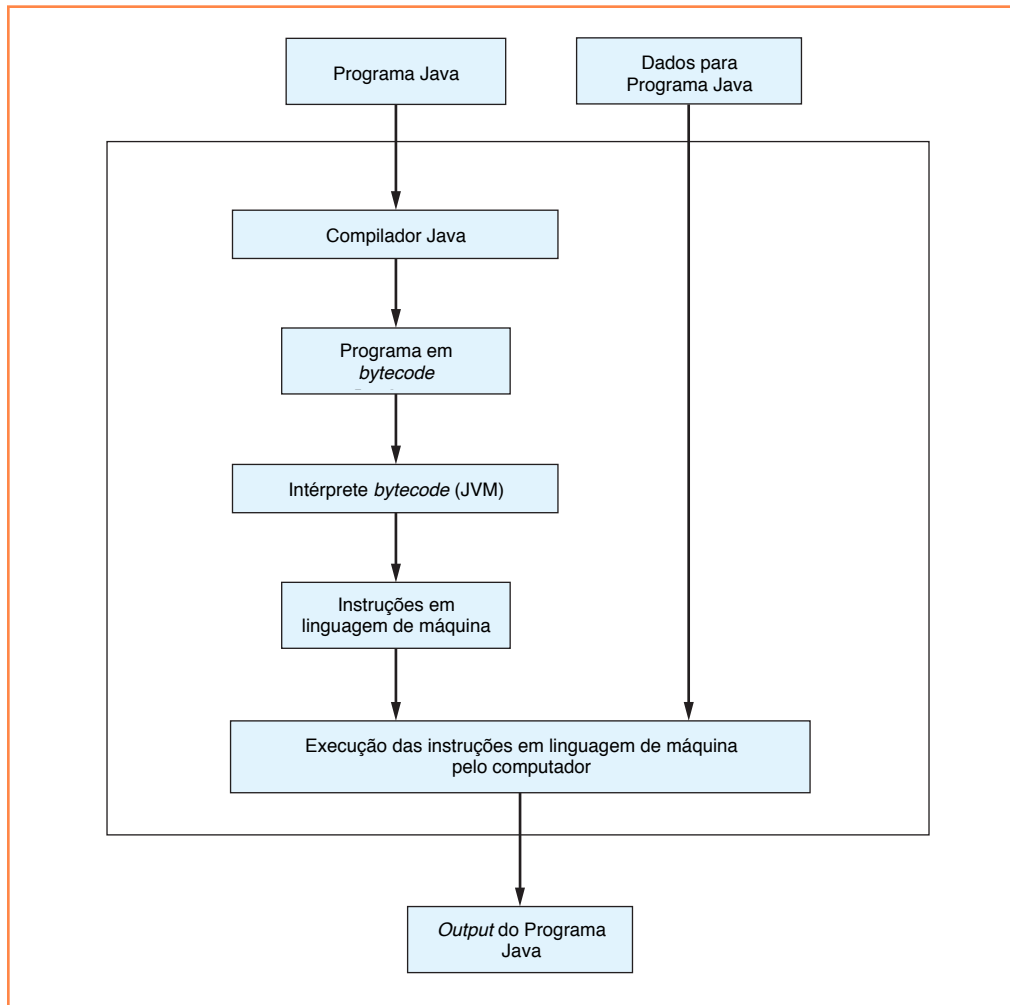
<sup>2</sup> A palavra código aqui significa programa total ou parcial.



## O Java bytecode

O compilador do Java não traduz os programas para a linguagem da máquina de um computador específico. Em vez disso, traduz para uma linguagem com algo de particular, o *bytecode*. O *bytecode* não é nenhuma linguagem específica para um computador, mas antes para uma **máquina virtual**. O programa que faz estas traduções, para uma linguagem típica de máquina (*bytecode*), chama-se **Java Virtual Machine**, ou **JVM** (Máquina Virtual Java).

**bytecode**  
**máquina virtual**  
**JVM**



**figura 6**  
esquematização dos  
processos de programação  
em Java

## 3. Introdução à linguagem Java

A linguagem Java teve a sua criação em 1991, quando James Gosling e a sua equipa na empresa *Sun Microsystems* começaram a escrever uma nova linguagem de programação. Esta linguagem foi pensada para equipar aparelhos utilitários de uma casa, desde uma TV a uma torradeira. Parece uma tarefa sinceramente fácil, mas pelo contrário, é uma autêntica obra de engenharia. A maior parte dos componentes de um destes aparelhos é constituída por um processador de um computador. Algo pequeno, de facto, mas difícil para quem quer criar uma linguagem que funcione com todos os tipos de processadores. Para além disso, há o problema que dificulta quem cria a linguagem, que passa por o fabricante do processador da máquina não ter investido na

criação de compiladores e intérpretes de código, daí o baixo preço do objeto. Para poder contornar este obstáculo, Gosling e a sua equipa construíram um software que traduzisse a linguagem numa linguagem de nível “intermédio”, tradutor esse descrito anteriormente. Quanto à linguagem intermédia, essa é o Java bytecode.

Já em 1994, Gosling teve a ideia de usar a linguagem que criara - já chamada Java<sup>3</sup> - serviria perfeitamente para ser usada num meio de internet ou, anterior, num *browser*.

## Estrutura de um programa

Mais abaixo podemos encontrar um **programa** em Java, na forma de código-fonte e na forma de execução. À pessoa que usará este programa chamar-lhe-emos **utilizador**. Neste documento, a estrutura do código-fonte dos programas estará destacado, por partes a cores, o que não significa que o mesmo padrão de cores seja o que o editor de texto que usemos, tenha como pré-definido. Para escrever um programa, precisa de um editor de texto ou um programa especificamente feito para o efeito. Um programa em Java, tem um código-fonte com o formato de ficheiro **\*.java**.

Um programa simples, que podemos fazer em Java é o seguinte:

```
import java.util.Scanner;
public class MilesToKm {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        double km, miles;
        System.out.print("Distance in miles: ");
        miles = sc.nextDouble();
        km = 1.609 * miles;
        System.out.println("In kilometers, the distance is " + km);
    }
}
```

**programa**

**utilizador**

**\*.java**

**programa simples em Java**

Este exemplo permite-nos compreender algumas bases da **sintaxe**, como também nos permite avaliar a forma como se pensa na resolução do problema, por outras palavras, o **algoritmo**. O algoritmo é uma descrição detalhada e rigorosa da solução de um problema, pois é este mesmo que redigido numa linguagem de programação específica dá origem a um programa. Em java, o conjunto de operações descritas no algoritmo é realizado ordenadamente, do primeiro até ao último. A esta organização lógica dá-se o nome de **execução sequencial**. Mais à frente demonstraremos isso.

**sintaxe**

**algoritmo**

**execução sequencial**

Neste caso em concreto, temos um programa que nos permite converter milhas para quilómetros. Mas antes de um programador ter, efetivamente, um programa, este tem de criar o seu algoritmo. Aqui, analisaremos uma forma prática de organizar e encadear a lógica do algoritmo. Assim, podemos primeiro pensar num **objetivo**. Este objetivo, neste exemplo, seria “converter milhas para quilómetros”. De seguida, devemos **descrever** o que queremos, e como é que queremos, que o programa execute a resolução ao nosso problema - “dada uma distância, expressa em milhas, que é lida do teclado, convertê-la para quilómetros e escrevê-la no ecrã do computador (terminal)”. Num terceiro passo, devemos passar para a **especificação das variáveis**. A especificação das variáveis, basicamente, traduz-se na descrição do que vamos ter que inserir como dados, e no que vamos receber de informação - nós, como utilizadores. Em concreto,

**objetivo**

**descrever**

**especificação das variáveis**

3 O nome Java foi idealizado enquanto os criadores, num dia, tomavam café. Daí o símbolo mais comum do Java ser uma chávena de café. Os criadores avançam também de que, inicialmente, chamava-se Oak, tendo mudado porque, mais tarde, souberam da já existência de uma linguagem de programação com o nome Oak. Esta informação foi retirada de KOFFMAN, Elliot B., *Problem Solving with JAVA*, Addison-Wesley.

com o programa com que estamos a trabalhar, a variável de entrada será *miles* para a distância expressa em milhas, com valor numérico positivo ou nulo, e a nossa variável de saída será, claramente, *km*, a distância expressa em quilómetros, com valor numérico representado com três casas decimais. Um último passo é mesmo a resolução do cálculo, de modo a obter **solução**. Aqui, o cálculo seria “ $km = 1.609 * miles$ ”.

Interpretemos, linha a linha, o programa que foi escrito atrás. Começemos então pela primeira linha.

```
import java.util.Scanner;
```

linha 1 do programa

Esta linha simplesmente diz ao compilador que o programa em questão usa a **classe** *Scanner*. Passemos em frente, no que toca à definição de classe, por agora, mas fiquemos apenas com a ideia de que se trata de um pequeno software que nós podemos utilizar num programa. Esta classe está definida no pacote (*package*) *java.util*, abreviatura de *Java utility* (utilitários do Java). Um **pacote** é uma biblioteca de classes, previamente criadas para uso comum.

classe

pacote

As linhas seguintes definem a classe *MilesToKm*, espaço que se estende desde a primeira chaveta ({) até à última (}). Esta classe tem o nome que nós escolhemos, sendo que esta tem de ser, expressa e unicamente, o nome do nosso programa.

```
public class MilesToKm {
    ...
}
```

linha 2 do programa

Tipicamente, dentro destas últimas chavetas, encontram-se uma ou mais partes às quais se dá o nome de **métodos**, sendo que qualquer aplicação em Java tem o método *main*. A definição destes métodos estendem-se desde a chaveta seguinte até à penúltima de todo o programa.

métodos

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

linha 3 do programa

Mais uma vez, teremos que deixar como mistério, o significado de *public static void*, embora destaquemos que são de grande importância para o programa, sendo vitais para o seu funcionamento.

Qualquer **declaração** ou instrução que se encontre dentro de um método define uma tarefa e cria o **corpo** do mesmo.

declaração

corpo

Na linha seguinte, basicamente temos um componente que diz ao compilador que o programa, naquele passo, vai aceitar e ler a inserção de dados via teclado.

```
Scanner sc = new Scanner(System.in);
```

linha 4 do programa

De seguida encontramos isto:

```
double km, miles;
```

linha 5 do programa

Esta linha significa que estão a ser designadas *km* e *miles* como variáveis deste programa. Uma **variável** é algo que pode gravar dados. Por sua vez, a palavra, ou comando, *double*, significa que o dado tem de um número real. Este tipo de comandos são caraterizadores de dados (*data type*).

variável

```
System.out.print("Distance in miles: ");
```

linha 6 do programa

Estas linhas começadas por *system.out.print* significam que tudo aquilo que está dentro de aspas, dentro dos parênteses, será imprimido no ecrã. Neste caso, o programa simplesmente irá imprimir a seguinte frase “*Distance in Miles:*”.

De seguida, a linha

```
miles = sc.nextDouble();
```

linha 7 do programa

lê o número que é digitado no teclado do computador e grava-o na variável *miles*.

Por sua vez, a linha a seguir, contém as instruções para o cálculo do valor a ser gravado na variável *km* (quilómetros).

```
km = 1.609 * miles;
```

linha 8 do programa

Por outras palavras, esta linha faz com que o valor armazenado na variável *miles*, por exemplo 1, seja multiplicado (\*) por 1.609 o que dá o valor de 1 milha em quilómetros. O resultado deste cálculo será, depois, imprimido no ecrã, através da instrução da linha seguinte:

```
System.out.println("In kilometers, the distance is " + km);
```

linha 9 do programa

Esta linha vai-se mostrar “*In kilometers, the distance is 1.609*”, onde o símbolo + significa que à mensagem “*In kilometers, the distance is* ” (com espaço no fim), se adiciona o valor da variável calculada *km*.

