Programação II
Guião das Aulas Práticas

Departamento de Electrónica, Telecomunicações e Informática
Universidade de Aveiro

2018–2019

Resumo

Este guião propõe problemas para serem resolvidos pelos alunos de Programação II durante as respectivas aulas práticas e fora delas.

Aula Prática 0

Introdução ao UNIX

Resumo:
- Introdução ao sistema operativo UNIX.
- O ambiente de trabalho para as aulas práticas.
- Edição, compilação e execução de programas em Java.

As aulas práticas de Programação II decorrem em salas equipadas com computadores pessoais correndo o sistema operativo Linux. O Linux (ou mais correctamente, GNU/Linux) é uma variante *livre* e *gratuita* do conhecido sistema operativo UNIX. O número de instalações deste sistema tem registado um crescimento impressionante nos últimos anos, sendo actualmente um dos sistemas operativos mais populares em computadores pessoais, juntamente com o Windows e o MAC OS.[1] Dada a sua arquitectura aberta, existem inúmeras variantes de Linux adaptadas a diferentes utilizações que correm numa grande diversidade de máquinas desde computadores de bolso a servidores de Internet ou supercomputadores para cálculo científico. Na Universidade de Aveiro, um grupo de utilizadores de Linux denominado GLUA[2] disponibiliza diversas distribuições populares de Linux e organiza sessões de esclarecimento e de ajuda para quem estiver interessado em instalar e utilizar este sistema.

0.1 O Arranque, *Login* e *Logout*

Os computadores das salas de aula têm actualmente dois sistemas operativos instalados: o Windows e o Linux.

Assim, ao ligar o computador será confrontado com um menu para escolher o sistema que deseja iniciar. Terá alguns segundos para escolher a opção certa (o Linux, neste caso), usando as teclas de direcção ↑ ou ↓ e a tecla `Enter` ↩. Se o computador já se encontrar ligado e a correr Windows, deverá seleccionar a opção para reiniciar e poder voltar ao menu de arranque.

[1] As últimas versões do MAC OS também implementam uma variante de UNIX.
[2] http://glua.ua.pt

Logo que o sistema esteja em funcionamento, aparece um ecrã de boas-vindas onde terá de se identificar, introduzindo o *nome-de-utilizador* (*username*) do tipo `a12345` (sem `@ua.pt`), e a *palavra-passe* (*password*) correspondente. Estes dados são os mesmos que utiliza para aceder ao ambiente Windows. Se introduziu os dados correctos, surge um ambiente gráfico que lhe permite interagir com o sistema e completar os exercícios da aula. Chama-se *entrar no sistema* (em Inglês *log in* usualmente escrito *login*) a este processo de autenticação para ter acesso ao sistema.

Quando terminar de usar o sistema, deve sempre *sair do sistema* (*log out* ou *logout*) de forma a que mais ninguém tenha acesso à sua área de trabalho. Se quiser desligar ou reiniciar o computador deve escolher a acção desejada no ecrã de boas-vindas que entretanto reaparece.

Exercício 0.1

Entre no sistema, introduzindo o seu nome-de-utilizador e palavra-chave na janela de *login*. Explore os menus e ícones do ambiente gráfico. Descubra a opção de *Log Out* (geralmente *System/Quit/Log Out*) e seleccione-a para sair do sistema. Repita o processo de *login* para regressar ao sistema.

0.2 A Linha de Comandos UNIX

Quando o sistema UNIX foi concebido, os computadores eram controlados essencialmente através de *consolas* ou *terminais* de texto: dispositivos dotados de um teclado e de um ecrã onde se podia visualizar somente texto. A interacção com o sistema fazia-se tipicamente através da introdução de comandos escritos no teclado e da observação da resposta produzida no ecrã pelos programas executados. Actualmente existem ambientes gráficos que correm sobre o UNIX e permitem visualizar informação de texto e gráfica, e interagir por manipulação virtual de objectos gráficos recorrendo a um rato e ao teclado. É o caso do Sistema de Janelas X, ou simplesmente X, que está instalado nos computadores das salas de aula. Apesar das novas formas de interacção proporcionadas pelos ambientes gráficos, continua a ser possível e em certos casos preferível usar a interface de *linha de comandos* para muitas operações. No X, isto pode fazer-se usando um *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.

Exercício 0.2

Abra uma janela de terminal (a partir do menu principal)[3] e quando surgir o *prompt*[4] execute o comando `date`.

Observe que a resposta foi impressa imediatamente a seguir à linha do comando, de forma concisa, sem distracções nem grandes explicações. Este comportamento é usual

[3]Possivelmente: *Applications/Accessories/Terminal*.

[4]http://pt.wikipedia.org/wiki/Prompt

em muitos comandos UNIX e é típico de um certo estilo defendido pelos criadores deste sistema. Simples, mas eficaz.

Exercício 0.3
Execute o comando `cal` e observe o resultado. Descubra em que dia da semana nasceu, passando o mês e o ano como *argumentos* ao comando `cal`, por exemplo: `cal jan 1981`.

Os comandos em UNIX têm sempre a forma:

```
comando argumento1 argumento2 ...
```

onde `comando` é o nome do programa a executar e os argumentos são cadeias de caracteres, que podem ser incluídas ou não, de acordo com a sintaxe esperada por esse programa.

Na linha de comandos é possível recapitular um comando dado anteriormente usando as teclas de direcção ↑ e ↓. 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`.

0.2.1 Navegação no Sistema de Ficheiros

Tal como noutros sistemas operativos, no UNIX a informação é armazenada numa estrutura hierárquica formada por directórios, subdirectórios e ficheiros. O directório-raiz desta árvore é representado simplesmente por uma barra "/". Cada utilizador possui um directório próprio nesta árvore, a partir do qual pode (e deve) criar e gerir toda a sua subárvore de directórios e ficheiros: é o chamado *directório do utilizador* ou *home directory*. Após a operação de *login* o sistema coloca-se nesse directório. Portanto neste momento deve ser esse o *directório actual* (*current directory*). Para saber qual é o directório actual execute o comando **pwd**. Deve surgir um nome como

```
/homermt/a12345
```

que indica que está no directório `a12345` que é um subdirectório de `homermt` que é um subdirectório directo da raiz `/`. Para listar o conteúdo do directório actual execute o comando **ls**. Deve ver uma lista dos ficheiros (e subdirectórios) contidos no seu directório neste momento, por exemplo:

```
arca  Desktop  Examples
```

Neste caso, observam-se dois subdirectórios e um *soft link* que é um tipo de ficheiro especial que serve de atalho para outro ficheiro ou directório. Dependendo da configuração do sistema, os nomes nesta listagem poderão aparecer com 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. (Num ambiente gráfico a mesma informação está disponível numa representação mais visual. Experimente, por exemplo, escolher *Places/Home Folder* para ver o conteúdo do seu directório pessoal.)

Ficheiros cujos nomes começam por "." não são listados por defeito, são ficheiros *escondidos*, usados geralmente para guardar informações de configuração de diversos programas. Para listar todos os ficheiros de um directório, incluindo os escondidos, deve executar a variante **ls -a**.

Por vezes é necessário listar alguns atributos dos ficheiros para além do nome. Pode fazê-lo executando as variantes **ls -l** ou **ls -la**.

```
total 88
  drwx------ 13 a12345 users 4096 2007-01-26 14:03 .
  drwxr-xr-x  3 root   root  4096 2007-01-25 10:52 ..
  drwx------  1 a12345 users    0 2007-01-26 08:00 arca
  drwxr-xr-x  2 a12345 users 4096 2007-01-25 10:52 Desktop
  lrwxrwxrwx  1 a12345 users   26 2007-01-25 10:52 Examples -> ...
```

Os principais atributos mostrados nestas listagens longas são:

Tipo de ficheiro identificado pelo primeiro carácter à esquerda, sendo `d` para directório, `-` para ficheiro normal, `l` para *soft link*, etc.

Permissões representadas por 3 conjuntos de 3 caracteres. 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 da máquina.

Propriedade indica a que utilizador e a que grupo pertence o ficheiro.

Tamanho em número de bytes.

Data e hora da última modificação.

Nome do ficheiro.

Normalmente existe um *alias*[5] `ll` equivalente ao comando `ls -l`.

Além do `ls` e variantes, existem outros comandos importantes para a observação e manipulação de directórios, por exemplo:

`cd` — o directório actual passa a ser o directório do utilizador.

`cd dir` — o directório actual passa a ser o directório `dir`.

`mkdir dir` — cria um novo directório chamado `dir`.

`rmdir dir` — remove o directório `dir`, desde que esteja vazio.

[5]Um *alias* é um nome alternativo usado em representação de um determinado comando. São criados usando o comando interno **alias**.

O argumento `dir` pode ser dado de uma forma absoluta ou relativa. Na forma absoluta, `dir` identifica o caminho (*path*) para o directório 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 directório pretendido a partir do directório actual; tem a forma `subdir1/.../subdirN`. Há dois nomes especiais para directórios: "`.`" e "`..`" que representam respectivamente o directório actual e o directório pai, ou seja, o directório ao qual o actual pertence.