O programa foi, assim, explicado, linha a linha, faltando apenas descrever o significado do **ponto e vírgula** no fim de cada linha. O ponto é vírgula no fim de cada linha significa e indica ao computador de que uma instrução termina nesse ponto.

ponto e vírgula

Evidentemente que a escrita em Java tem de ter regras precisas para cada um dos seus componentes programáveis. Essas regras formam a **gramática** (*grammar*) da linguagem Java. Agora, a linguagem Java tem várias gramáticas, formando-se assim, uma **sintaxe** própria e específica.

gramática

sintaxe

## Elementos básicos da linguagem Java

Na linguagem Java existem vários **elementos** que não podem ser replicados, ao longo do código do programa, como texto simples. Entre estes elementos, encontramos as **palavras reservadas** - símbolos que têm um significado bem definido como *break*, *switch*, *final*, *if*, ... -, **identificadores** - nomes utilizados para designar todos os objetos existentes num programa, devendo, estes, começar por uma letra e só podendo conter letras, números e o símbolo “\_”, como *name*, *age*, *idade*, *i*, *cont\_1*, ... -, **comentários** - palavras e frases que melhoram a legibilidade de um programa, sendo identificados se seguidos de dupla barra “//” ou dentro de um bloco “/\* *texto* \*/” -, **constantes** - “valores específicos” de um tipo determinado como *1*, *10*, *-23*, “*Porto*”, *true*, ... - e **operadores e separadores** - símbolos gráficos ou combinações de símbolos que especificam operações e que são usados na construção de instruções, sendo eles, “( )”, “[ ]”, “{ }”, “< >”, “.”, “:”, “;”, “?”, “|”, “ ”, “ ”, “&”, “|”, “=”, “+”, “-”, “\*”, “/”, “%”, “~”, “^”, “#”, “\”, “\_” e “\$”.

elementos

palavras reservadas

identificadores

comentários

constantes

operadores e separadores

## Tipos de dados primitivos

Para além dos vários elementos básicos usados na linguagem Java, existem vários tipos de dados que podem ser inseridos para caracterizar, ou não, uma variável. Numa primeira fase, falaremos de **dados primitivos**, embora mais tarde trabalhem outros tipos de dados.

dados primitivos

Na tabela abaixo podemos estudar os vários tipos de dados primitivos que podemos usar e que certamente, aparecerão em qualquer aplicação que se criar.

**tabela 1**  
**vários tipos de dados**  
**primitivos**

nome do tipo	descrição	exemplo de aplicação	
byte	este tipo de dados pode assumir tanto valores positivos como negativos e requer apenas 8 bits. O seu valor mínimo é -128 e o seu valor máximo é de 127. Os dados são gravados em complemento para dois.	-128 -35 0 127	números inteiros
short	este tipo de dados pode assumir tanto valores positivos como negativos e requer apenas 16 bits. O seu valor mínimo é -32.768 e o seu valor máximo é de 32.767.	-32.768 -128 127 32.767	
int	este tipo de dados pode assumir tanto valores positivos como negativos e requer apenas 32 bits. O seu valor mínimo é -2.147.483.648 e o seu valor máximo é de 2.147.483.647.	-2.147.483.648 -32.768 32.767 2.147.483.647	
long	este tipo de dados pode assumir tanto valores positivos como negativos e requer apenas 64 bits. O seu valor mínimo é -9.223.372.036.854.775.808 e o seu valor máximo é de 9.223.372.036.854.775.807.	-9.223.372.036.854.775.808 -2.147.483.648 2.147.483.647 9.223.372.036.854.775.807	
float	este tipo de dados requer apenas 32 bits. Ótimo quanto a economia de espaço, para grandes conjuntos de números decimais. Não é muito preciso, mas é ótimo para trabalhar com moeda.	-10.5 10.5	números decimais
double	este tipo de dados requer apenas 64 bits. Geralmente, para se trabalhar com valores decimais, usa-se este tipo de dados.		
boolean	este tipo de dados pode assumir apenas o valor de false e true. Apenas ocupa 1 bit de informação.	false true	lógica

# 21 PROGRAMAÇÃO I

nome do tipo	descrição	exemplo de aplicação	
char	este tipo de dados pode assumir qualquer carater Unicode (16 bits).	a A ?	carateres

Note-se que, muitas vezes, na declaração de variáveis, coloca-se a palavra reservada *final* para declarar que a variável que se segue é uma **constante**.

**constante**

```
final double weight; // definição de constante real
final int age; // definição de constante inteira
```

**constante real e inteira**

Como se pode constatar pelo exemplo acima, uma variável pode ser considerada como uma caixa cujo conteúdo inicialmente é vazio, mas à qual é atribuída um valor. Esse valor pode ser atribuído em várias instâncias:

- ▶ Na altura da definição:

```
double num = 10.5;
int age = 18;
```

**atribuição direta**

- ▶ Usando uma instrução de atribuição (=):

```
double weight;
...
peso = 50.5;
```

**atribuição instrutiva**

- ▶ Lendo um valor do teclado:

```
double miles;
...
miles = sc.nextDouble("Real value: ");
```

**atribuição de leitura**

## Conversões

Sempre que uma expressão tenha operandos aritméticos de tipos diferentes, os operandos com menor capacidade de armazenamento, são automaticamente convertidos para o tipo com maior capacidade (*byte>short>int>long>float>double*). Por outras palavras, estamos a designar a **conversão**, que se traduz numa mudança de tipo de dados que é **unívoca**, ou seja, a conversão inversa não é admitida, pois gera um erro de compilação. Podemos sempre forçar uma conversão através de um operador de conversão (*cast*):

**conversão unívoca**

```
double x;
int y;
y = (int)x; // forçar a conversão para int
```

**conversão (cast)**

## Operadores

Na programação em Java existem vários tipos de operadores, com os quais se podem efetuar todas as operações que se desejam num programa. Os operadores-base podem ser, **aritméticos** (\*, +, -, /, %), **relacionais** (<, <=, >, >=, ==, !=), **lógicos** (!, ||, &&) e de **manipulação de bits** (&, ~, |, ^, >>, <<).

Na tabela a seguir, em suma, encontram-se todos estes operadores com a descrição e com o respetivo exemplo.

**operadores aritméticos, relacionais, lógicos e de manipulação de bits**

operador	nome	exemplo
+	"adição"	1+1=2
-	"subtração"	1-1=0
/	"quociente inteiro"	4/2=2
*	"multiplicação"	2*4=8
%	"resto"	10%3=1
<	"menor que"	se x<10
<=	"menor ou igual que"	se x≤10
>	"maior que"	se x>10
>=	"maior ou igual que"	se x≥10
==	"igual a"	se x==10
!=	"diferente de"	se x!=10
!	inverte valores boolean	!false=true
	operador OR	x    y
&&	operador AND	x && y
&	operador AND (para bits)	x & y
~	complemento	x ~ y
	operador OR (para bits)	x   y
^	operador XOR (para bits)	x ^ y
>>	signed right shift	x >> y
<<	signed left shift	x << y
++	incremento de 1	x++ ; ++x
--	decremento de 1	x- - ; - -x
=	atribuição	x = 1

**tabela 2**  
**vários operadores**

Note-se que as expressões em Java são lidas da esquerda para a direita e dão relevo às prioridades das operações e dos parênteses.

A esta tabela adicionaram-se, também, dois **operadores aritméticos unários** (incremento e decremento). Estes operadores mais o **simétrico** - "-" (-x) - só podem ser utilizados com variáveis e atualizam o seu valor com uma unidade. Estes símbolos colocados antes das variáveis, são **pré-incremento** e **pré-decremento** - neste caso a variável é utilizada depois de ser alterada. Caso sejam colocados depois das variáveis,

**operadores aritméticos unários, simétrico**

**pré-incremento, pré-decremento**

são **pós-incremento** e **pós-decremento**, e neste caso a variável é primeiro usada na expressão onde é utilizada e depois atualizada.

**pós-incremento, pós-decremento**

## Leitura e escrita de dados

Para que os dados pudessem ser imprimidos no ecrã no programa que criámos há pouco, o programa precisava da classe que contivesse todos os caminhos para que o conseguisse ler o produto digitado. A classe que contém os passos para este tipo de leitura é a *java.util.Scanner*.

Mas como o *java.util.Scanner* apenas contém o como fazer, o programa precisa de uma instrução, numa instância específica, para lançar os dados no ecrã. Para isso o utilizador deve usar o comando *print()*, *println()* ou *printf()*, sendo que:

```
System.out.print("Value: " + x); // não muda de linha
System.out.println("Value: " + x); // muda de linha
System.out.printf("Value: " + x); // linha formatada
```

**print(), println() e printf()**

## Escrita formatada

Como vimos atrás, a função *printf* permite escrever informação formatada.

```
System.out.printf("formato de escrita" "lista de variáveis");
```

O formato de escrita é uma sequência de caracteres, que pode conter **especificadores de conversão**. O especificador de conversão é composto pelo % seguido de um carater que indica qual o tipo de dados que queremos escrever (%d, %f, %c, %s, ...). Este carater pode ser precedido de um número com o qual se controla o formato (%3d, %5.1f, %3c, %10s, ...).

**especificadores de conversão**

```
System.out.print("Int.: %6d", 15); // Int.: _ _ _ 1 5
System.out.print("Int.: %6.2f", 14.2); // Int.: _ 1 4 . 2 0
```

**formatação decimal**

## Estruturas de controlo e estruturas de decisão

Uma das particularidades de um computador é a capacidade de repetir tarefas ou executar tarefas consoante determinadas condições. Para implementar programas mais complexos, temos a necessidade de executar instruções de forma condicional. Determinadas instruções só podem ou só devem ser executadas após a avaliação de determinadas condições. As instruções que permitem condicionar a execução de outras designam-se por **estruturas de controlo**. Por agora, vamos apresentar as **estruturas de decisão**. Em Java, temos dois tipos de instruções de decisão (*if* e *switch*).

**estruturas de controlo  
estruturas de decisão**

Relembremos agora o tipo de dados *boolean*. Estes dados podem ser *true* ou *false*. O que a partir daqui iremos estudar, para sempre será usado em programação.

```
boolean cond1, cond2, cond3, cond4, cond5;
cond1 = 3 > 0; // cond1 fica com true
cond2 = 5 != 5; // cond2 fica com false
cond3 = cond1 || cond2; // cond3 fica com true
cond4 = cond1 && cond2; // cond4 fica com false
cond5 = !cond4; // cond5 fica com true
```

**tipo de dados boolean**

## Instrução de decisão if

Nos programas, tal como na vida real, as coisas podem correr de uma ou de mais maneiras. Se nós tivermos dinheiro na nossa conta bancária, o banco pagar-nos-á um certo montante, por interesse. Mas se não tivermos dinheiro na conta bancária, ou



se o nosso saldo for negativo, o banco aumentará multas e fará com que esse saldo ainda se torne mais negativo. Transportando isto para linguagem Java, poderíamos apresentar da seguinte forma:

```
if (balance >= 0) {
    balance = balance + (INTEREST_RATE * balance) / 12; }
else {
    balance = balance - OVERDRAWN_PENALTY; }
```

instrução if

Nesta **condição**, se a variável *balance* tomar o valor maior ou igual a zero, o programa tomará o rumo de *if*, ignorando o braço *else*. Caso contrário (*else*), o braço *if* é ignorado, e o cálculo *balance - overdrawn\_penalty* será tomado como o correto.

condição

Numa forma geral, a função *if-else* é tomada da seguinte forma:

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

sintaxe da instrução if

Um outro tipo de condição é dado através de **instruções de decisão múltipla**, como a *switch*.

instruções de decisão múltipla

```
switch (expressão)
{
    case valor1:
        bloco1;
        break;
    case valor2:
        bloco2;
        break;
    default:
        bloco3;
}
```

sintaxe da instrução switch

Numa condição deste género, a expressão deve ser do tipo enumerado (número inteiro ou carater no caso dos tipos primitivos o Java - *byte*, *short*, *int* ou *char*. As constantes que constituem a lista de alternativas são do mesmo tipo da expressão. Primeiro é calculada a expressão e depois o seu valor é pesquisado na lista de alternativas existentes em cada *case*, pela ordem com que são especificados. Se a pesquisa for bem sucedida, o bloco de código correspondente é executado. Caso não exista na lista e se o *default* existir, o bloco de código correspondente é executado. A execução do *switch* só termina com o aparecimento da instrução *break*.