Exercício 0.4

Execute os comandos seguintes e interprete os resultados:

```
ls -l /
cd /
pwd
ls -l
cd usr
ls
cd local/src
pwd
ls
cd ../../bin
ls
cd
pwd
```

Exercício 0.5

Experimente utilizar o programa gráfico gestor de ficheiros[6] para navegar pelos mesmos directórios que no exercício anterior: `/`, `/usr`, `/usr/local/src`, etc.

Exercício 0.6

Mude o directório actual para o seu subdirectório `arca`. Liste o seu conteúdo. Reconhece algum dos ficheiros?

Importante: O subdirectório `arca` não é um directório local do PC onde está a trabalhar; é na verdade a sua área privada de armazenamento no Arquivo Central de Dados (ARCA[7]), um servidor de ficheiros da Universidade de Aveiro. Esta área também é acessível a partir do ambiente Windows e através da Web, e é natural que já aí tenha colocado ficheiros noutras ocasiões. É neste directório que deve gravar os ficheiros e directórios que criar no decurso das aulas práticas. Os computadores das salas de aulas foram programados para apagarem o directório de utilizador (e.g. `/homermt/a1245/`) sempre que são

[6] Acessível no menu *Places*.

[7] https://arca.ua.pt

reiniciados. Só o conteúdo do subdirectório `arca` é salvaguardado. É portanto aí que deve colocar todo o seu trabalho.

Exercício 0.7
Crie, no directório `arca`, um subdirectório chamado `prog2` e, dentro desse, um directório chamado `aula00`.

0.2.2 Manipulação de ficheiros

O Linux (UNIX) dispõe de diversos comandos de manipulação de ficheiros. Eis alguns:

`cat fic` — imprime no dispositivo de saída *standard* (por defeito o ecrã) o conteúdo do ficheiro `fic`.

`rm fic` — remove (apaga) o ficheiro `fic`.

`mv fic1 fic2` — muda o nome do ficheiro `fic1` para `fic2`.

`mv fic dir` — move o ficheiro `fic` para dentro do directório `dir`.

`cp fic1 fic2` — cria uma cópia do ficheiro `fic1` chamada `fic2`.

`cp fic dir` — cria uma cópia do ficheiro `fic` dentro do directório `dir`.

`head fic` — mostra as primeiras linhas do ficheiro de texto `fic`.

`tail fic` — mostra as últimas linhas do ficheiro de texto `fic`.

`more fic` — imprime no dispositivo de saída *standard* (por defeito o ecrã), página a página, o conteúdo do ficheiro `fic`.

`grep padrão fic` — selecciona as linhas do ficheiro texto `fic` que satisfazem o critério de selecção `padrão`.

`wc fic` — conta o número de linhas, palavras e caracteres do ficheiro `fic`.

`sort fic` — ordena as linhas do ficheiro `fic`.

`find dir -name fic` — procura um ficheiro com o nome `fic` a partir do directório `dir`.

Além destes pode ainda considerar outros tais como: `less`, `cut`, `paste`, `tr`, etc. Todos estes comandos podem ser invocados usando argumentos opcionais que configuram o seu modo de funcionamento.

Exercício 0.8
Transfira o ficheiro `Totoloto.java` da página da disciplina para o directório `aula00` que criou no exercício anterior. Imprima o seu conteúdo no ecrã. Experimente outros comandos da lista acima.

0.2.3 Ajuda *On-line*

O Linux dispõe de vários mecanismos de ajuda imediata para a maioria dos seus comandos. Dois dos mais importantes são acedidos através dos comandos `man` e `info`, sendo o primeiro comum em todos os sistemas UNIX e o segundo mais específico do projecto GNU. Muitos comandos aceitam também uma opção `--help` que apresenta um resumo da sua forma de utilização.

Por exemplo, para conhecer as muitas opções de execução do comando `ls` pode executar `man ls`, ou `info ls`, ou `ls --help`.

Nota: Para navegar ao longo das páginas apresentadas pelo `man` ou pelo `info` pode usar as teclas de direcção ↑, ↓ ou as teclas `PageUp`, `PageDown`. Para abandonar as páginas de ajuda e regressar à linha de comando deve premir a tecla `q`. Estes programas têm outras possibilidades de navegação e pesquisa que poderá ficar a conhecer fazendo por exemplo, `man man` ou `info info`.[8]

0.3 Ambiente de Programação em Java

0.3.1 Edição

Comece por editar o programa `Totoloto.java`. Para esse efeito dispõe de vários editores de texto. Aconselhamos, no entanto, a usar o `geany`, o `gedit` (*Text Editor*) ou o `gvim` (*VI editor*), visto possuírem a função de realce da sintaxe da linguagem Java. Na janela de terminal pode usar o editor `vim`, embora este editor tenha uma aprendizagem mais difícil.[9]

0.3.2 Compilação e Execução

O ficheiro que acabou de editar é usualmente designado por programa fonte. O passo seguinte consiste em gerar um programa executável a partir do programa fonte. Isto é feito usando o comando:

```
javac Totoloto.java
```

Verifique se existem erros de compilação e corrija-os. Quando não houver erros, o `javac` gera um programa executável chamado `Totoloto.class`.

Para executar o novo programa, use o comando:

```
java Totoloto
```

[8] Pelo contrário, `busca busca`, `mata mata`, não têm qualquer significado conhecido em UNIX, mas pode sempre tentar!

[9] Também é possível lançar a partir da linha da comandos qualquer outro editor ou programa. Por exemplo, experimente o comando `geany &`.

0.3.3 Documentação

Por vezes, para compreender o significado e a forma de utilização de uma classe ou de uma função em Java temos que consultar a sua documentação. Por exemplo, já deve ter consultado a documentação da classe String[10].

A linguagem Java fornece ferramentas para a extracção automática de documentação associada a programas. Assim, é possível manter a documentação nos próprios ficheiros de código fonte, dentro de comentários com a forma `/** ... */`.

Experimente utilizar o comando `javadoc` sobre o programa fonte fornecido.

```
javadoc -charset utf8 -d doc Totoloto.java
```

Abra o ficheiro `doc/index.html` com um *browser* à sua escolha (por exemplo o `firefox`). Compare a documentação gerada automaticamente com a estrutura e os comentários existentes no programa fonte. Experimente alterar os comentários e volte a gerar a documentação.

[10] https://docs.oracle.com/javase/8/docs/api/java/lang/String.html

Aula Prática 1

Resumo:

- Revisões.
- Utilização da biblioteca *standard* do Java para entrada/saída.
- Programas com argumentos na linha de comando.

Exercício 1.1

Crie uma calculadora simples que leia (do dispositivo de entrada) operações matemáticas como

```
12.3 + 7.2
```

e escreva o resultado respectivo (`19.5` neste exemplo).

As operações serão sempre do género `<número> <operador> <número>`, com as três partes separadas por espaços ou em linhas diferentes. Implemente as quatro operações básicas usando os operadores `+`, `-`, `*` e `/`. Note que o operador é uma palavra (string) que contém apenas um símbolo. Se for introduzido um operador inválido, deve escrever uma mensagem apropriada para o dispositivo de saída de erros (`System.err`).

Exercício 1.2

Estude o programa `p12` e complete-o para determinar a nota final de um aluno de Programação 2 na época normal. O programa pede as notas dos vários momentos de avaliação e terá de calcular e apresentar as notas das componentes, a nota final e indicar se o aluno está aprovado ou não. A nota final deve ser -66 se alguma das componentes for inferior à nota mínima. A fórmula de cálculo está no guião da unidade curricular.

Exercício 1.3

No programa fornecido, complete a função para verificar se um número é primo. Um algoritmo simples consiste em testar se o número é divisível por 2, por 3, etc.[1] Qual o maior divisor que é preciso testar?

[1] Existem algoritmos muito mais eficientes de testar primalidade, mas envolvem conceitos mais avançados.

Exercício 1.4

Na terra do Alberto Alexandre (localmente conhecido por Aubeto Auexande), o dialecto local é semelhante ao português com duas excepções:

- Não dizem os Rs
- Trocam os Ls por Us

Implemente um tradutor de português para o dialecto do Alberto. Por exemplo "lar doce lar" deve ser traduzido para "ua doce ua". A tradução deve ser feita linha a linha, até que surja uma linha vazia.

Exercício 1.5

Escreva um programa que leia uma lista de números e imprima a sua soma e a sua média. O fim da lista é indicado pela leitura do número zero, que não deve ser considerado parte da lista. (Note que se a lista for vazia, a soma será zero, mas a média não pode ser calculada.)

Exercício 1.6

Escreva um programa que implemente o jogo "Adivinha o número!".

Neste jogo, o programa escolhe um número aleatório no intervalo $[0; 100]$[2] e depois o utilizador tenta descobrir o número escolhido. Após cada tentativa, o programa deve indicar se o número escolhido é maior, menor ou igual à tentativa feita. O jogo termina quando o utilizador acertar no número correcto. A pontuação do jogador é o número de tentativas.

Exercício 1.7

Crie um programa que copie um ficheiro de texto para outro[3]. Os nomes dos dois ficheiros envolvidos devem ser dados como argumentos na linha de comandos[4]. Assim a execução do programa com os argumentos `Texto1.txt Texto2.txt` deve criar um ficheiro `Texto2.txt` com um conteúdo igual ao do ficheiro `Texto1.txt`.

Nota: Tente fazer com que o programa tenha alguma robustez, não só detectando a existência do ficheiro original (apresentando uma mensagem de erro quando este não existe), como também a possível existência do ficheiro destino (neste caso fará sentido perguntar ao utilizador se de facto deseja destruir esse ficheiro). Deve também verificar se sobre os ficheiros se podem realizar as operações de leitura e escrita, e se algum deles é um directório (caso em que deve ser apresentada uma mensagem de erro).

[2] `(int)(Math.random()*(100+1))`

[3] Pretende-se que seja uma versão simplificada do comando `UNIX cp`.

[4] No programa Java esses valores farão parte do *array* de *strings* que é passado como argumento da função `main`.

Aula Prática 2

Resumo:
- Classes e objectos em Java.

Exercício 2.1
Analise o código dos ficheiros `Complex.java` e `TestComplex.java`.

a. Compile e execute o programa em modo de linha de comando:

```
$ javac TestComplex.java
$ java TestComplex
```

b. Execute o programa pré-compilado `TestComplex.jar`, com e sem argumentos, e confira o seu funcionamento.

```
$ java -jar TestComplex.jar
$ java -jar TestComplex.jar 1.5 -1.5
$ java -jar TestComplex.jar 3
```

Altere o programa `TestComplex.java` para que tenha esse mesmo comportamento.

Exercício 2.2
A classe `Contacto` representa um contacto telefónico. Já inclui campos privados para guardar o nome e o telefone de um contacto. Acrescente à classe uma forma de guardar um endereço de email e os métodos necessários para que o programa `TestContacto.java` funcione corretamente.

Exercício 2.3
Crie uma cópia da classe `Contacto` num pacote `pt.ua.prog2` e altere-lhe o método `nome` para devolver o nome em maiúsculas. Copie o programa anterior para `TestContacto2.java` e altere-o de forma a funcionar com a nova classe. Note que o ficheiro do programa não deve pertencer ao mesmo pacote da classe `Contacto`.

Exercício 2.4

A classe `Data` permite criar objetos que representam datas. Esta classe define três campos inteiros privados para registar o dia, o mês e o ano de uma data e já tem um construtor, um método que devolve a data no formato normalizado `"AAAA-MM-DD"` e um método de classe que permite verificar se um ano é bissexto, mas falta completar vários outros métodos. O ficheiro `TestData1.java` tem um programa que usa objetos de tipo `Data` e testa vários métodos.

Comece por tentar compilar este programa. Terá de completar vários métodos da classe `Data` com instruções `return` que devolvem valores do tipo adequado, ainda que incorretos. Quando conseguir compilar, execute o programa e analise o resultado. A seguir, complete progressivamente os métodos que faltam.

- Acrescente à classe `Data` os métodos `dia`, `mes` e `ano` e descomente algumas linhas do programa para os testar. Esses métodos devem ser `static` ou não? Devem ser declarados `public` ou `private`?

- Complete e teste o método `diasDoMes`, que deve devolver o número de dias de um dado mês (de dado ano). Faça uso do método `bissexto` e da tabela `diasMesComum`, que já estão definidos. Repare que essa tabela é um atributo de classe (`static`): não ocupa memória no contexto de qualquer objeto que venha a ser criado.

- Complete o método `mesExtenso` e teste-o no programa. Sugestão: pode criar uma tabela de nomes e indexá-la com o mês. Essa tabela deve ser um atributo de classe ou de instância?

- Complete o método `extenso` para devolver uma string com a data por extenso, como em "25 de Abril de 1974".

- Complete o método estático `dataValida`, que deve verificar se um terno de inteiros (dia, mês, ano) forma uma data válida, isto é, se o dia e o mês estão dentro dos limites esperados.

- Complete o construtor `Data(dia, mes, ano)`, que inicializa o objecto com essa data.

- Complete o método `seguinte` que deve avançar a data para o dia seguinte.

Exercício 2.5

Usando a classe `Data`, faça um programa `DatasPassadas.java` que escreva, por extenso, todas as datas desde o Natal do ano passado até à data atual.

Aula Prática 3

Resumo:
- Programação modular.

Exercício 3.1

Recupere a classe `Data` desenvolvida no exercício 2.4 e acrescente-lhe:

- Um construtor que inicie a data a partir de uma string no formato ISO (como `"2018-02-28"`). Se a string for inválida, o construtor deve indicá-lo e terminar o programa (com `exit`).[1]
- Um método `compareTo` para comparar datas. O resultado de `x.compareTo(y)` deve ser um inteiro nulo, positivo ou negativo consoante `x` seja igual, maior ou menor que `y`, respetivamente.

Utilize o programa `TestData2` para testar esta funcionalidade.

Exercício 3.2

Experimente executar o comando: `java -jar SortDates1.jar dates1.txt`. Esse programa lista, por ordem cronológica, um conjunto de datas lidas do ficheiro dado no argumento.

`SortDates1.java` é uma implementação incompleta e com vários erros desse programa. Comece por corrigir os erros sintáticos até conseguir executá-lo. Depois analise os erros de execução, identifique e corrija as suas causas. Finalmente, descomente a chamada à função de ordenação e teste de novo para detetar e corrigir os erros restantes.

Exercício 3.3

Crie uma classe `Tarefa` que permita representar um texto associado a um intervalo entre duas datas. Utilize o programa `TestTarefa1.java` para inferir qual tem de ser a *interface* da classe, isto é, o conjunto dos seu membros públicos. Teste usando o comando `java TestTarefa1 tasks1.txt`.

[1] Veremos formas melhores de lidar com falhas noutra aula.

Exercício 3.4

Utilizando as classes desenvolvidas nos exercícios anteriores construa uma nova classe `Agenda` onde seja possível registar até mil tarefas. Nos objetos deste tipo deve ser possível realizar as seguintes operações:

Método `novaTarefa`: Acrescenta uma nova tarefa à agenda. As tarefas devem ser mantidas por ordem crescente das suas datas iniciais.

Método `escreve`: Mostra o conteúdo completo da agenda.

Método `filtra`: Extrai para uma nova agenda as tarefas que intersectam um certo intervalo de datas. Sugestão: acrescente à classe `Tarefa` um método `t1.intersecta(t2)` que permita verificar se duas tarefas se intersectam.

Teste a classe com o programa `TestAgenda.java`. Compare o resultado com o do comando `java -jar TestAgenda.jar`.

Exercício 3.5

Uma empresa de construção precisa de gerir a informação, bem como extrair diversas propriedades, das habitações que constrói.

Desenvolva um conjunto de classes para esta aplicação. Fornece-se em anexo o programa `TestHouses`, que permite testar as funcionalidades pretendidas, bem como a classe `Point` usada para representar pontos num espaço cartesiano.

a. Desenvolva uma classe `Room` para representar as divisões das habitações. Considera-se que cada divisão tem uma forma rectangular, alinhada com os eixos de um determinado sistema de coordenadas. Esta classe deverá ter os seguintes métodos públicos:

   - Construtor com três argumentos, nomeadamente o tipo da divisão (uma cadeia de caracteres), e as coordenadas dos cantos inferior esquerdo e superior direito;
   - `roomType()` - devolve o tipo da divisão;
   - `bottomLeft()` - devolve o canto inferior esquerdo;
   - `topRight()` - devolve o canto superior direito;
   - `geomCenter()` - devolve o centro geométrico da divisão;
   - `area()` - devolve a área da divisão.

b. Desenvolva uma classe `House` para representar as habitações, com os métodos:

   - `House(String)` - Construtor que recebe como argumento e regista o tipo da habitação (uma cadeia de caracteres que poderá ser `"house"` ou `"apartment"`); além disso, este construtor deve reservar memória para 8 divisões e deve também registar que, caso venha a ser preciso armazenar informação sobre mais divisões, a memória das divisões será expandida em blocos de 4 divisões adicionais (inicialmente 8, depois sucessivamente 12, 16, etc., conforme as necessidades);

- `House(String, int, int)` - Construtor que recebe como argumentos o tipo da habitação, o número de divisões para as quais se vai inicialmente reservar memória, bem como o número de divisões adicionais a reservar sempre que a memória esteja cheia;
- `addRoom(Room)` - adiciona uma nova divisão à habitação;
- `size()` - devolve o número de divisões da casa;
- `maxSize()` - devolve o número máximo de divisões que é possível armazenar num dado momento;
- `room(int)` - dado um índice de uma divisão (um inteiro entre `0` e `size()-1`), devolve a divisão correspondente;
- `area()` - devolve a área total da habitação, dada pela soma das áreas das divisões;
- `getRoomTypeCounts()` - devolve os tipos de divisões existentes com o número de divisões de cada tipo, na forma de um vector (array) de elementos da seguinte classe:

```
public class RoomTypeCount {
   String roomType;
   int count;
}
```

  Nota: este vector não deverá estar sobre-dimensionado.
- `averageRoomDistance()` - devolve a distância média entre as divisões da casa, tomando como referência os respectivos centros geométricos;