## Instrução de repetição while

Para além da execução condicional de instruções, por vezes existe a necessidade de executar instruções repetidamente. A um conjunto de instruções que são executadas repetidamente dá-se o nome de **ciclo**. Um ciclo, geralmente, é constituído por uma **estrutura de repetição** (ou de controlo), que comanda quantas vezes as instruções são repetidas. Estas estruturas de repetição podem ser do tipo condicional (*while* ou *do...while*) ou do tipo contador (*for*). Normalmente utilizamos as estruturas do tipo condicional quando o número de iterações é desconhecido, e as estruturas do tipo contador quando sabemos à partida o número de iterações.

ciclo

estrutura de repetição

Como já dissémos no parágrafo anterior, uma forma de construir uma repetição em Java é utilizando uma **instrução while**, também conhecida como *while*

instrução while

*loop*. Esta instrução repete a sua ação infinitamente enquanto que a sua **expressão booleana de controlo** seja verdadeira. Por essa mesma razão, é que esta instrução tem o nome de *while* (enquanto), pois lê-se da seguinte forma: enquanto a expressão for verdadeira, o *loop* é repetido. Quando a expressão booleana é falsa, a repetição acaba. Tomemos como exemplo o seguinte programa:

```
import java.util.Scanner;
public class CountingNumbers
{
    public static void main(String[] args)
    {
        int count, number;

        System.out.print("Enter a number:");
        Scanner kb = new Scanner(System.in);
        number = kb.nextInt();

        count = 1;
        while (count <= number)
        {
            System.out.print(count + ", ");
            count++;
        }

        System.out.println();
        System.out.println("Buckle my shoe.");
    }
}
```

**expressão booleana de controlo**

**aplicação da instrução while**

Este programa tem a instrução *while*, seguida de uma expressão booleana dentro de parênteses. Esta é a expressão que controla a instrução *while*. Enquanto que esta expressão seja verdadeira, a instrução repetir-se-á. Assim, sendo verdadeira, o programa ativará o **corpo** da instrução, que se encontra dentro de chavetas {}. Geralmente, é no corpo que se encontra uma instrução que provoque, depois de 1 ou de mais repetições, a alteração do estado da expressão booleana, passando-a para falsa, terminando assim o programa.

**corpo (body)**

Faremos então uma rápida execução do programa, emitindo, quando nos pedirem, o valor de 2, que será atribuído, automaticamente, à variável *number*.

```
$ java CountingNumbers
Enter a Number:
2
1, 2,
Buckle my shoe.
```

**execução do programa**

Como já sabemos, quando inserimos o número 2, a variável *number* retém o valor 2. Sendo que a nossa expressão booleana diz que apenas se ignora a instrução *while* caso a variável *count* não seja menor ou igual ao valor da variável *number*, como inserimos o número 2, o qual é menor ou igual ao valor da variável *count* (atualmente 1), a nossa expressão booleana é verdadeira, pelo que o programa executará a instrução uma vez, experimentando, de novo, a condição booleana. Ao executar a instrução *while*, esta irá imprimir no ecrã o valor da variável *count* e uma vírgula com um espaço, e irá incrementar o valor da variável *count* em uma unidade, sendo que, neste exemplo, após a primeira execução da instrução *while*, a variável *count* irá tomar o valor de  $1+1=2$ .

Agora, sendo que 2 é o valor da variável *count* e da variável *number*, simultaneamente, o ciclo repetir-se-á uma e uma só vez mais, dado que num ciclo seguinte a variável *count* tomaria o valor de 3 (maior do que 2). Sendo que o valor seria maior do que o valor da variável *number*, a condição booleana traduziria-se em falsa, dado que a condição  $3 \leq 2$  é falsa. Assim, o programa terminará a instrução *while*

e seguirá com as instruções que se seguem às chavetas, imprimindo, na linha seguinte, a frase (*string*) “*Buckle my shoe.*”.

*string*

Se, numa outra execução do programa inserirmos o valor 3, o programa reagirá da seguinte forma:

```
$ java CountingNumbers
Enter a Number:
3
1, 2, 3,
Buckle my shoe.
```

execução do programa

Se, por outro lado, inserirmos um valor inferior ao valor da variável *count*, como o valor 0, o programa, sendo que a condição booleana ganha o valor de falso logo no início, não vai correr a instrução *while*, imprimindo, apenas, numa linha abaixo, a *string* “*Buckle my shoe.*”.

```
$ java CountingNumbers
Enter a Number:
0

Buckle my shoe.
```

execução do programa

**Nota!** Com este último exemplo, podemos constatar que a **instrução *while* pode fazer zero repetições**. Em síntese, podemos referir que a sintaxe da instrução *while* é,

nota

```
while (Expressão booleana)
    Corpo da instrução
```

sintaxe da instrução *while*

onde o corpo da instrução pode ser constituído por uma só instrução, ou mais provavelmente, por um conjunto de instruções englobadas por chavetas {}.

Um outro tipo de instrução de repetição condicional é a **instrução *do...while***. Esta instrução é muito semelhante à que estudámos anteriormente. A principal diferença entre ambas reside no facto de que, enquanto que a *while* pode executar zero ciclos, a instrução *do...while* executa, sempre, pelo menos, um ciclo. O que acontece, de facto, é que na instrução *while* a condição booleana é verificada no início, sendo que, caso dê falso, o corpo do *while* é ignorado. No caso da instrução de repetição *do...while* a condição booleana é sempre avaliada no fim de cada ciclo, sendo que, para a primeira avaliação, o programa já executou um ciclo.

instrução *do...while*

O funcionamento, respetivamente à avaliação e prosseguimento da instrução, é igual à *while*. O que apenas muda é, de facto, o momento em que é avaliada a condição booleana. Portanto, quando a condição for falsa, a instrução acaba imediatamente. A sintaxe, assim, para a instrução *do...while* é a seguinte:

```
do
    Corpo da instrução
while (Expressão Booleana);
```

sintaxe da instrução  
*do...while*

**Nota!** A seguir a expressão booleana encontra-se um ponto e vírgula, essencial para o bom funcionamento do programa!

nota

Na página seguinte podemos encontrar um exemplo de programa que usa a instrução *do...while*, semelhante ao exemplo que usámos para a introdução à instrução *while*, sendo que, como valores de entrada (*inputs*) usaremos exatamente os mesmos que usámos no exemplo anterior, de forma a conseguir obter conclusões no final de uma análise dos seus funcionamentos.

```
import java.util.Scanner;
public class CountingNumbers
{
    public static void main(String[] args)
    {
        int count, number;

        System.out.print("Enter a number:");
        Scanner kb = new Scanner(System.in);
        number = kb.nextInt();

        count = 1;
        do
        {
            System.out.print(count + ", ");
            count++;
        } while (count <= number);

        System.out.println();
        System.out.println("Buckle my shoe.");
    }
}
```

aplicação da instrução  
do...while

Se inserirmos o valor de 2, quando nos pedem para inserir um número, o resultado obtido no terminal é o seguinte:

```
$ java CountingNumbers
Enter a Number:
2
1, 2,
Buckle my shoe.
```

execução do programa

Como podemos constatar, o resultado deste programa, para *number = 1* é igual em ambos os casos de *while* e *do...while*. Experimentemos, agora para 3.

```
$ java CountingNumbers
Enter a Number:
3
1, 2, 3,
Buckle my shoe.
```

execução do programa

Mais uma vez, obtivemos o mesmo resultado com a instrução *do...while* que com a instrução *while*. Tentemos agora, com o número 0.

```
$ java CountingNumbers
Enter a Number:
0
1,
Buckle my shoe.
```

execução do programa

Como prova da teoria que afirma que é executada a avaliação após execução das instruções, este último exemplo funciona, pois, com a instrução *while*, nós não obtivemos contagem, porque 0 não é maior ou igual a 1, o que fez com que o seu corpo fosse ignorado. Pelo contrário, com a instrução *do...while*, o corpo da instrução foi executado uma vez e só depois foi avaliada a condição, obtendo o resultado de falso, que provocou o *terminus* do programa.

## Análise de um problema e criação de um algoritmo

Antes de um programa estar operacional, um vasto conjunto de tarefas foram realizadas por programadores para preparar a criação deste. Assim, estudemos um caso de preparação de um programa, que possa satisfazer um cliente, resolvendo o seguinte problema:

*A nossa cidade foi palco de uma infestação de baratas. Este não é um assunto muito bom de se falar, mas felizmente uma companhia local chamada Debugging Experts Inc. tem um tratamento que consegue eliminar as baratas, numa casa. Tal como o seu slogan diz “É um trabalho sujo, mas alguém o tem de fazer”. O único problema é que os concidadãos são muito preocupados com a saúde da população das baratas em toda a cidade, preferindo que estas saiam sozinhas para fora dela. Dado este problema, a empresa Debugging Experts instalou um computador no centro comercial local para informar da gravidade do problema em casas em particular. Esse programa instalado no computador calcula o número de semanas que uma população de baratas demora a propagar-se numa casa, por completo.*

*Uma população de baratas cresce significativamente devagar - para baratas - mas mesmo assim, a situação é má. A população atingirá cerca do seu dobro a cada semana que passa. Se a população cresce o dobro cada semana, significa que a taxa de crescimento é de 100% por semana, mas felizmente é apenas de 95%. Estas baratas, para além de outras estatísticas, são bastante grandes. Em termos de volume, o seu tamanho médio é de cerca de 0.056 litros. O programa do centro comercial faz certas simplificações. Ele assume que uma casa não tem mobília instalada e que as baratas ocuparão todo o espaço da casa, sem qualquer exceção.*

Pensemos então numa possível solução para o nosso problema. Começemos por fazer um pequeno **esboço** do nosso algoritmo. Ele deve-se parecer com algo deste género:

- ▶ 1. Obter o volume da casa;
- ▶ 2. Obter o número inicial de baratas na casa;
- ▶ 3. Calcular o número de semanas até a casa estar cheia de baratas;
- ▶ 4. Apresentar os resultados.

Sabendo a taxa de crescimento da população de baratas e o seu tamanho médio, nós não vamos fazer com que o programa leia esses valores, mas antes vamos defini-los como **constantes finais**.

**constantes finais**

O nosso algoritmo parece bastante simples, mas não nos apressemos a chamar programadores. O nosso passo terceiro do algoritmo é, sem dúvida, o centro de todo o programa, mas, até agora, não nos diz nada sobre como calcular o número de semanas. Pensemos então sobre o assunto. Nós precisamos de calcular um número de semanas, logo usemos um contador que comece no zero e conte semana a semana, adicionando uma unidade. Sendo que nós já sabemos o volume de uma barata, em média, e o número de baratas inicial, nós podemos calcular o volume total, multiplicando, simplesmente, estes dois valores. Isto dar-nos-á o volume de baratas na semana 0. Já acabámos? Não. Nós só acabamos quando a casa estiver toda cheia de baratas. Se tal não acontecer, nós precisamos de adicionar o volume das novas baratas que chegaram à casa na semana seguinte. Então, mas acabámos agora? Não. Mais uma vez, caso a casa não esteja já cheia, temos de voltar a adicionar o valor do volume que acaba de chegar à casa na semana 2. E agora volta a perguntar: já acabou? E eu volto a responder: Não! Chegamos então à conclusão de que precisamos de uma instrução de repetição: um *loop*.

Mas que instrução de repetição é que vamos usar aqui? Em algumas situações, podemos não ser capazes de decidir, tão cedo, mas também é uma questão, em caso de escolha desadequada, de substituir mais tarde. Mas para este caso, nós já constatámos que a infestação inicial pode, de facto, encher a casa. Se for esse o caso, o nosso

programa já não vai ter de calcular nada. Assim, teremos que usar uma instrução *while* em vez de uma *do...while*.

Começemos agora, uma lista de variáveis e de constantes.

- *GROWTH\_RATE* - constante que representa a taxa de crescimento de uma população de baratas (igual a 0.95);
- *ONE\_BUG\_VOLUME* - constante que representa o volume médio de uma barata (igual a 0.056);
- *houseVolume* - representa o volume de uma casa;
- *startPopulation* - representa o número inicial de baratas;
- *countWeeks* - representa a contagem de semanas;
- *population* - representa o número atual de baratas;
- *totalBugVolume* - representa o número total de baratas;
- *newBugs* - representa o número de novas baratas numa nova semana;
- *newBugVolume* - representa o volume das novas baratas.