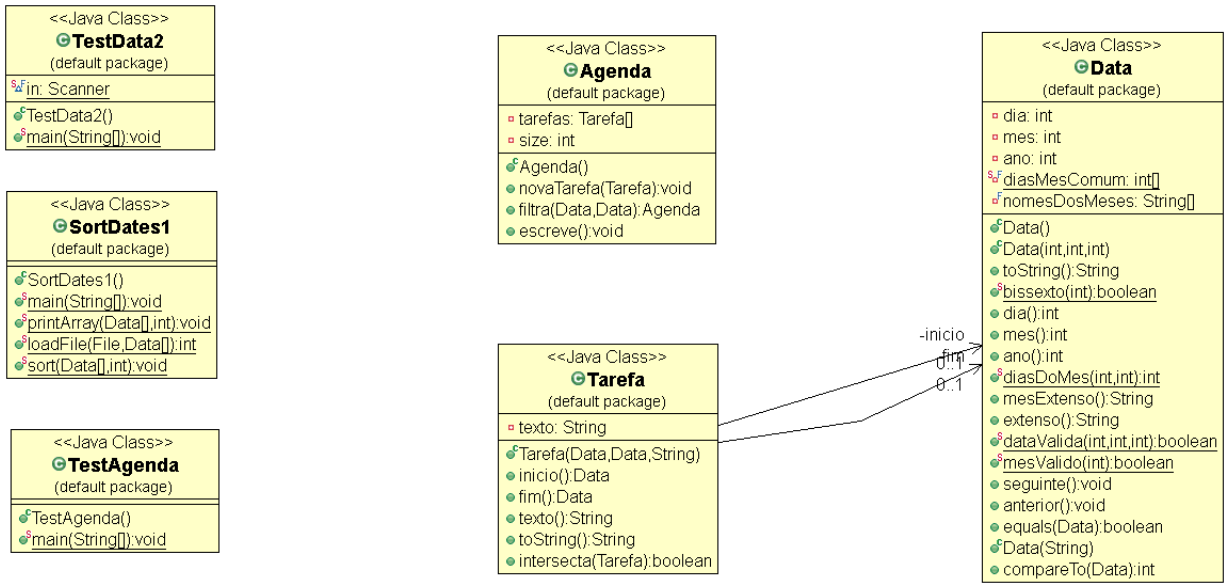


Figura 3.1: Diagrama das classes dos exercícios 3.1, 3.2, 3.3 e 3.4.

Aula Prática 4

Resumo:
- Programação por Contrato.

Exercício 4.1

Modifique a classe `Data` da aula anterior por forma a garantir que os seus objectos correspondem sempre a datas válidas. Para isso, acrescente instruções `assert` para testar pré-condições adequadas aos métodos `Data(d, m, a)` e `diasDoMes(m, a)` e para testar o invariante desejado no final do método `seguinte()`. Recorra ao método estático `dataValida(d, m, a)` que já criou. Por conveniência, pode também criar e usar um método de objeto `valida()` que verifique a validade do próprio objeto.

Teste esta classe correndo o programa `TestData3` com as asserções ativadas.

```
java -ea TestData3
```

Nesse programa, descomente a linha

```
Data.diasDoMes(0, 1900); // should fail!
```

compile e volte a testar. O programa deveria falhar com um erro semelhante ao seguinte:

```
java.lang.AssertionError: Mês inválido;
```

indicando que o teste à pré-condição falhou.

Volte a comentar a linha, descomente a próxima e volte a testar. Para cada uma dessas linhas, o programa deverá detetar uma falha de asserção. Devem ser asserções no início dos métodos, correspondentes a pré-condições, o que indica que a responsabilidade do erro é do programa cliente.

Introduza erros propositados na implementação dos métodos `diasDoMes(...)` e de `seguinte()` e volte a testar. As asserções que falham agora devem estar no fim dos métodos, correspondendo a pós-condições, o que indica que a responsabilidade do erro é do próprio método.

Exercício 4.2

Modifique também a classe `Tarefa`, acrescentando asserções que garantam as seguintes propriedades em todos os objectos deste tipo:

- a data de fim da tarefa não pode ser anterior à data de início;
- o texto de uma tarefa não pode ser vazio.

Teste esta classe correndo o programa `TestTarefa2` sempre com as asserções ativadas.

```
java -ea TestTarefa2
```

Exercício 4.3

A classe `number.Fraction` é inspirada num exemplo desenvolvido numa aula teórico-prática (ver `aula03-print.pdf`). A classe já tem um construtor e métodos para adicionar e multiplicar frações. Note que esta classe garante que os seus objetos nunca têm denominador nulo. O construtor já tem asserções que definem e testam o seu contrato.

- Altere o invariante da class `Fraction` para garantir que o denominador armazenado é sempre positivo. Este invariante mais forte implica alterar vários métodos. Por exemplo, `new Fraction(3, -5)` deve continuar a funcionar, mas o novo objeto deverá ficar com `num = -3` e `den = 5`. Por outro lado, a implementação de `toString()` pode ser simplificada. Precisa de alterar o método `parseFraction`? E os restantes?
- Complete o método `equals` que serve para verificar se duas frações são iguais.
- Complete os métodos para dividir e subtrair. Qual a pré-condição do método `divide`? Inclua uma asserção para o testar. Inclua também asserções para testar a pós-condição de cada operação. Note que o quociente $q = a/b$ tem de satisfazer $q \cdot b = a$. Analogamente, a diferença $d = a - b$ tem de satisfazer $d + b = a$. Garanta ainda que o resultado satisfaz o invariante das frações.

Use o programa `FractionCalcB` para experimentar a classe. Calcule o resultado das expressões abaixo e outras que se lembre.

```
3/4 + 4/3
25/12 - -3/-4
64/48 == 4/3
3/2 x 1/3
2/-3 : -1/3
```

Exercício 4.4

As asserções, além de serem uma excelente ferramenta para verificar os contratos das classes, também podem ser usados para especificar testes *externos*. O programa `TestFractions` é um exemplo disso: faz uma série de testes à classe `Fraction`. Experimente compilá-lo e executá-lo. Corrija quaisquer erros que detete e volte a tentar.

Este programa tem vários outros testes em comentário. Deverá descomentar os testes progressivamente à medida que for implementando as funcionalidades abaixo.

- Complete o método de comparação na classe `Fraction`. Pretende-se que `a.compareTo(b)` devolva um inteiro negativo se $a < b$, positivo se $a > b$ e nulo se $a = b$. Descomente os testes a essa função no `TestFractions` e volte a correr.

- A biblioteca *standard* de Java inclui a função `Arrays.sort` que permite ordenar arrays genéricos com elementos de qualquer tipo de dados que tenha uma relação de ordem definida. Por exemplo, é possível ordenar um array de inteiros, ou um array de strings. Descomente os testes para ver se consegue ordenar um array de frações. Funcionou?

- Para o `Arrays.sort` funcionar com arrays de frações, além de ter o método `compareTo`, também é preciso que a classe `Fraction` declare que implementa essa operação com o funcionamento e propriedades prescritas na interface `Comparable`.

  ```
  public class Fraction implements Comparable<Fraction>
  ```

  Faça essa alteração e volte a testar o programa.

- Acrescente dois atributos `Fraction.ZERO` e `Fraction.ONE` que correspondam às constantes 0 e 1, em forma de fração. Que qualificadores deve usar na sua declaração? `public`? `private`? `static`? `final`? Corra os testes e confirme que não consegue compilar o programa se descomentar a instrução `Fraction.ZERO = two`.

- Verifique que a adição de frações satisfaz a propriedade comutativa. Acrescente asserções para testar a propriedade associativa da adição e as propriedades comutativa, associativa e distributiva da multiplicação.

Exercício 4.5

Pretende-se implementar um programa para o jogo "Advinha o número". Neste jogo, o programa escolhe aleatoriamente um número secreto num determinado intervalo [min, max] e o objectivo é adivinhar esse número com o mínimo de tentativas. Por cada tentativa feita, o jogo deve indicar se acertou (e terminar), ou se é menor ou maior do que o número secreto.

Implemente o comportamento essencial do jogo na classe `GuessGame` respeitando a seguinte interface pública. Sempre que possível, use instruções `assert` para tornar explícitos os contratos dos métodos, particularmente as suas pré-condições.

- Um construtor com dois argumentos (`min` e `max`) que inicializa o jogo com um número aleatório no intervalo `[min,max]` (que não poderá ser vazio). Pode utilizar o método `Math.random()` ou a classe `Random` para gerar o número aleatório.

- Dois métodos inteiros – `min()` e `max()` – que indiquem os limites do intervalo definido para o objecto.

- Um método booleano – `validAttempt` – que indique se um número inteiro está dentro do intervalo definido.

- Um método booleano – `finished()` – que indique se o segredo já foi descoberto.
- Um método – `play` – que registe uma nova tentativa para adivinhar o número secreto. Este método só deverá aceitar tentativas que estejam dentro do intervalo definido e só enquanto o jogo não tiver terminado ou seja, enquanto o segredo não tiver sido descoberto.
- Dois métodos booleanos – `attemptIsHigher()` e `attemptIsLower()` – que indiquem se a última tentativa foi, respectivamente, acima ou abaixo do número secreto.
- Um método inteiro – `numAttempts()` – que indique o número de tentativas (jogadas) já realizadas.