Nós podemos pensar nas variáveis como nós melhor pretendermos. Podemos também não pensar nelas todas, simultaneamente. Enquanto refinamos o nosso **pseudocódigo**, nós podemos adicionar outras variáveis à lista, enquanto elas nos vêm à cabeça.

**pseudocódigo**

Antes de passarmos à programação em Java, tomemos tempo para desenhar o programa: escrever pseudocódigo, desenhar figuras e fazer uma lista de variáveis propostas e os seus significados.

Nós agora podemos substituir o nosso terceiro passo pelos seguintes passos:

- 3a. *countWeeks* = 0
- 3b. Repetir até a casa estar cheia de baratas:

```
{
    newBugs = population * GROWTH_RATE
    newBugVolume = newBugs * ONE_BUG_VOLUME
    population = population + newBugs
    totalBugVolume = totalBugVolume + newBugVolume
    countWeeks = countWeeks + 1
}
```

Olhando para o passo 3b, nós podemos reparar que a linguagem Java não tem uma instrução de *repetir até*, construída. Nós podemos reconstruir a frase da seguinte forma: *repetir enquanto a casa não está cheia de baratas*, ou *repetir enquanto o volume de baratas é menor que o volume da casa*. Assim, escrevemos da seguinte forma:

```
while (totalBugVolume < houseVolume)
```

**expressão booleana do while**

Juntando todas as peças no nosso algoritmo, temos:

- 1. Ler *houseVolume*;
- 2. Ler *startPopulation*;
- 3. *population* = *startPopulation*;
- 4. *totalBugVolume* = *population* \* *ONE\_BUG\_VOLUME*;
- 5. *countWeeks* = 0;

- ▶ 6. *while* (*totalBugVolume* < *houseVolume*)
  - {
  - newBugs* = *population* \* *GROWTH\_RATE*
  - newBugVolume* = *newBugs* \* *ONE\_BUG\_VOLUME*
  - population* = *population* + *newBugs*
  - totalBugVolume* = *totalBugVolume* + *newBugVolume*
  - countWeeks* = *countWeeks* + 1
  - }
- ▶ 7. Apresentar *startPopulation*, *houseVolume*, *countWeeks*, *population* e *totalBugVolume*.

O nosso *loop* simplesmente atualiza a população de baratas, o volume de baratas e o contador de semanas. Porque a taxa de crescimento e o volume médio de uma barata são números positivos, nós sabemos o valor de *population* e, por conseguinte, o valor de *totalBugVolume*, irá crescer em cada ciclo. Eventualmente, o valor de *totalBugVolume* irá exceder o valor de *houseVolume*, e a expressão booleana de controlo *totalBugVolume* < *houseVolume* apresentará o valor falso, terminando a repetição *while*.

A variável *countWeeks* inicia com o valor de zero e é-lhe incrementado uma unidade, em cada ciclo. Assim, quando o *loop* termina, o valor de *countWeeks* é o número total de semanas que as baratas demoram a encher uma determinada casa.

O produto final, assim, terá a seguinte forma:

```
import java.util.Scanner;
public class BugTrouble
{
    public static final double GROWTH_RATE = 0.95;
    public static final double ONE_BUG_VOLUME = 0.056;

    public static void main(String[] args)
    {
        System.out.println("Enter the total volume of your house");
        System.out.print("in liters: ");
        Scanner kb = new Scanner(System.in);
        double houseVolume = kb.nextDouble();

        System.out.println("Enter the estimated number of");
        System.out.print("roaches in your house: ");
        int startPopulation = kb.nextInt();
        int countWeeks = 0;
        double population = startPopulation;
        double totalBugVolume = population * ONE_BUG_VOLUME;
        double newBugs, newBugVolume;

        while (totalBugVolume < houseVolume)
        {
            newBugs = population * GROWTH_RATE;
            newBugVolume = newBugs * ONE_BUG_VOLUME;
            population = population + newBugs;
            totalBugVolume = totalBugVolume + newBugVolume;
            countWeeks++;
        }

        System.out.println("Starting with a roach population of " +
            startPopulation);
        System.out.println("and a house with a volume of " +
            houseVolume + " liters,");
        System.out.println("after " + countWeeks + " weeks,");
        System.out.println("the house will be filled with " +
            (int)population + " roaches.");
        System.out.println("They will fill a volume of " +
            (int)totalBugVolume + " liters.");
        System.out.println("Better call Debugging Experts Inc.");
    }
}
```

código do programa

Num terminal, o programa teria o seguinte aspeto:

```
$ java BugTrouble
Enter the total volume of your house
in liters: 20000
Enter the estimated number of
roaches in your house: 100
Starting with a roach population of 100
and a house with a volume of 20000.0 liters,
after 18 weeks,
the house will be filled with 16619693 roaches.
They will fill a volume of 33239 liters.
Better call Debugging Experts Inc.
```

execução do programa

## Instrução de repetição *for* (contador)

A **instrução *for*** permite-nos escrever facilmente um *loop* que é controlado por um tipo de contador. Por exemplo, o seguinte pseudocódigo define um *loop* que repete três vezes e é controlado pelo contador *count*.

instrução *for*

Faz o seguinte para cada valor de *count* de 1 a 3:  
Apresenta *count*

pseudocódigo da instrução  
*for*

Este pseudocódigo em particular pode ser representado em Java da seguinte forma:

```
for (count = 1; count <= 3; count++)
    System.out.println(count);
```

aplicação da instrução *for*

Esta instrução *for* causa o seguinte num terminal:

```
1
2
3
```

execução do programa

Depois de uma instrução *for*, qualquer instrução que se siga, executa-se.

Tentemos compreender melhor a iteração causada pela instrução *for*. A primeira das três expressões em parênteses, *count = 1*, diz o que acontece antes do *loop* iniciar pela primeira vez. A terceira expressão, *count++*, é executada depois de cada ciclo. A expressão do meio, *count <= 3*, é uma expressão booleana que determina quando é que o ciclo termina, e resolve-se da mesma forma que numa instrução *while* e *do...while*. Assim, o corpo da instrução *for* é executada enquanto o valor de *count* for menor ou igual a 3.

Concluindo sobre a sintaxe da instrução *for*, esta sumariza-se no seguinte:

```
for (Ação_inicial; Expressão_Booleana; Ação_de_atualização)
    Corpo_da_instrução
```

sintaxe da instrução *for*

Tomemos o seguinte programa como exemplo:

```
import java.util.Scanner;
public class Count123
{
    public static void main(String[] args)
    {
        int countDown;
        for (countDown = 3; countDown >= 0; countDown--) {
            System.out.println(countDown);
            System.out.println("and counting."); }
        System.out.println("Blast off!"); }
}
```

aplicação da instrução *for*



Num terminal, este programa teria o seguinte aspeto:

```
$ java Count123
3
and counting.
2
and counting.
1
and counting.
0
and counting.
Blast off!
```

execução do programa

## Instruções break e continue

Dado que já estudámos instruções de repetição (*loops*), entre as quais, a *while*, *do...while* e *for*, falta-nos estudar formas de terminar a sua execução quando a expressão booleana de controlo dá falsa. Tomemos como exemplo o seguinte programa:

```
import java.util.Scanner;
public class SpendingSpree
{
    public static final int SPENDING_MONEY = 100;
    public static final int MAX_ITEMS = 3;
    public static void main(String[] args)
    {
        Scanner kb = new Scanner(System.in);
        boolean haveMoney = true;
        int leftToSpend = SPENDING_MONEY;
        int totalSpent = 0;
        int itemNumber = 1;
        while (haveMoney && (itemNumber <= MAX_ITEMS))
        {
            System.out.println("You may buy up to " + (MAX_ITEMS -
                itemNumber + 1) + " items");
            System.out.println("costing no more than $" + leftToSpend
                + ".");
            System.out.print("Enter cost of item #" + itemNumber +
                ": $");
            int itemCost = kb.nextInt();
            if (itemCost <= leftToSpend)
            {
                System.out.println("You may buy this item. ");
                totalSpent = totalSpent + itemCost;
                System.out.println("You spent $" + totalSpent + " so
                    far.");
                leftToSpend = SPENDING_MONEY - totalSpent;
                if (leftToSpend > 0)
                    itemNumber++;
            }
            else
            {
                System.out.println("You are out of money.")
            }
        }
        else
        {
            System.out.println("You cannot buy that item.");
        }
        System.out.println("You spent $" + totalSpent + ", and are
            done shopping.");
    }
}
```


programa de exemplo

Neste exemplo a expressão booleana de controlo relaciona-se com a variável booleana *haveMoney*. Portanto, quando *haveMoney* é falso, a expressão de controlo é falsa e o *loop* acaba.

Para além disso, uma repetição pode ser finalizada através de uma **instrução *break***, terminando o *loop* e o que resta de instruções do programa, terminando-o por **instrução *break***

completo. A linguagem Java permite que a instrução *break* seja usada dentro de uma instrução *while*, *do...while* e *for*. Também pode ser usada em *if...else* e em *switch*. Se alterarmos o esquema do programa anterior, adicionando um *break* no fim do primeiro *else*, no caso de *else* acontecer, o *loop* termina.

```
while (haveMoney && (itemNumber <= MAX_ITEMS))
{
    if (itemCost <= leftToSpend)
    {
        if (leftToSpend > 0)
            itemNumber++;
        else
        {
            System.out.println("You are out of money.");
            break;
        }
    }
    else
    {
        . . .
    }
}
System.out.println( . . . );
```



aplicação da instrução *break*

Podemos constatar que, com a mudança que fizemos no programa, o *loop* acaba tanto quando ficamos sem dinheiro ou quando excedemos o máximo número de itens.

Uma **instrução *continue*** num corpo de uma repetição acaba a iteração atual e faz seguir as instruções. Usando esta instrução desta forma tem os mesmos problemas que usando uma instrução *break*. O meu conselho é evitar tanto o uso das instruções *break* como da *continue*.

instrução *continue*

## 4. Introdução à programação modular

Quando resolvemos um algoritmo, aquando da criação de um programa, tentamos sempre ter, pelo menos, uma solução para um determinado problema. Para a especificação de um problema, várias ações devem ser tomadas, no contexto de um programa - ele deve ler, calcular e imprimir dados.

Até agora só estudámos exemplos de baixa complexidade, pois ainda nos estávamos a introduzir na linguagem Java. Mas agora, dado que os nossos conhecimentos devem ser muito superiores, comparado a quando começámos, o grau de complexidade começa a subir, pelo que a designação do algoritmo para a solução de um problema deve ser diferente. É assim que podemos introduzir o conceito de **método** ou **função**. A linguagem Java permite-nos, então, a implementação de sub-programas, de modo a poder haver uma grande simplificação do nosso programa. Dividindo o nosso programa, no programa *main* e em outros sub-programas, cada um contendo partes do nosso algoritmo, permite a fácil avaliação do mesmo e a captação de erros. Assim, uma função deve ser sempre capaz de realizar um determinado número de operações e depois devolver um valor ao programa *main*.

método ou função

As funções criadas pelo programador são denominadas da mesma forma que as funções criadas por terceiros (como as funções de leitura ou as funções da classe *Math*).

Relembrando a estrutura normal de um programa, tendo em conta a convenção usada pelo Java, temos que primeiro existe uma inclusão de classes externas, uma classe pública, a qual contém uma função *main*, declaração de variáveis

(finais e variáveis) normais. Com a mesma indentação que a função *main*, encontram-se, abaixo, as funções definidas pelo programador.

```
inclusão de classes externas;
public class Programa
{
    public static void main(String[] args)
    {
        declaração de constantes e variáveis;
        sequências de instruções;
    }
    funções definidas pelo programador
}
```

localização de funções

Uma função tem uma estrutura, da mesma forma, própria. No cabeçalho da função indicam-se os **qualificadores de acesso** (por enquanto, dado os nossos conhecimentos, será sempre *public static*), o **tipo de resultado de saída**, o **nome da função** e dentro de parênteses curvos, a lista de argumentos do sub-programa.

```
public static tipo nome (argumentos)
{
    declaração de variáveis;
    sequências de instruções;
}
```

quantificadores de acesso

tipo de resultado de saída,  
nome da função

sintaxe de funções

Com mais rigor, uma função é uma unidade auto-contida que recebe dados do exterior através de argumentos, caso necessários, realizando tarefas e devolvendo um resultado, também, se necessário. O resultado da saída de uma função pode ser de qualquer tipo primitivo (*int*, *double*, *char*, ...), de qualquer tipo de referência (como iremos ver mais à frente) ou do tipo *void* (caso uma função não devolver valor). A lista de argumentos (ou parâmetros) corresponde a uma lista de pares de identificadores separados por vírgula, onde para cada argumento se indica o seu tipo de dados e o seu nome. Se uma função devolve um valor, então usa-se a palavra reservada *return*. Uma condição essencial para que o valor possa ser retornado, reflete-se no facto de ter de ser compatível com o tipo de saída da função *main*.

return

Suponhamos então, que temos o seguinte programa em mãos.

```
int sum = 0;
for (int i = 1; i <= 10; i++)
    sum += i;
System.out.println("Sum from 1 to 10 is " + sum);

int sum = 0;
for (int i = 20; i <= 37; i++)
    sum += i;
System.out.println("Sum from 20 to 37 is " + sum);

int sum = 0;
for (int i = 35; i <= 49; i++)
    sum += i;
System.out.println("Sum from 35 to 49 is " + sum);
```

programa de exemplo

Este programa, como podemos analisar, à variável inteira *sum* vai adicionando, com o incremento *i* calculado na instrução de repetição *for*, todos os valores entre 1 e 10, imprimindo-os no ecrã. Este programa não tem a intervenção do utilizador, pelo que não tem grande interesse, a não ser académico. Usando métodos, podemos não só reduzir o tamanho do programa (código-fonte), como podemos simplificá-lo, reutilizando uma pequena porção dele, generalizando a instrução de repetição, invocando-a apenas quando precisarmos dela. Na página seguinte podemos comprovar como é muito mais fácil a utilização de métodos para a criação dos programas.

```

public static int sum(int i1, int i2) {
    int result = 0;
    for (i = i1; i <= i2; i++)
        result += i;
    return result;
}

public static void main(String[] args) {
    System.out.println("Sum from 1 to 10 " + sum(1,10));
    System.out.println("Sum from 1 to 10 " + sum(20,37));
    System.out.println("Sum from 1 to 10 " + sum(35,49));
}

```

exemplo de função

Se analisarmos agora o mesmo programa, mas escrito desta forma, podemos primeiro constatar que estamos na presença de duas partes: um programa *main* e um sub-programa *sum*, este último em destaque no exemplo. Basicamente estamos a “ensinar” o computador a ler as nossas novas instruções e a usá-las sempre que houver **invocação**.

invocação

Façamos uma análise do programa linha-a-linha. Começemos pelo sub-programa *sum*.

```
public static int sum(int i1, int i2) {
```

linha 1 da função

Tal como em qualquer outro programa que tenhamos feito até agora, usamos o quantificador de acesso *public static* seguido do tipo de valores que são devolvidos. Como até agora não fizemos nenhum programa que devolvesse valores, nós temos colocado, como tipo de valor devolvido, *void*, o que significa que não tem nada para devolver. Pelo contrário, agora estamos a criar sub-programas cujo intuito principal é de calcular um valor, devolvendo-o no final. Por essa mesma razão, vamos ter de passar a colocar o tipo de valores que queremos que sejam devolvidos, com a especial atenção de que estes têm de ser exatamente iguais ao tipo de valores especificados aquando da invocação. Assim, neste caso em específico, usámos o tipo *int* (inteiro) para identificar que os valores admitidos para esta operação são unicamente valores inteiros. De seguida, aparece *sum(int i1, int i2)*, que não é nada mais, nada menos, que o nome do nosso sub-programa (método). Pensemos nisto como numa função matemática. Quando trabalhamos com funções matemáticas, dizemos que uma dada função *f* tem como expressão  $x^2+x$ , sendo que para calcular  $f(21)$  basta substituir o nosso *x* na expressão por 21 e efetuar o cálculo. O mesmo acontece para os métodos. Neste caso temos a função *sum(i1, i2)*, o qual espera por um valor inteiro para a incógnita *i1* e outro para a incógnita *i2*. A expressão desta função é traduzida através do que se segue. Começando por definir uma variável inteira de nome *result* e atribuindo-lhe o valor de 0, a expressão consiste na substituição de valores para *i1* e para *i2*.

```
int result = 0;
```

linha 2 da função

Pela substituição, temos que a instrução de repetição *for* vai executar o seu conteúdo para todos os valores de *i1* que são menores ou iguais a *i2*, incrementando, por cada ciclo, uma unidade a uma determinada variável designada por *i*.

```
for (i = i1; i <= i2; i++)
```

linha 3 da função

Por cada ciclo completado, a variável *result* ganha mais um incremento de *i*, adicionando ao “antigo” *i* um “novo” *i*.

```
result += i;
```

linha 4 da função

De seguida o programa devolve o valor calculado e guardado na variável *result*, quando o valor de *i1* é igual ao valor de *i2*.

```
return result;
```

linha 5 da função

O sub-programa já está analisado. Agora basta analisar a composição do nosso programa *main* e compreender onde é que os dados são invocados.

```
System.out.println("Sum from 1 to 10 " + sum(1,10));
```

linha 2 da função main

Quando no programa *main* se pede, como variável *sum(1,10)* significa que estamos a invocar a função *sum* com o valor de *i1* igual a 1 e com o valor de *i2* igual a 10. O mesmo acontece para as seguintes linhas.

Dá-se especial atenção para um pequeno pormenor que muitas vezes passa despercebido e que não retém o compilador com erro: aquando da definição de argumentos no sub-programa escrever *sum(int i1, int i2)* está correto, enquanto que escrever *sum(int i1, i2)* está totalmente errado. Este é um erro muito comum, pelo que merece toda a nossa atenção.

## Métodos void

Como foi referido atrás, um método *void* não devolve qualquer valor. Os exemplos seguintes mostram dois programas semelhantes que diferem entre o tipo de método que usam. No primeiro caso usa-se um método *void* que não devolve valor, mas sim uma frase (*statement*), enquanto que no segundo usaremos um tipo *char* (carater).

```
public class VoidMethodDemo {
    public static void main(String[] args) {
        System.out.print("The grade is ");
        printGrade(78.5);
        System.out.print("The grade is ");
        printGrade(49.5);
    }
    public static void printGrade(double score) {
        if (score >= 90) {
            System.out.print("A");
        }
        else if (score >= 80) {
            System.out.print("B");
        }
        else if (score >= 70) {
            System.out.print("C");
        }
        else if (score >= 60) {
            System.out.print("D");
        }
        else if (score >= 50) {
            System.out.print("E");
        }
        else {
            System.out.print("F");
        }
    }
}
```

aplicação de método void

Se executássemos este programa iríamos obter o seguinte resultado.

```
$ java VoidMethodDemo
The grade is C
The grade is F
```

execução do programa

Como podemos constatar, o método *printGrade* não devolveu nada, apenas deu continuidade às instruções que se seguiam, para que fossem executadas.

Vejam os então o mesmo programa com um tipo de devolução de dados *char*.

```
public class CharMethodDemo {
    public static void main(String[] args) {
        System.out.print("The grade is " + getGrade(78.5));
        System.out.print("\nThe grade is " + getGrade(49.5));
    }
    public static char getGrade(double score) {
        if (score >= 90) {
            return "A";
        }
        else if (score >= 80) {
            return "B";
        }
        else if (score >= 70) {
            return "C";
        }
        else if (score >= 60) {
            return "D";
        }
        else if (score >= 50) {
            return "E";
        }
        else {
            return "F";
        }
    }
}
```

aplicação de um método char

Neste caso o método usado tem como tipo *char*, pelo que, sendo esta uma variável que guarda um valor numérico respetivo a um determinado carater, que é devolvido, ao contrário do que acontecia com métodos *void*.

## Estruturas de Dados

Até ao momento estudámos, como simplificação de programas com algoritmos mais complexos, a criação de métodos. Os métodos são, basicamente, funções que são criadas e que contêm parâmetros passíveis de serem substituídos. Agora introduziremos novas **estruturas de dados**, por outras palavras, **tipos compostos** de dados.

estruturas de dados,  
tipos compostos

Lembre-mo-nos que a linguagem Java tem base numa hierarquia que contém vários objetos inseridos em várias classes. Antes de entrarmos em maior detalhe, analisemos o que já podemos concluir sobre os conceitos de classes e objetos. Um **objeto** num programa é tanto análogo a objetos no mundo real, como carros, casas, ..., como a objetos abstratos como cores, figuras ou palavras. Por sua vez, uma **classe** é uma definição de um tipo de objeto. Tomemos como exemplo uma descrição de uma dada classe. De uma forma, de novo, análoga, imaginemos uma classe chamada *automóvel*. A classe *automóvel* é um resumo geral do que é um dado automóvel e do que ele é capaz de fazer. Um objeto é um automóvel em específico.

objeto  
classe

Voltando à linguagem Java, quando estamos perante problemas mais complexos, com mais dados de entrada, torna-se mais complicada, a decomposição do problema em várias funções, dado que apenas podemos devolver uma variável de tipo primitivo, por função. Assim, há problemas em que seria excelente, poder otimizar a sua solução através de um novo tipo de dados devidamente ajustado ao efeito. Para além disso, em muitas situações práticas, podemos precisar de armazenar informação relacionada entre si, eventualmente de tipos diferentes, na mesma variável. Todas as linguagens de programação permitem que o programador defina tipos de dados particulares para adequar a representação da informação às condições concretas do problema. Na linguagem Java podemos utilizar classes para a construção de novos

tipos de dados ou **registos**. Um registo é então um tipo de dados que pode conter campos de cada um dos tipos primitivos (*int*, *double*, *char*, *boolean*, ...), ou outros tipos compostos. Lembremo-nos de que existem dois géneros de tipos de dados: **primitivos** e de **referência**. Os tipos de dados referência são os compostos e os *String*, por exemplo, ou os *arrays*, como vamos estudar mais à frente.

**registos**  
**dados primitivos, dados referência**

Os registos são novos tipos de dados de referência que se criam abaixo das funções, por uma questão de pura convenção.

```
inclusão de classes externas;
public class Programa
{
    public static void main(String[] args)
    {
        declaração de constantes e variáveis;
        sequências de instruções;
    }
    funções definidas pelo programador
}
class NomeDoRegisto {
    definição do tipo de dados
}
```

localização de registos

Os registos também têm uma forma única de serem criados. Para além da sua posição no programa, a sua sintaxe é importante. A sintaxe é a seguinte:

```
class NomeDoRegisto {
    tipo_de_dado1 nome_do_campo1;
    tipo_de_dado2 nome_do_campo2;
}
```

sintaxe de registos

Em suma, a *class* define um novo tipo de dados referência constituído por vários campos. A partir desta definição passa a existir um novo tipo de dados, sendo possível declarar variáveis deste novo tipo. O acesso a cada um dos campos faz-se através do nome do campo correspondente. Tenhamos como exemplo o seguinte caso:

```
class ComplexNumber {
    double realPart;
    double imaginaryPart;
}
```

exemplo de registo

Para declarar novas variáveis do tipo *ComplexNumber* temos que utilizar o operador **new**.

operador new

```
ComplexNumber num = new ComplexNumber();
```

exemplo de declaração

Aqui, *num* é uma variável que uma referência para um objeto criado do tipo *ComplexNumber*. Neste caso, o operador *new* vai reservar espaço na memória do computador para armazenar a variável, o que permite a posterior utilização da mesma para guardar os dados.