Note que terá de definir um conjunto de campos (que devem ser privados) adequados para a implementação da classe. Ou seja, necessita de campos que permitam saber os valores do intervalo, o número secreto, o número de tentativas, ou determinar se a última tentativa é igual, maior ou menor que o segredo.

Para facilitar a depuração da classe, é fornecido um programa (`main`) embutido na própria classe, que utiliza a instrução `assert` para fazer alguns testes ao funcionamento da classe. Assim, para testar a classe, deve compilá-la e executá-la com as asserções activadas.

```
javac GuessGame.java
java -ea GuessGame
```

Exercício 4.6

Complete o programa `PlayGuessGame` que usa a classe `GuessGame` para implementar o jogo "Advinha o número". Sem argumentos, o programa escolhe um número secreto no intervalo $[0, 20]$.

Altere o programa para aceitar dois argumentos que indiquem os limites do intervalo. A interação com o utilizador deve ser análoga à da solução fornecida, que pode testar com o comando seguinte.

```
java -ea -jar PlayGuessGame.jar
```

Exercício 4.7

Altere a implementação da class Fraction para garantir que as frações são sempre armazenadas na forma reduzida (ou irredutível), com o menor denominador possível (e positivo). Dará jeito criar uma função para achar o máximo divisor comum de dois inteiros. Altere a definição do invariante interno e modifique os restantes métodos adequadamente.

Aula Prática 5

Resumo:

- Robustez e Excepções.

Exercício 5.1

O programa `TestInput` utiliza funções para leitura interativa de valores reais. Experimente-o. Se o utilizador introduzir uma linha mal formatada, o programa termina com um `NumberFormatException`.

Altere as funções na classe `util.Input` para as tornar robustas. Mesmo que o utilizador introduza dados mal formatados, a função deverá avisar e voltar a pedir. Pode observar o comportamento pretendido com o programa `TestInput.jar`.

```
Real value X? pi
Invalid input format!
Real value X? 1 2
Invalid input format!
Real value X? 1.2
1.2
Nota? -1
Value must be in range [0.000000, 20.000000]
Nota? vinte
Invalid input format!
```

Acrescente a pré-condição adequada à função `readDouble(String, double, double)`. Descomente a última instrução no programa para testar que a asserção falha.

Exercício 5.2

Altere o programa `PlayGuessGame` da aula anterior por forma a torná-lo totalmente robusto. Em particular, o programa deve garantir a sanidade dos argumentos e dos valores introduzidos pelo utilizador.

Exercício 5.3

Crie um programa `CopyFile` que copie um ficheiro de texto. Os nomes dos dois ficheiros envolvidos devem ser dados como argumentos na linha de comandos.[1] Por exemplo, `java -ea CopyFile Texto1.txt Texto2.txt` deve criar um ficheiro `Texto2.txt` com um

[1] Pretende-se que seja uma versão simplificada do comando `UNIX cp`.

conteúdo igual ao do ficheiro `Texto1.txt`. Se `Texto2.txt` já existir, deve perguntar ao utilizador se deseja reescrever o seu conteúdo.

O programa deve ser robusto. Deve detetar falhas e apresentar mensagens de erro apropriadas. Várias condições têm de estar reunidas para que o programa funcione. O ficheiro original tem de existir, tem de ser um ficheiro normal e o utilizador tem de ter permissão para o ler. Se o ficheiro de destino existir, tem de ter permissão de escrita. Se não existir, então tem de ter permissão de escrita no diretório pai. Mesmo assim, podem ocorrer falhas imprevisíveis, como erros no disco ou de comunicação, que devem ser reportadas.

Exercício 5.4

Crie um programa `ListDir` que faça uma listagem de um diretório com alguma meta-informação dos seus ficheiros. O nome do directório é passado como argumento do programa. Se for chamado sem argumentos, o programa deve listar o directório actual (`.`). Para cada ficheiro, deve ser indicado se é um ficheiro normal ou um directório (`F` ou `D`) e se tem permissão de leitura ou não (`R` ou `-`) e se tem permissão de escrita ou não (`W` ou `-`), como se vê no exemplo abaixo.

```
DRW  ./dir1
DR-  ./dir2
F-W  ./file1
FRW  ./file2
```

Neste programa deve usar os métodos adequados da classe `File`, em particular: `listFiles()`, `getPath()`, `isDirectory()`, `canRead()`, `canWrite()` e outros. Recorra à documentação dessa classe para perceber a sua utilização.

Para testar o programa, pode criar pastas e ficheiros e alterar as suas permissões com os comandos UNIX: `mkdir`, `touch` e `chmod`.

Exercício 5.5

Construa um programa `CutColumn` que, dado um ficheiro de texto, escreva somente a n-ésima palavra de cada linha do texto. Quando a linha não tem n palavras, deve escrever uma linha em branco. Considere que as palavras são separadas por um ou mais espaços ou caracteres de tabulação. (Pode usar `line.split("[ \t]+")` para dividir cada linha.)

Quer o nome do ficheiro quer o número da coluna a extrair devem ser dados como argumentos do programa.

Por exemplo, no caso de termos o seguinte texto de entrada:

```
1 2 3 4 5 6
    boa     tarde
um dois tres quatro cinco seis
```

A saída do programa, se for escolhida a coluna número `4`, deverá ser:

```
4

quatro
```

Aula Prática 6

Resumo:

- Funções recursivas.

Exercício 6.1

A função de Fibonacci[1] de um número inteiro (não negativo) n pode ser definida por:

$$F(n) = \begin{cases} 0 & \text{se } n = 0 \\ 1 & \text{se } n = 1 \\ F(n-2) + F(n-1) & \text{se } n > 1 \end{cases} . \tag{6.1}$$

a. Complete a função `fibonacci` e teste-a no programa `Fibonacci`, que escreve os números $F(n)$ para os valores de n dados nos argumentos. Experimente calcular $F(5), F(10), F(20), F(40)$, por exemplo.

b. A implementação recursiva direta desta relação de recorrência é muito ineficiente porque invoca a função repetidamente com os mesmos argumentos. Uma técnica geral que permite colmatar este problema é a chamada *memoização*. Consiste em memorizar os resultados de invocações anteriores numa tabela e devolver o valor memorizado caso volte a ser pedido. Implemente uma versão memoizada da função e confirme o seu desempenho.

Exercício 6.2

Construa uma função recursiva que imprima um array de strings, uma string por linha. O algoritmo é simples: para imprimir N linhas, imprimimos as $N-1$ primeiras e depois a última. Use-a no programa `PrintArgs` para imprimir os argumentos da linha de comando.

Repare que, além do array, é conveniente a função ter um parâmetro extra que permita indicar que parte do array deve imprimir. Este "truque" permite usar o mesmo array em todas as invocações, e variar apenas o segundo parâmetro. Assim não é preciso criar um novo array em cada invocação.

[1] Matemático italiano dos Séculos XII-XIII, responsável, entre outros feitos, pela introdução da chamada numeração árabe na Europa.

Exercício 6.3

Copie o programa anterior para `ReverseArgs.java` e altere a função recursiva por forma a que agora escreva as strings por ordem inversa.

Exercício 6.4

Construa uma função recursiva – `reverseString` – que inverta uma qualquer `String` passada como argumento. Para testar a função, implemente um programa que a aplique a cada um dos argumentos.

Exercício 6.5

Escreva um programa que mostre o conteúdo de um directório e de todos os seus subdirectórios recursivamente.

Por exemplo, se for executado o comando `java -ea ListRec ../aula02` , o resultado deverá assemelhar-se ao seguinte.

```
../aula02
../aula02/pt
../aula02/pt/ua
../aula02/pt/ua/prog2
../aula02/pt/ua/prog2/Contacto.java
../aula02/Complex.java
../aula02/Contacto.java
../aula02/TestContacto.java
(...)
```

Nota: Sugere-se a utilização das funções `listFiles` e `getPath` da classe `File` para obter, respectivamente, a lista de ficheiros existentes num directório e a localização de cada ficheiro.

Exercício 6.6

O programa `Ngrams` tem uma função `all3grams` que permite gerar todos os arranjos possíveis de três símbolos escolhidos de um dado alfabeto. Experimente compilar e correr `java -ea Ngrams ab`.

Crie uma função `allNgrams` que permita generalizar o programa para gerar todos os n-gramas (sequências de n símbolos) possíveis de um dado alfabeto. Um algoritmo recursivo para obter cada um dos n-gramas consiste em obter a lista de todos os $(n-1)$-gramas e a cada um deles acrescentar cada um dos símbolos do alfabeto. Qual será o caso base? E que resultado lhe corresponde?

Exercício 6.7

a. Escreva um programa que encontre numa árvore de directórios todos os ficheiros com um determinado nome.

   Por exemplo, se executar o comando `java -ea FindFile Contacto.java ..` , a saída deverá ser:

```
../aula02/Contacto.java
../aula02/pt/ua/prog2/Contacto.java
```

   Nota: Pode usar a função `getName` da classe `File` para obter o nome do ficheiro.

b. Generalize o programa anterior por forma a encontrar ficheiros que contenham um determinado texto no seu nome.

   Por exemplo, se for executado o comando `java -ea FindFile acto.jav ..` , o resultado deverá incluir os ficheiros anteriores, mas também outro(s).

   Nota: Sugere-se a utilização da função `indexOf` da classe `String` para verificar a ocorrência do texto no nome.

Exercício 6.8

O máximo divisor comum (mdc) de dois números inteiros não negativos a e b pode ser calculado usando o algoritmo de Euclides que se pode expressar pela seguinte definição recursiva:

$$mdc(a,b) = \begin{cases} a & \text{se } b = 0 \\ mdc(b, a \bmod b) & \text{se } b \neq 0 \end{cases} . \tag{6.2}$$

onde o operador mod corresponde à operação *resto da divisão inteira* implementada em `Java` pelo operador `%`.

Escreva uma função que implemente este algoritmo e teste-a num programa simples.

Exercício 6.9

Um cliente de um banco pede um empréstimo de M Euros com uma taxa de juro de $T\%$ ao mês e uma prestação de P Euros no fim de cada mês.

a. Determine a relação de recorrência que descreve o montante em dívida D_n ao fim de n meses.

b. Implemente, com o método iterativo, uma função para determinar D_n.

c. Implemente, com o método recursivo, uma função para determinar D_n.

Por exemplo, com um empréstimo de $M = 1000$ Eur, a uma taxa de $T = 1\%$ e prestação mensal de $P = 20$ Eur, a dívida ao fim de 2 meses pode calcular-se com o comando `java -ea -jar Loan.jar 2 1000 1 20`.

Exercício 6.10

Construa uma função recursiva que determine a chamada distância de Levenshtein entre duas palavras. Esta medida é o menor número de inserções, remoções ou substituições de um carácter necessárias para converter uma palavra na outra. Por exemplo, a distância entre as palavras `"lista"` e `"lata"` é 2, porque se consegue converter `"lista"` em `"lata"` com, no mínimo dos mínimos, duas operações (uma remoção e uma substituição).

Note que qualquer palavra não vazia P pode ser decomposta no seu primeiro carácter C e o resto da palavra S, ou seja: $P = C + S$ (se $\text{length}(P) > 0$).[2] Dessa constatação surge naturalmente a seguinte relação de recorrência para a distância de Levenshtein:

$$d(P_1, P_2) = \begin{cases} \text{length}(P_1) & \text{se length}(P_2) = 0 \\ \text{length}(P_2) & \text{se length}(P_1) = 0 \\ d(S_1, S_2) & \text{se } C_1 = C_2 \quad (*) \\ 1 + min(d(S_1, P_2), d(P_1, S_2), d(S_1, S_2)) & \text{se } C_1 \neq C_2 \quad (*) \end{cases} . \tag{6.3}$$

$(*)$ Nestes casos, obviamente que nem P_1 nem P_2 podem ser vazias.

[2]Em Java, pode determinar `C = P.charAt(0)` e `S = P.substring(1)`.

Aula Prática 7

Resumo:
- Funções e estruturas de dados recursivas.

Exercício 7.1

Escreva um programa `FilterLines` que processe um ficheiro de texto e imprima primeiro as linhas com menos de 20 carateres, depois as linhas com 20 a 40 carateres e finalmente as linhas mais longas. Para cumprir o objectivo, o programa poderá ler e armazenar as linhas de texto, divididas em três conjuntos distintos, imprimindo-os no fim. Como não se conhece a priori o número de linhas existentes no ficheiro, sugere-se a utilização de três listas para guardar as frases separadamente. Use para esse efeito a classe `LinkedList` do pacote `p2utils` em anexo.

Exercício 7.2

A classe `p2utils.LinkedList` define o tipo *lista ligada* com os métodos essenciais para inserção e remoção de elementos. Analise os métodos `print()` e `contains()`. Repare que ambos recorrem a funções auxiliares privadas e recursivas, que usam um parâmetro extra de tipo `Node`.

Acrescente à classe os seguintes métodos, implementado-os também de forma recursiva.

- `clone()` - devolve uma nova lista com a mesma sequência de elementos.
- `reverse()` - devolve uma nova lista com os mesmos elementos por ordem inversa.
- `get(pos)` - devolve o elemento na posição `pos` da lista, em que `pos` varia entre `0` e `size()-1`.
- `concatenate(lst)` - devolve uma nova lista com os elementos da lista (em que o método é chamado) seguidos dos elementos da lista dada no argumento.
- `remove(e)` - remove da lista a primeira ocorrência do elemento dado no argumento.

Teste a classe com o programa `TestList1`.

Exercício 7.3

No exercício 6.5 fez uma função para listar recursivamente o conteúdo de um diretório.

No programa `ListRec2` pretende-se fazer o mesmo, mas agora recorrendo a uma função `recListFiles` que, em vez de imprimir a lista, deve devolver o resultado numa lista ligada de elementos de tipo `File`.

Exercício 7.4

Considere agora um problema parecido com o do exercício 7.1. Neste caso, o programa deve imprimir primeiro as linhas com menos de 20 carateres, por ordem inversa daquela em que estavam no ficheiro, seguidas das linhas mais longas, pela ordem original. Note que pode resolver o problema com apenas uma lista, se inserir elementos pelas duas extremidades.

Exercício 7.5

O programa `TestDArray` demonstra a utilização de um tipo de dados que funciona como um array, mas sem limitação de capacidade. Complete a classe `p2utils.DynamicArray`, que implementa esse tipo de dados. Considere que:

- A classe é genérica. O parâmetro de tipo é usado para indicar o tipo dos elementos a armazenar no array dinâmico.
- O campo `array` servirá para guardar os elementos introduzidos. O construtor deve criar um array com uma dimensão inicial nula.
- A operação `a.set(n, v)` serve para armazenar o valor `v` na posição `n` do array dinâmico `a` (análogo a `a[n] = v`). O indíce `n` não pode ser negativo, mas pode ser maior ou igual a `a.length()`. Quando isso acontece, o array tem de ser redimensionado para ter um tamanho maior que o índice. O tamanho deverá aumentar sempre para um múltiplo da constante `BLOCK`.
- A operação `v = a.get(n)` deve devolver o valor que tiver sido armazenado na posição `n` de `a` (análogo a `v = a[n]`, se `a` fosse um array nativo). Se nenhum valor tiver sido armazenado nessa posição, deve devolver `null`.
- A operação `v = a.get(n, d)` só difere da anterior por devolver o valor `d` em vez de `null`.

Experimente compilar o programa de teste. Para o executar, tem de fornecer um texto pelo dispositivo de entrada, terminando com a marca de fim-de-ficheiro (`Ctrl+D` em Unix), ou usar redirecionamento de *input*, por exemplo fazendo:
`java -ea TestDArray < /etc/dictionaries-common/words`.

Aula Prática 8

Resumo:

- Listas e vectores ordenados.
- Recursão e iteração.

Exercício 8.1

A função seguinte calcula a soma de um subarray de números reais:

```
// sum of subarray [start,end[ of arr:
static double sum(double[] arr, int start, int end) {
  assert 0 <= start && start <= end && end <= arr.length;
  double res = 0;
  for(int i = start; i < end; i++)
    res += arr[i];
  return res;
}
```

Implemente uma versão recursiva – `sumRec` – desta função. Para a testar, complete o programa `SumArgs` para fazer a soma de todos os seus argumentos.

Exercício 8.2

No material da aula, encontra a classe `SortedListInt`, idêntica à desenvolvida na aula teórica. Usando esta classe, faça um programa `SortInts` que leia números inteiros de um ou mais ficheiros e, no final, imprima todos os números lidos por ordem. Os nomes de ficheiros serão passados como argumentos ao programa. O ficheiro poderá conter outras palavras: as que não representem inteiros devem ser ignoradas.

Exercício 8.3

Faça as adaptações necessárias para transformar a classe `SortedListInt` numa classe genérica `SortedList` que deverá colocar no pacote `p2utils`. Na declaração da classe, deve especificar que os elementos do tipo `E` são comparáveis:

```
public class SortedList<E extends Comparable<E>> {...}
```

Na implementação, terá que usar a função `compareTo()` para comparar elementos.

O programa `TestSortedList` demonstra a utilização da nova classe. Experimente correr `java -ea TestSortedList 9 3.14 24 10`. Altere o programa para aceitar também

argumentos não numéricos e mostrá-los ordenados lexicograficamente numa lista separada. Experimente `java -ea TestSortedList 9 dois pi 24 dez 3`. (Compare com o programa `.jar` fornecido.)

Exercício 8.4

Acrescente progressivamente os métodos necessários à classe `SortedList` para conseguir compilar e executar o programa `TestSortedList2`.

- O método `lst.contains(e)` verifica se um elemento `e` existe na lista `lst`. Como a lista está ordenada, pode implementar uma solução mais eficiente do que a desenvolvida para a classe `LinkedList` da aula anterior. Comece por criar e testar uma solução iterativa. Em seguida, desenvolva e teste uma solução recursiva.
- O método `lst.toString()` devolve uma cadeia de caracteres que representa o conteúdo da lista, num formato semelhante a `"[1, 2, 3]"`. Desenvolva uma solução iterativa.
- O método `lst1.merge(lst2)` devolve uma nova lista ordenada contendo os elementos das listas `lst1` e `lst2`, mas sem as alterar. Desenvolva uma solução recursiva.