A atribuição de um valor a uma variável, dado este novo tipo de dados, é feita “chamando” a variável e o seu subtipo específico, separados por um ponto. Exemplifiquemos através de um exemplo parcial de um programa.

```
(...) { public static void main(String[] args) {
    ComplexNumber a, b;
    a = new ComplexNumber;
    b = new ComplexNumber;
    a.realPart = nextDouble();
    a.imaginaryPart = nextDouble();
}
```

exemplo de atribuição

**Nota!** Atenção à cópia de uma variável tipo referência: é necessário distinguir uma cópia de objeto, de uma cópia de referência, propriamente dita. Este é um dos erros frequentemente cometido pelos programadores.

nota

```
ComplexNumber x = new ComplexNumber;
ComplexNumber y = new ComplexNumber;
x.realPart = 10;
x.imaginaryPart = 20;
y = x; // estamos a copiar a referência, não o conteúdo
y.realPart = x.realPart; // cópia do campo realPart
y.imaginaryPart = x.imaginaryPart; // cópia do campo imag..Part
```

cópia de conteúdo  
e referência

## 5. Sequências de caracteres

Muitos programas, como requisito principal, não só pedem valores numéricos para serem processados, como texto. Por outras palavras, **sequências de caracteres** não é simplesmente uma sequência capaz de armazenar caracteres, pois estes têm particularidades especiais e necessitam de um conjunto de operações específicas para a sua manipulação. Tal como já foi referido atrás, em Java existe um tipo de dados de referência muito especial, que manipula texto: o ***String*** - este tipo de dados é promovido pela **classe *String*** que disponibiliza um vasto conjunto de funções para a sua execução. Uma outra classe de relevo, é a **classe *Character***. No apêndice A deste documento, podemos encontrar uma tabela de codificação de caracteres, muito conhecida, denominada de **ASCII**. A tabela ASCII (lê-se “ás-qui” e significa *American Standard Code for Information Interchange*) foi uma tabela criada como método para codificar todos os caracteres do inglês, incluindo sinais de pontuação e de moeda, entre outros. Esta tabela está definida como o conjunto de caracteres (***charset***) pertencente à classe *Character*.

sequências de caracteres

**String**  
**classe String**  
**classe Character**

**ASCII**

**charset**

### A classe “Character”

A classe *Character*, tal como enunciada atrás, é uma classe que contém um conjunto de funções para o processamento de caracteres. Estas funções são passíveis de serem divididas em dois grandes grupos: funções de teste de caracteres que devolvem um valor booleano se o argumento pertence ao grupo associado - como as funções *isLetter*, *isDigit*, *isLetterOrDigit*, *isSpace*, *isLowerCase*, *isUpperCase*, ... -, e funções de conversão que devolvem outro carater.

Tal como qualquer outra função, como as que aprendemos a criar na unidade anterior, estas também se utilizam da seguinte forma:

```
Character.nomeDaFuncao (...)
```

invocar função de Character

Vejamos o seguinte exemplo de aplicação. O programa que se segue traduz uma leitura de caracteres até aparecer um carater específico, o “.”.

```
...
char c;
do {
    System.out.print("Insert a character: ");
    c = sc.nextLine.charAt(0);
    if (Character.isLetter(c)) {
        System.out.println("You've inserted a letter."); }
    else if (Character.isDigit(c)) {
        System.out.println("You've inserted a digit."); }
    else {
        System.out.println("You've inserted anything."); }
} while (c != '.');
```

exemplo de aplicação com  
classe Character



## Operações com caracteres

Já que acabámos de estudar a classe *Character*, devemos saber o porquê do mesmo. Não só o porquê, mas o como, é o que vamos discutir nesta sub-unidade.

A classe *Character* é uma classe que é composta por objetos os quais efetuam, nada mais, nada menos, que **operações sob caracteres**. Mas nenhuma das operações que revimos serve para transformar um carater noutro carater. Para o fazer, necessitamos de nos recorrer ao código ASCII. Consideremos o seguinte exemplo que mostra o deslocamento de caracteres, três posições para a frente:

**operações sob caracteres**

```
if (Character.isLowerCase(letra)) {
    pos = (int) (letra - 'a'); // posição relativa da letra
    novaPos = (pos + 3) % 26; // deslocamento circular ??
    novaLetra = (char) ('a' + novaPos); // nova letra
}
else if (Character.isUpperCase(letra)) {
    pos = (int) (letra - 'A');
    novaPos = (pos + 3) % 26;
    novaLetra = (char) ('A' + novaPos);
} ...
```

**deslocamento de caracteres**

## Propriedades das Strings

Na linguagem Java, tal como já foi outrora referido, a sequência de caracteres é um tipo de dados de referência, pelo que adjacentes a essa natureza estão certas limitações, ao nível da alteração do seu conteúdo. Por outras palavras, e num termos mais estatístico, o maior problema na gestão das sequências de caracteres fica definida quando esta é criada, sendo que não nos é possível modificar mais tarde o seu conteúdo (é imutável). Na passagem como argumento a funções, apesar de ser um tipo referência, o seu conteúdo não pode ser modificado, tal como teremos a oportunidade de ver mais à frente.

A **declaração** de variáveis do tipo *String* obedece às mesmas regras de declaração dos tipos de dados referência. Assim:

**declaração**

```
String s1;
s1 = new String("Portugal"); // String com texto Portugal
String s2;
s2 = new String(); // String nula4
String s3 = "Aveiro"; // Declaração de String simplificada
```

**declaração de variáveis  
do tipo String**

## Leitura e escrita de Strings

Uma *String* pode ser lida do teclado através de uma função especial, designada por ***nextLine()***, da classe *Scanner*. Esta função lê todos os caracteres introduzidos pelo utilizador até encontrar um “\n” (mudança de linha).

**função *nextLine()***

Para imprimir no terminal o conteúdo de uma *String*, podemos utilizar qualquer das classes seguintes, tais como nós já as conhecemos: *System.out.print()*, *println()* e *printf()*. Neste último caso utilizamos o especificador de conversão “%s” para escrever uma *String*. Este pode ser precedido de um número com o qual toma um formato definido, como no seguinte exemplo:

```
System.out.printf("%10s", s);
System.out.printf("%-10s", s);
```

**especificadores de conversão  
de Strings numa printf**

4 Usando as potencialidades do software Dr.Java, disponibilizado gratuitamente em [www.drjava.org](http://www.drjava.org), no painel "Interações", podemos constatar que enquanto que os tipos de dados primitivos se inicializam com 0 (zero), os tipos de dados referência inicializam-se com o valor "null" (nulo).

# 41 PROGRAMAÇÃO I

Tomemos também como exemplo, mais completo:

```
String s = new String();
s = sc.nextLine();
System.out.printf("The read text is %s\n", s);
System.out.println("The read text is " + s);
```

exemplo de aplicação de  
diferentes formas de  
imprimir Strings

## A classe "String"

A classe *String* disponibiliza um vasto conjunto de funções que podemos aplicar e separar em dois tipos: funções que se aplicam sobre variáveis do tipo *String* - *variable.nameOfFunction()*;

```
char charAt(int) // devolve o carater numa determinada posição
int length() // devolve a dimensão de uma String
int indexOf(char) // pesquisa a primeira ocorrência do carater
boolean equals(String) // verifica se duas Strings são iguais
boolean compareTo(String) // compara duas Strings
```

várias funções da classe  
String

funções que se aplicam sem a necessidade de ter uma variável do tipo *String*: *String.nameOfFunction()*.

Vejamos o seguinte exemplo de aplicação onde usamos a classe *String* no algoritmo, para efetuar uma re-escritura dos caracteres de uma *String*, no programa.

```
String statement = new String();
char letter;
int i;
System.out.print("Insert a statement: ");
statement = sc.nextLine();
System.out.printf("This statement has the letters:\n");
for (i = 0; i < frase.length(); i++) {
    letter = statement.charAt(i);
    System.out.println(letter);
}
```

exemplo de aplicação com a  
classe String

## Passagem de Strings a funções

Na passagem de *Strings* como **argumento de funções**, apesar de ser um tipo de referência, o seu conteúdo não pode ser modificado, dado que são objetos imutáveis. Isto quer dizer que, quando atribuímos um novo valor a uma *string*, o seu **endereço** na memória do computador muda.

**argumento de funções**  
**endereço**

```
...
String phrase = new String("New Orleans");
f(phrase); // argumento da função passa a referenciar phrase
System.out.printf("%s\n", phrase); // imprime "New Orleans"
}
public static void f(String s) {
    s = "hello"; // s passa a referenciar algo diferente
    System.out.printf("%s\n", s);
}
```

passagem de Strings como  
argumento de funções

## 6. Arrays como sequências, vetores ou tabelas

Anteriormente vimos que é possível criar novos tipos de dados que permitem declarar variáveis onde é possível guardar mais do que um valor. No entanto, existem outras aplicações informáticas que precisam de lidar com grandes quantidades/volumes de dados, pelo que não é eficiente ter uma variável para cada "valor" a armazenar.

A linguagem Java disponibiliza, tais como muitas outras linguagens, outro tipo de dados referência, os **arrays**, os quais, em português muitas vezes são descritos como sequências, vetores ou tabelas, onde é possível numa só variável armazenar vários

**arrays**

valores do mesmo tipo. Como variável do tipo referência, o nome da variável é apenas uma referência para uma zona de memória que será reservada para uma zona de memória que será reservada posteriormente com o operador *new*.

## Introdução aos arrays

Um *array*, tal como já foi dito, é uma organização de memória que se caracteriza pelo facto de ser um **agregado de células** contíguas, capaz de armazenar um conjunto de valores do mesmo tipo e aos quais se pode aceder de forma indexada.

agregado de células

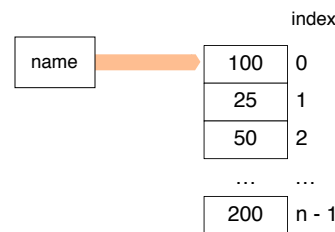


figura 7

conceito de array

Um *array* é identificado pelo nome da variável e o acesso a cada elemento é feito através da respetiva posição. A sua declaração é feita da seguinte forma:

```
type identifier[]; // qualquer tipo
identifier = new type[dimension];
```

sintaxe de declaração de array

Vejamos os seguintes exemplos

```
double x[]; // array real
x = new double[3]; // x[] tem 3 elementos: x[0], x[1] e x[2]

int y[]; // array inteiro
y = new int[4]; // y[] tem 4 elementos: y[0], y[1], y[2] e y[3]

char z = new char[2]; // array de 2 caracteres z[0] e z[1]
```

exemplos de aplicação

## Acesso aos elementos de um array

O tipo *array* é concebido como um conjunto de tipos base, sendo apenas possível processar um elemento de dados de cada vez. Um elemento de um *array* é acedido usando *identifier[index]*. Em Java, os índices são sempre valores numéricos inteiros positivos, sendo o primeiro elemento o zero e o último *n-1*, sendo *n* a dimensão do *array*. O **índice** pode ser também usado através de uma expressão cujo resultado tem que ser inteiro.