Exercício 8.5

A classe `SortedList` genérica permite, por exemplo, manter uma lista ordenada dos aniversários dos seus familiares e amigos. Para isso:

- Desenvolva uma classe `Pessoa` com as seguintes funcionalidades: construtor tendo como parâmetros o dia de nascimento (uma `Data`) e o nome da pessoa (`String`); métodos de acesso aos campos data de nascimento e nome; método `toString()`, que devolve uma representação da pessoa em cadeia de caracteres.
- Adicione um método de comparação que deve comparar apenas o dia e mês de nascimento. `p1.compareTo(p2)` deve devolver um inteiro negativo, positivo ou zero, consoante a pessoa `p1` faça anos antes, depois ou no mesmo dia que `p2`, respetivamente. Finalmente, para que `Pessoa` seja reconhecido como um tipo comparável, e portanto aceitável como argumento numa `SortedList`, a declaração da classe terá que ser: `public class Pessoa implements Comparable<Pessoa> { ... }`
- Complete o programa `Birthdays` que obtém os dados de algumas pessoas pelos argumentos e os apresenta ordenados por data de aniversário. Utilize uma lista ordenada de pessoas para resolver o problema.

Exercício 8.6

Acrescente uma nova classe `SortedArray` ao pacote `p2utils`. Esta classe deverá ter funcionalidade semelhante à da `SortedList`, mas deverá ser implementada com base num vector de dimensão fixa. A dimensão será dada como argumento do construtor. Como o vector tem capacidade limitada, a classe deverá ter um método `isFull()`, que devolve `true` se o vector estiver cheio e `false` caso contrário. Use o programa `TestSortedArray2` para testar.

Aula Prática 9

Resumo:

- Complexidade algorítmica;
- Algoritmos de ordenação;
- Recursividade.

Exercício 9.1

Na classe `p2utils.Sorting`, implemente a função `mergeSort` com o algoritmo de ordenação por fusão para vetores de números inteiros. Utilize o programa `TestMergeSort` para testar a função. Podemos utilizar este programa de inúmeras formas. Por exemplo:

```
$ # input as output of command echo (piping):
$ echo 4 3 2 6 1 2 3 5 7 8 9 5 | java -ea TestMergeSort

$ # input as output of command cat (piping):
$ cat numbers.txt | java -ea TestMergeSort

$ # redirecting input from file (assuming file numbers.txt exists):
$ java -ea TestMergeSort < numbers.txt

$ # no piping nor redirecting (Ctrl-d to terminate the input):
$ java -ea TestMergeSort
4 5 2 3 2
2 3
<Ctrl-d>
```

Exercício 9.2

Na classe `p2utils.Sorting`, implemente a função `insertionSort` com o algoritmo de ordenação por inserção. Utilize o programa `TestInsertionSort` para testar a função.

Exercício 9.3

O programa `TestSorting` mede os tempos necessários para ordenar arrays de N inteiros usando diversos algoritmos. Experimente-o e analise-o para perceber o seu funcionamento.

a. A classe `p2utils.Sorting` inclui dois campos `static` para contabilizar o número de operações elementares requeridas pelos algorimtos de ordenação:

   `comparisonCount` para contar comparações entre elementos do array;

`assignmentCount` para contar atribuições de valor a elementos.

Acrescente instruções para incrementar esses contadores nos locais apropriados dos vários algoritmos. Teste o programa com $N = 100$ e $N = 1000$ e observe as contagens de operações nos vários casos.

b. Altere o programa para aceitar vários valores de N na linha de comando e produzir uma tabela com todos os resultados das diversas medições para todas essas dimensões, todos os algoritmos de ordenação, e para os três casos de teste (arrays aleatórios, ordenados crescentemente e ordenados decrescentemente). Para cada dimensão, todos os algoritmos devem ser aplicados exatamente aos mesmos dados.

Exercício 9.4

Em Java, além de classes genéricas, também é possível criar métodos genéricos. No fim da classe `p2utils.Sorting` está declarado um método genérico para ordenar arrays com elementos de tipo `E`. Complete esse método com uma cópia do algoritmo de ordenação por fusão adaptado para esse tipo de elementos. (Tem de usar o método `compareTo` para comparar elementos.) Teste a função num pequeno programa `TestGenericSort` que ordene os seus argumentos (strings).

```
$ java -ea TestGenericSort maria joao ana tomas antonio
[ana, antonio, joao, maria, tomas]
```

Exercício 9.5

O programa `ListSort` lê um ou mais ficheiros de texto e ordena as suas linhas. Para isso, armazena as linhas numa lista ligada e depois ordena-as usando uma versão do algoritmo QuickSort para listas genéricas. Preencha as lacunas na função de ordenação para completar o programa.

Exercício 9.6

No programa `GetWords`, a função `extractWords` extrai para um *array* todas as palavras existentes num ficheiro de texto. Altere o programa para extrair palavras de vários ficheiros de texto passados como argumentos e produzir uma listagem por ordem lexicográfica, sem palavras repetidas e indicar o número total de palavras listadas.

Exercício 9.7

Altere o programa anterior por forma a que a informação por ele gerada passe a ser persistente. Assim, no fim da execução do programa a lista de palavras (sem repetição) por ele gerada deverá ser registada num ficheiro de texto (`words.txt`), e a (eventual) lista de palavras existente nesse ficheiro deverá ser lida pelo programa no início da sua execução.

Aula Prática 10

Resumo:
- Pilhas (*stacks*) e filas (*queues*).

Exercício 10.1

Faça um programa `Palindrome` que detecte se uma sequência de letras e dígitos é um palíndromo (ou seja, se a sequência lida do início para o fim é igual à sequência lida do fim para o início). O programa deve ignorar todos os caracteres que não sejam letras ou dígitos, assim como ignorar a diferença entre maiúsculas e minúsculas. Por exemplo as frases: `"somos"` e `"O galo nada no lago"` são palíndromos. Utilize uma *pilha* e uma *fila* para resolver este problema.

Exercício 10.2

O programa `SolveHanoi` permite visualizar a resolução do problema das Torres de Hanói, passo a passo. Crie a classe `HanoiTowers` que simula o problema. A classe deve incluir três campos do tipo pilha, um para cada torre. O construtor deve iniciar a primeira torre com n discos e manter as outras duas vazias. O método `solve` deve aplicar um algoritmo idêntico ao discutido na aula de recursividade para resolver o problema.

A classe pode (e deve) definir métodos auxiliares, em particular:

- Um método `moveDisk(a, b)` que simule a operação fundamental de mover um disco de uma torre para outra, modificando as pilhas correspondentes.
- Um método, a ser chamado após cada movimento, que mostre o estado atual do problema, incluindo os conteúdos das três torres. O aspeto pretendido poderá ser como nos exemplos abaixo:

```
After 0 moves:          After 1 moves:    ...       After 31 moves:
A: [5, 4, 3, 2, 1]      A: [5, 4, 3, 2]             A: []
B: []                   B: []                       B: []
C: []                   C: [1]                      C: [5, 4, 3, 2, 1]
```

- Deve acrescentar à classe `Stack` um método `reverseToString` que devolva, sem modificar, o conteúdo da pilha numa string, indo da base para o topo.

Exercício 10.3

As técnicas de processamento digital de sinais usam frequentemente uma estrutura de armazenamento chamada *linha de atraso* (*digital delay line*), que é essencialmente uma fila de tamanho fixo que permite acesso direto (de leitura) a qualquer dos elementos. Quando a linha de atraso é criada, todos os elementos são iniciados com um valor pré-definido (geralmente zero). Quando se introduz um novo elemento, é colocado no fim da fila e, simultaneamente, o elemento mais antigo é descartado automaticamente.

O programa `TestDelayLine.java` usa uma linha de atraso para mostrar a temperatura média das últimas 24 horas, hora-a-hora ao longo de vários dias.

a. Complete a classe `DelayLine` com os métodos necessários para que o programa funcione corretamente.

b. Uma vez que a linha de atraso requer acesso aleatório e o tamanho é fixo, é vantajoso implementá-la com a técnica de array circular. Crie uma classe `DelayArray` com a mesma interface, mas usando essa implementação. Teste a nova classe num programa `TestDelayArray`.

Exercício 10.4

Construa uma calculadora com as quatro operações aritméticas básicas que funcione com a notação pós-fixa (*Reverse Polish Notation*). Nesta notação os operandos são colocados antes do operador. Assim `2 + 3` passa a ser expresso por `2 3 +`. Esta notação dispensa a utilização de parênteses e tem um implementação muito simples assente na utilização de uma pilha de números reais. Sempre que aparece um operando (número) ele é carregado para a pilha. Sempre que aparece um operador, são retirados os dois últimos números da pilha e o resultado da operação é colocado na pilha. Se a pilha não tiver o número de operandos necessário, então há um erro sintático na expressão.