índice

Caso se tente referenciar um elemento fora do *array* (índice inferior a zero ou maior que *n-1*), é gerado um **erro** na execução do programa, dizendo “acesso fora dos limites”.

erro

Uma variável do tipo *array* distingue-se de uma variável simples devido ao uso do **operador** “[ ]”, na sua declaração.

operador “[ ]”  
length

A linguagem Java associa a cada *array* um campo de dimensão (*length*) que pode ser usado sempre que seja necessário determinar a sua capacidade de armazenamento. Pode ser usado da forma *identifier.length*. É também possível declarar e atribuir um conjunto de valores a um *array*, através de uma expressão de inicialização da seguinte forma:

```
int numeros[] = {1, 2, 3, 4, 5, ..., 30}
```

inicialização de um array

## Leitura e escrita do conteúdo de arrays

Para escrever num *array*, basta atribuir a  $a[index]$ , através do operador “=”, de atribuição. Vejamos o seguinte exemplo:

```
final int dimension = 10;
int a[];
a = new int[dimension];

/* leitura do conteúdo de um array */
for (int i = 0; i < dimension; i++) {
    System.out.print("Valor para posição " + i);
    a[i] = sc.nextInt();
}

/* escrita de um array */
for (int k = 0; k < dimension; k++) {
    System.out.printf("a[%d] contém o valor %d\n", k, a[k]);
}
```

leitura e escrita de conteúdo  
num array

## Passagem de arrays a funções

Uma variável do tipo *array*, é sempre passada por **referência** a uma função. De facto, estamos a passar o endereço do início do *array* em memória (como nos registos). Deste modo, uma variável do tipo *array* é sempre um argumento de entrada/saída, podendo o seu conteúdo ser modificado dentro da função.

referência

Já dentro de uma função, podemos saber com quantos elementos foi criado um *array* através do campo *length*. No entanto, nem sempre uma sequência está preenchida, pelo que nessas circunstâncias é usual utilizar uma variável inteira adicional, para além do *array*, onde armazenamos o número de elementos preenchidos. Confirmemos isso no seguinte exemplo:

```
/* Leitura de valores até zero */
public static int readSequence(int a[]) {
    int n = 0;
    do {
        System.out.print("Integer value: ");
        tmp = sc.nextInt();
        if (tmp != 0) {
            a[n] = tmp; // armazena o valor na posição n
            n++; // avança para a posição seguinte
        }
    } while (tmp != 0 && n < a.length);
    return n; // devolve o número de valores lidos
}

public static void printSequence(int a[], int n) {
    for (int i = 0; i < n; i++) {
        System.out.printf("a[%d] contains %d\n", i, a[i])
    }
}

public static double calcAverage(int a[], int n) {
    int sum = 0;
    double m;
    for (int i = 0; i < n; i++) {
        sum += a[i];
    }
    m = (double)sum / n;
    return m;
}
```

exemplo de aplicação

## Arrays como valor de retorno de uma função

O valor de **retorno** de uma função pode ser de qualquer tipo de dados (primitivo ou referência). Supondo que queríamos copiar o conteúdo de um *array*, para outro, podíamos implementar a seguinte função:

```
public static int[] copyArrays(int[] a, int n) {
    int tmp[] = new int[n];
    for (int i = 0; i < n; i++) {
        tmp[i] = a[i];
    }
    return tmp;
}
```

retorno com arrays

A outra alternativa seria ter uma função que recebia como argumento, dois *arrays* já criados.

## 7. Ficheiros de texto

Em todos os programas desenvolvidos até ao momento, a informação manipulada era perdida sempre que terminávamos os programas. Isto deve-se ao facto de as variáveis que declarámos, reservarem espaço na memória do computador, que depois é libertada quando o programa termina. De forma a armazenarmos permanentemente **informação** gerada pelos nossos programas, temos que a guardar no disco rígido do computador (ou em qualquer outro dispositivo de memória em massa). Isto é possível através da utilização de **ficheiros**.

informação

ficheiros

### Ficheiros e Diretórios

Sabendo que é possível armazenar conteúdos trabalhados num programa, em ficheiros, só nos resta, de facto, saber o que é um ficheiro. Um ficheiro é uma estrutura de armazenamento de informação, **codificada** numa sequência de 0's e 1's (informação binária). Um **diretório** é um tipo especial de ficheiro, o qual suporta uma lista de referências a ficheiros.

codificada

diretório

Os ficheiros e os diretórios têm duas características muito importantes e únicas: a sua localização é no sistema de ficheiros (diretório e nome); têm a si associadas várias permissões de leitura, escrita e execução, tal como estudámos na segunda unidade deste documento (ver §O sistema UNIX).

### Ficheiros de texto em Java

Em Java é-nos permitido trabalhar com ficheiros de texto, lendo e escrevendo-os, em pleno programa. A **classe *File***, localizada no *package java.io.File*, permite-nos confirmar a existência de ficheiros, verificar e modificar as permissões de ficheiros, verificar qual o tipo de ficheiro (diretório, ficheiro normal, ...), criar diretórios, listar o conteúdo de diretórios, apagar ficheiros, entre muitas outras tarefas...

classe File

Os dados são interpretados então, e transformados, de acordo com determinados formatos de representação de texto. Cada carater é codificado através de tabelas de codificação como é exemplo o ASCII, já referido, ou o Unicode, UTF-8, ... Em Java, os tipos de dados *char* e *String* codificam o texto com a codificação **Unicode** de 16 bits. Esse detalhe de implementação do Java é, no entanto, transparente para o programador. Os métodos ou funções de entrada/saída fazem automaticamente a

Unicode

tradução de, ou para a codificação escolhida. Existem também constantes literais para alguns caracteres especiais, nossos já conhecidos, como o “\n”, “\t” ou “\”.

Por outro lado, a **classe *PrintWriter***, pertencente ao pacote genérico *java.io*, tem uma interface muito semelhante à do *PrintStream* (*System.out*). Aqui, a sua utilização é muito relevante, dado que é exatamente esta classe a que nos permite criar uma entidade (objeto) *File* associado ao nome do ficheiro de texto desejado.

**classe *PrintWriter***

```
File fout = new File(fileName)
```

**criar objeto *File***

A declaração e criação de um objeto do tipo *PrintWriter* associado a esse objeto *File* faz-se da seguinte forma:

```
PrintWriter pwFile = new PrintWriter(fout)
```

**criar objeto *PrintWriter***

Para escrever no ficheiro e depois fechá-lo, de forma a concluir a edição do mesmo, escrevem-se as seguintes instruções:

```
pwFile.println(line); // escrever sobre o ficheiro
pwFile.close(); // fechar o ficheiro
```

**escrever e fechar ficheiros**

Por vezes a utilização destas instruções falha, pelo que, para lidar com este tipo de situações, a linguagem Java utiliza uma aproximação defensiva gerando (*checked*) exceções. A classe *PrintWriter*, da biblioteca Java, obriga o programador a lidar explicitamente com a **exceção *IOException***. Para evitar este erro, programa-se para que se possa ignorar a exceção, usando a **palavra reservada *throws***, como continuação do programa-mãe, escrevendo *public static void main(String[] args) throws IOException { ...*

**exceção *IOException***

**palavra reservada *throws***

A leitura de ficheiros de texto em Java é realizada através de um tipo de dados já conhecido, o *Scanner*. Como desta vez nos interessa ler um ficheiro, em vez da entrada via teclado, de dígitos, ao invés de colocarmos *System.in*, colocamos antes o ficheiro de tipo *File*, enunciado antes. Assim, e traduzindo os exemplos acima para a leitura, obtemos as seguintes instruções:

```
File fin = new File(fileName); // criar objeto File
Scanner scf = new Scanner(fin); // ler ficheiro fin
while (scf.hasNextLine()) {
    String line = scf.nextLine(); // ler do ficheiro
}
scf.close(); // fechar o ficheiro
```

**ler um ficheiro *fin***

Vejamos o seguinte exemplo, ordenado:

```
Scanner kb = new Scanner(System.in);
System.out.print("Input File: ");
String nameIn = kb.nextLine();
File fin = new File(nameIn);
Scanner scf = new Scanner(fin);

System.out.print("Output filename: ");
String nameOut = kb.nextLine();
File fout = new File(nameOut);
PrintWriter pw = new PrintWriter(fout);

while (scf.hasNextLine()) {
    pw.println(scf.nextLine());
}
scf.close();
pw.close();
```

**exemplo de aplicação completo**

## 8. Pesquisa e ordenação de valores em arrays

Em inúmeros problemas temos a necessidade de procurar por valores em sequências. A esta tarefa designamos de pesquisa.

Existem vários algoritmos em programação para a pesquisa de valores em sequências, mas nesta disciplina apenas vamos analisar dois dos mais simples: a **pesquisa sequencial** e a **pesquisa binária**.

A pesquisa é uma tarefa computacionalmente dispendiosa se estivermos a tratar grandes quantidades de informação. O desenvolvimento de algoritmos eficientes torna-se essencial e, como vamos reparar, a complexidade dos algoritmos não é sempre a mesma.

**pesquisa sequencial, pesquisa binária**

### Pesquisa sequencial

Uma pesquisa sequencial consiste em analisar todos os elementos de uma sequência, de forma metódica. Assim, a pesquisa começa por analisar o primeiro valor da sequência e percorre todos os seus valores até encontrar o valor pretendido, ou até atingirmos o último elemento. Este método é normalmente moroso, dependendo da dimensão da sequência, e não do arranjo dos valores.

Lembremo-nos, enquanto fazemos um programa, de criar uma forma de sinalizar a não localização (não encontrar) de um determinado valor pretendido.

Vejamos o seguinte exemplo:

```
public static int ArraySearch(int seq[], int nElem, int value) {
    int pos = -1; // inicializamos com um valor inválido
    for (int i = 0; i < nElem; i++) {
        if (seq[i] == value) {
            pos = i;
            break;
        }
    }
    return pos;
}
```

**exemplo de pesquisa sequencial**

No exemplo dado temos uma função que devolve um valor inteiro e que requer três valores de entrada, entre os quais um *array* de inteiros, e dois números inteiros. O que vai acontecer é que na dimensão de toda a sequência *seq[ ]*, com uma instrução de repetição *for*, iniciada pela variável *i*, até que esta não seja menor que *nElem*, vai ver se o elemento do *array* no índice *i* é igual ao valor pretendido para pesquisa. Caso assim seja, ele vai guardar, como posição do valor encontrado, o valor de *i*. Encontrado o valor, a repetição termina, pela instrução *break*, o qual força a devolução do valor de *pos*. Constatemos que a variável *pos* se iniciou com um valor inválido, dado que não existe posição -1. Assim, podemos criar uma regra de validação que nos permitiria verificar se o valor não foi encontrado, com uma condição que se *pos == -1*, então apareceria no terminal uma mensagem do género “Valor não encontrado.”.

### Pesquisa binária

Se tivermos **informação a priori** sobre os elementos de uma determinada sequência, podemos acelerar o processo de pesquisa. Por exemplo, se uma sequência se encontrar ordenada por ordem crescente, ou decrescente, podemos fazer pesquisa binária. Assim, o algoritmo começa por selecionar o elemento central da sequência e compara-o com o elemento procurado. Se o elemento for maior, podemos excluir a primeira metade da sequência, caso contrário podemos excluir a segunda metade. Este

**informação a priori**

processo é repetido até que o elemento seja encontrado ou até deixarmos de ter elementos para analisar. Vejamos o seguinte exemplo:

```
public static int BinarySearch(int seq[], int nElem, int value) {
    int pos = -1;
    int beginning = 0, end = nElem - 1, middle;
    while (beginning <= end) {
        middle = (end + beginning) / 2;
        if (seq[middle] == value) {
            pos = middle;
            break;
        }
        if (seq[middle] > value) {
            end = middle - 1;
        }
        else {
            beginning = middle + 1;
        }
    }
    return pos;
}
```

exemplo de pesquisa binária

Como podemos visualizar, voltámos a inicializar a variável *pos* com um valor inválido para a sua gama de valores possíveis, e depois criámos três variáveis centrais neste tipo de pesquisa: o *beginning*, o *end* e o *middle*. Enquanto que o início (*beginning*) seja mais pequeno que o fim (*end*) da sequência, o programa primeiro verifica se o elemento do meio (*middle*) é o valor que procuramos - caso seja, ele tomará o valor de *pos* como o valor do índice de *middle* - e depois verifica se o elemento do meio é maior que o valor que pretendemos encontrar. Caso aconteça o do meio ser maior que o valor pretendido, o programa irá re-configurar a variável *end* para *middle* - 1, o valor anterior a *middle*, pelo que podemos constatar que estamos perante uma sequência em ordem crescente. Se nenhuma das duas primeiras condições funcionar então o programa avança para a outra metade da sequência, repetindo todo este algoritmo até se encontrar o valor pretendido, ou até não haver mais nenhum valor para analisar.

## Ordenação de sequências

Em muitos outros problemas, para além da pesquisa de valores, temos a necessidade de ordenar sequências. Existem, também neste caso, imensos algoritmos em programação para a ordenação de sequências, mas nesta disciplina vamos apenas analisar dois: a **ordenação sequencial** e a **ordenação por flutuação**.

Na ordenação sequencial vamos colocando em cada posição da sequência, o valor correto, começando no primeiro. Pelo contrário, na ordenação por flutuação iremos comparando pares de valores da sequência e trocamos se estes tiverem fora de ordem. Este processo repetir-se-á enquanto houver necessidade de trocas. Vejamos um exemplo de ordenação sequencial:

ordenação sequencial,  
ordenação por flutuação

```
public static void SeqOrder(int seq[], int n) {
    int tmp, i, j;
    for (i = 0; i < n - 1; i++) {
        for (j = i + 1; j < n; j++) {
            if (seq[i] > seq[j]) {
                tmp = seq[i];
                seq[i] = seq[j];
                seq[j] = tmp;
            }
        }
    }
}
```

exemplo de ordenação  
sequencial



Analisando o exemplo de ordenação sequencial temos que enquanto fixamos uma posição na sequência `seq[ ]`, através da primeira instrução de repetição *for*, vamos percorrendo as outras através da segunda instrução *for*, na qual, se o elemento onde nos fixamos for maior ao que estamos a analisar, trocamos de ordem.

Atentemos agora no seguinte exemplo, onde se tenta demonstrar a ordenação por flutuação.

```
public static void FloatOrder(int seq[], int n) {
    int tmp, i, j;
    boolean changes;
    do {
        changes = false;
        for (i = 0; i < n - 1; i++) {
            if (seq[i] > seq[i+1]) {
                tmp = seq[i];
                seq[i] = seq[i+1];
                seq[i+1] = tmp;
                changes = true;
            }
        }
    } while (changes);
}
```

exemplo de ordenação  
por flutuação

A ordenação por flutuação tem como algoritmo, a seguinte base do exemplo acima: partindo do princípio que já não há trocas a fazer (*changes = false*), para cada *i* de 0 até *n - 1*, o programa experimentará comparar um elemento ao seu seguinte, trocando se for necessário. Caso haja trocas, o programa irá repetir constantemente o processo.

## Utilização de pesquisa e ordenação

Até aqui foram dados exemplos de dois tipos de pesquisa e de dois tipos de ordenação, mas a sua aplicação não foi feita... Se repararmos melhor nos exemplos, podemos constatar que se tratam de funções (métodos), os quais têm de ser invocados no programa. Vejamos como invocar ambos os tipos de pesquisa que estudámos:

```
...
System.out.print("Insert a value to search: ");
value = sc.nextInt();
index = ArraySearch(seq_main, n_main, value);
// index = BinarySearch(seq_main, n_main, value);
if (index != -1) {
    System.out.println("The number is in the position " + index);
}
else {
    System.out.println("The number doesn't exist");
}
...
```

invocação de pesquisas

Do mesmo modo, vejamos um modo de aplicação da ordenação:

```
...
int nElem = 0;
int seq[] = new int[100];

nElem = Read(seq);
Write(seq, nElem);
SeqOrder(seq, nElem);
// FloatOrder(seq, nElem);
Write(seq, nElem); // os valores serão mostrados por ordem
...
```

invocação de ordenação

## 9. Arrays de strings, registos e bidimensionais

Até agora estudámos que podemos armazenar, numa variável do tipo *array*, vários valores do mesmo tipo. Neste último capítulo veremos que também podemos ter *arrays* de tipos referência, sendo que numa variável deste tipo podemos guardar várias referências de um certo tipo de dados. Assim, estudaremos ***arrays de Strings***, ***arrays de registos*** e ***arrays bidimensionais*** (*arrays de arrays*). A criação deste tipo de *arrays* necessita de uma declaração semelhante à dos *arrays* de tipos primitivos, mas cada seu elemento necessita depois, de ser “criado” segundo as regras correspondentes ao tipo de dados em causa.

**arrays de Strings**  
**arrays de registos, arrays**  
**bidimensionais**

### Arrays de strings

É então possível criar uma sequência de *Strings*, ou seja, uma sequência bidimensional de caracteres. A declaração de uma sequência de *Strings* cria um *array* de referências nulas para *String* que depois serão preenchidas por instruções de atribuição.

```
String cities[];
cities = new String[3];
cities[0] = "New Orleans";
cities[1] = "New York";
cities[2] = "San Francisco";

/* ou em alternativa */
String cities[] = {"New Orleans", "New York", "San Francisco"};
```

sintaxe de arrays de Strings

Vejamos o seguinte exemplo de aplicação, no qual o programa deve ler frases até aparecer a palavra “fim”.

```
public static int readSent(String sents[]) {
    String s = new String();
    int n = 0;
    do {
        System.out.print("Sentence: ");
        s = sc.nextLine();
        if (is.equalsIgnoreCase("fim")) {
            sents[n] = s;
            n++;
        }
    } while (is.equalsIgnoreCase("fim") && n < sents.length);
    return n;
}

public static void printSent(String sents[], int n) {
    for (int i = 0; i < n; i++) {
        System.out.printf("[%d] -> %s\n", i, sents[i]);
    }
}
```

exemplo de aplicação

### Arrays de Registos

Uma maneira de armazenar informação em aplicações reais consiste na utilização de sequências de registos, normalmente designadas por bases de dados. A declaração de *arrays* de registos é em tudo semelhante às sequências de tipos primitivos ou *Strings*, com a exceção de que esta tem de ser decomposta em duas operações: a primeira consiste em criar a sequência de referências para os futuros elementos do tipo registo; e a segunda consiste em criar os elementos propriamente ditos, seguindo a regra para a criação de variáveis do tipo registo.

Vejamos o seguinte exemplo, onde se produz uma declaração de um *array* de números complexos:

```

/* declaração de um array de números complexos
Complex a[] = new Complex[3];
a[0] = new Complex();
a[1] = new Complex();
a[2] = new Complex();

```

sintaxe de arrays de registos

Vejamos agora um exemplo completo, onde o objetivo do programa é ler pontos até aparecer o ponto (0,0).

```

... main ... {
    Point2D points[] = new Point2D[10];
    Point2D p;
    int n = 0;
    do {
        System.out.println("Insert a point: ");
        p = readPoint();
        if (p.x != 0 || p.y != 0) {
            points[n] = p;
            n++;
        }
    } while ((p.x != 0 || p.y != 0) && n < points.length);
    printPoints(points, n);
}

public static Point2D readPoint() {
    Point2D tmp = new Point2D();
    System.out.print("X coordinates: ");
    tmp.x = sc.nextDouble();
    System.out.print("Y coordinates: ");
    tmp.y = sc.nextDouble();
    return tmp;
}

public static void printPoints(Point2D a[], int n) {
    for (int i = 0; i < n; i++) {
        System.out.printf("pnt %d:(%.1f, %.1f)\n", i, a[i].x,
                           a[i].y);
    }
}

class Point2D {
    double x, y;
}

```

exemplo de aplicação

## Arrays bidimensionais

Existem problemas em que a informação a ser processada é mais bem representada através de uma estrutura de dados com um formato bidimensional, como é exemplo, uma matriz. Uma sequência bidimensional é na prática, uma sequência de sequências. A sua declaração respeita as regras de uma sequência unidimensional, sendo a única diferença o facto de usar dois operadores sequência seguidos “[ ]”]. Pode ser vista como uma estrutura matricial de elementos, composta por linhas e colunas, sendo o acesso a cada um dos elementos feito através de dois índices.

```

int m[][] = new int[3][3];
m[0][0] = 5; // elemento na linha 0, coluna 0
m[0][2] = 10; // elemento na linha 0, coluna 2
m[2][2] = 50; // elemento na linha 2, coluna 2
m.length; // número de linhas
m[0].length; // número de colunas para a linha 0

```

sintaxe de arrays  
bidimensionais

Vejamos o último exemplo, para aplicação de *arrays* bidimensionais:

```

public static void readMatrix(int m[][]) {
    for (int l = 0; l < m.length; l++){
        for (int c = 0; c < m[l].length; c++) {
            System.out.print("pos [" + l + "][" + c + "]: ");
            m[l][c] = sc.nextInt();
        }
    }
} // continua na página seguinte

```

exemplo de aplicação

```
}  
public static void printMatrix(int m[][]) {  
    for (int l = 0; l < m.length; l++){  
        for (int c = 0; c < m[l].length; c++) {  
            System.out.printf(" %5d", m[l][c]);  
        }  
    }  
    System.out.println();  
}
```

E assim terminou a disciplina de Programação I, dando os requisitos necessários para a entrada na disciplina de Programação II, do segundo semestre, na qual, os seus apontamentos se traduzem numa continuação deste documento.

## Introdução aos Computadores

Unidade de Processamento Central (CPU).....	2
Bytes.....	3
Memória.....	4
Armazenamento em Massa.....	5
Sistemas Óticos.....	7
Dispositivos amovíveis (Flash Drives) .....	8
Escrita e leitura de ficheiros .....	8
Dispositivos de comunicação .....	9

## Introdução à Programação

Conceito de programa.....	10
O sistema UNIX .....	11
Linguagens de programação, compiladores e intérpretes .....	15
O Java bytecode .....	16

## Introdução à linguagem Java

Estrutura de um programa.....	17
Elementos básicos da linguagem Java.....	19
Tipos de dados primitivos .....	19
Conversões.....	21
Operadores.....	21
Leitura e escrita de dados.....	23
Escrita formatada .....	23
Estruturas de controlo e estruturas de decisão.....	23
Instrução de decisão if.....	23
Instrução de repetição while .....	24
Análise de um problema e criação de um algoritmo .....	27
Instrução de repetição for (contador).....	31
Instruções break e continue .....	32

## Introdução à programação modular

Métodos void .....	36
Estruturas de Dados .....	37

## Sequências de caracteres

A classe "Character" .....	39
Operações com caracteres.....	40
Propriedades das Strings.....	40
Leitura e escrita de Strings .....	40
A classe "String" .....	41
Passagem de Strings a funções.....	41

## Arrays como sequências, vetores ou tabelas

Introdução aos arrays.....	42
Acesso aos elementos de um array .....	42
Leitura e escrita do conteúdo de arrays .....	43
Passagem de arrays a funções.....	43
Arrays como valor de retorno de uma função.....	44

## Ficheiros de texto

Ficheiros e Diretórios .....	44
Ficheiros de texto em Java.....	44

## Pesquisa e ordenação de valores em arrays

Pesquisa sequencial .....	46
Pesquisa binária .....	46
Ordenação de sequências .....	47
Utilização de pesquisa e ordenação .....	48

## Arrays de strings, registos e bidimensionais

Arrays de strings .....	49
Arrays de Registos .....	49
Arrays bidimensionais .....	50

1ª edição. O conteúdo deste documento foi escrito, em grande parte, por Rui Lopes, tendo, mais em especial as unidades 5, 6, 7, 8 e 9, respetivamente "Sequências de Carateres", "Arrays como sequências, vetores ou tabelas", "Ficheiros de Texto", "Pesquisa e Ordenação de valores em arrays" e "Arrays de Strings, registos e bidimensionais", parcialmente copiadas dos apontamentos partilhados pelo professor doutor António J. R. Neves, regente da disciplina de Programação I, da Universidade de Aveiro, no ano letivo de 2013/2014. Algumas imagens provêm de referências posteriores, as quais também foram base para toda a criação deste documento, algumas delas como bibliografia base desta disciplina. "Problem Solving with JAVA", KOFFMAN, Elliot. Addison-Wesley; "Introduction JAVA Programming", LIANG, Y. Daniel. Pearson, Prentice-Hall; "Introdução à Programação em Java", ROCHA, António Adrego et al. FCA; "Computer Science, An overview", BROOKSHEAR, James. Addison-Wesley. Todas estas obras foram consultadas entre 16 de setembro de 2013 e 7 de janeiro de 2014. @Apple, @Safari, @Oracle e @Java são marcas registadas. © Rui Lopes 2014 Copyright: Pela Creative Commons, não é permitida a cópia e a venda deste documento. Qualquer fraude será punida. Respeite os autores e as suas marcas. Original - This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit [http://creativecommons.org/licenses/by-nc-nd/4.0/deed.en\\_US](http://creativecommons.org/licenses/by-nc-nd/4.0/deed.en_US).



**The ASCII Table**

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
00	00	NUL	32	20	SP	64	40	@	96	60	'
01	01	SOH	33	21	!	65	41	A	97	61	a
02	02	STX	34	22	"	66	42	B	98	62	b
03	03	ETX	35	23	#	67	43	C	99	63	c
04	04	EOT	36	24	\$	68	44	D	100	64	d
05	05	ENQ	37	25	%	69	45	E	101	65	e
06	06	ACK	38	26	&	70	46	F	102	66	f
07	07	BEL	39	27	'	71	47	G	103	67	g
08	08	BS	40	28	(	72	48	H	104	68	h
09	09	HT	41	29	)	73	49	I	105	69	i
10	0A	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	DEL