O programa deve ler os operandos e os operadores do *standard input*, separados por espaços. Por exemplo, para calcular $4 * (3 - 1)$ poderá usar o comando:

```
$ echo "4 3 1 - *" | java -ea RPNCalculator
Stack: [4.0]
Stack: [4.0, 3.0]
Stack: [4.0, 3.0, 1.0]
Stack: [4.0, 2.0]
Stack: [8.0]
```

Exercício 10.5

Um dos problemas que os vectores em Java podem colocar reside no facto de o seu número de elementos ser imutável (uma vez criado o vector). Essa limitação faz com que, na presença de problemas onde a dimensão máxima do vector possa ter de variar em tempo de execução, sejamos obrigados a tirar cópias do vector (para um novo vector de dimensão adequada), ou a utilizar outros tipos de dados mais flexíveis.

Neste exercício pretende-se desenvolver uma classe `BlockArrayInt` com uma interface tipo vector (acesso aleatório aos elementos por intermédio de um índice inteiro), mas em

que a representação interna desse vector assenta numa lista ligada de blocos, sendo cada bloco um vector de dimensão fixa (definida no construtor da classe). Se o número de elementos por bloco for `blockSize`, então os índices de `[0, blockSize-1]` apontarão para dentro do primeiro bloco (primeiro elemento da lista ligada), os índices `[blockSize, 2*blockSize-1]` apontarão para o segundo, etc.

Para testar a classe, o programa `TestBlockArray.java` armazena (num objecto do tipo `BlockArrayInt`) e escreve todos os números primos encontrados até um valor dado pelo utilizador como argumento do programa. Nesse ficheiro pode também encontrar a interface desejada para a classe, mas onde a implementação está incompleta (apenas define um bloco).

Comece por experimentar o programa com valores menores do que 233 para perceber o funcionamento do programa. Caso utilize um valor acima de 232 verificará que ocorre um erro em tempo de execução. Implemente a classe usando a representação interna acima descrita, por forma a que o programa funcione para qualquer valor numérico. Quando estiver a funcionar experimente utilizar o valor 1000000, por exemplo.

Aula Prática 11

Resumo:
- Aplicações de dicionários;
- Dicionários implementados como listas de pares chave-valor.

Exercício 11.1

Os utilizadores de um sistema informático são geralmente autenticados através de um nome e senha. Crie um programa `CheckPasswd` que leia os nomes e senhas de um ficheiro dado na linha de comando e em seguida simule o processo de autenticação de utilizadores. Use a classe `p2utils.KeyValueList` para guardar as associações entre nomes de utilizadores e senhas. O programa deve terminar quando for detectado *fim-de-ficheiro* (`EOF`), o que numa consola Unix se faz introduzindo `Ctrl+D` no início de uma linha.

```
$ java -ea -jar CheckPasswd.jar senhas.txt
Username: carlos
Password: minhasenha
Authentication successful

Username: ines.m
Password: ines#m
Authentication failed

Username: <Ctrl-d>
```

Exercício 11.2

O programa `CountWords` determina e apresenta a tabela de frequências de ocorrência das palavras contidas num ou mais ficheiros de texto cujos nomes são dados na linha de comando. O programa deve usar uma `KeyValueList` para resolver o problema. Cada palavra será uma chave de acesso e o valor associado será simplesmente o contador de ocorrências dessa palavra.

a. Complete a função principal para atualizar o contador correspondente a cada palavra processada. Para tornar o processo insensível a diferenças entre maiúsculas e minúsculas, pode usar a função `toLowerCase()` da classe `String` para converter as palavras para minúsculas antes de as introduzir na lista.

b. Complete o método `toString(left, sep, right)` da classe `KeyValueList` que deve devolver uma representação da lista como uma sequência de pares *(chave, valor)*

separados entre si pela string `sep` e delimitada pelas strings `left` e `right`. Verifique o resultado das invocações no programa.

c. Complete a função `mostFrequent` para descobrir a palavra mais frequente e indicar a sua frequência relativa.

Exercício 11.3

O ficheiro `numbers.txt` contém uma lista de números e respetivas traduções por extenso em língua inglesa. Fazendo uso de um dicionário, escreva um programa `TranslateNumbers` que traduza num texto, todas as ocorrências de números por extenso pelo respectivo valor numérico, mantendo as restantes palavras. Exemplo de utilização:

```
$ echo "A list of numbers: two hundred thousand five hundred twenty four" | java -ea TranslateNumbers
A list of numbers: 2 100 1000 5 100 20 4
```

Exercício 11.4

Utilizando o dicionário do exercício anterior, crie um programa que converta um número escrito por extenso em língua inglesa para o respectivo valor numérico. Por exemplo:

```
$ echo "two thousand thirty three" | java -ea NumberValue
two thousand thirty three -> 2033

$ echo "eight million two hundred thousand five hundred twenty four" | java -ea NumberValue
eight million two hundred thousand five hundred twenty four -> 8200524
```

Tenha em consideração as seguintes regras na construção do algoritmo:

- Os números são sempre descritos partindo das maiores ordens de grandeza para as mais pequenas (*million, thousand, ...*);
- Sempre que os números se sucedem por ordem crescente (*eight million* ou *two hundred thousand*), o valor vai sendo acumulado por multiplicações sucessivas ($8 * 1000000$ ou $2 * 100 * 1000$);
- Caso contrário, o valor acumulado é somado ao total.
- Não é preciso detetar números mal formados como: *one one million*, *eleven one* ou outros.

Exercício 11.5

O acesso a cada elemento numa `KeyValueList` tem complexidade linear no número de elementos. Assim, quanto maior o número de elementos, mais crítico se torna optimizar o acesso. Uma optimização possível consiste em manter a lista ordenada por chave. Crie uma classe `SortedKeyValueList`, com as seguintes melhorias em relação à classe `KeyValueList`:

- O método `set` deve garantir que a lista está sempre ordenada por chave.
- Modifique também o método `contains` para tirar partido da ordenação dos elementos. Assim, só precisa de percorrer a lista até encontrar a chave procurada (caso em que devolve `true`) ou até encontrar uma chave maior (caso em que devolve `false`).

Modifique uma cópia do programa `CountWords` para usar e testar a nova classe.

Aula Prática 12

Resumo:
- Aplicações de dicionários;
- Dicionários implementados como tabelas de dispersão.

Exercício 12.1

Acrescente à classe `HashTable` do pacote `p2utils` os seguintes métodos:

- `keys()` - devolve um array com todas as chaves existentes na tabela de dispersão.
- `toString()` - Devolve uma representação da tabela de dispersão em cadeia de caracteres de acordo com o seguinte formato: $\{(k_1, e_1), ..., (k_n, e_n)\}$. Dica: poderá invocar o `toString` generalizado para obter os elementos de cada `KeyValueList`.

Pode usar o programa `TestHashTable` para testar os novos métodos.

Exercício 12.2

O supermercado *DaEsquina* aceita encomendas e faz as respectivas entregas ao domicílio. O gerente decidiu instalar um sistema automático de processamento de encomendas que permita, não só registar encomendas recebidas e dar baixa de encomendas entregues, mas também saber em cada momento quantas unidades de cada produto estão pedidas. As encomendas, representadas pela classe `Order` (disponibilizada em anexo), são processadas por ordem de chegada. Implemente assim uma classe `SupermarketOrdering` com os seguintes métodos:

- `enterOrder(order)` - regista uma nova encomenda;
- `serveOrder()` - dá baixa da encomenda mais antiga e devolve-a;
- `query(product)` - devolve o número de unidades de um dado produto que estão pedidas nas encomendas actuais;
- `displayOrders()` - imprime a lista de encomendas, por ordem de chegada, e o número total de unidades encomendadas de cada produto;

Os métodos `enterOrder`, `serveOrder` e `query` devem ter uma complexidade temporal $O(1)$ no número de encomendas. O programa `TestSupermarket` permite testar a classe pedida.

Exercício 12.3

No exercício 11.2 obteve o número de ocorrências de palavras num texto e, com base nessa informação, a palavra mais comum e respectiva frequência relativa. Podemos generalizar este tipo de análise para obter estatísticas de ocorrência de sequências de várias palavras. Essa informação permite criar modelos para prever a próxima palavra num browser ou para auxiliar no reconhecimento de fala, por exemplo. Para simplificar, vamos considerar apenas sequências de 2 palavras consecutivas, também chamadas de *bigramas*.

a. Complete o programa `CountBigrams` para que determine o número de ocorrências de todos os bigramas existentes num conjunto de ficheiros de texto. Depois de processados os textos, o programa deve apresentar a lista completa de bigramas e respectivas frequências absolutas, sem qualquer exigência em termos de ordem. **Sugestão:** a chave para o dicionário pode ser uma combinação adequada das duas palavras do bigrama.

b. Altere o programa para apresentar a listagem por ordem alfabética dos bigramas.

c. Altere o programa para mostrar o bigrama mais comum.

Pode testar o programa com ficheiros de texto descarregados do Projeto Gutenberg, por exemplo.

Exercício 12.4

A solução proposta no problema anterior, usar o bigrama como chave, não é a mais conveniente se quisermos saber que palavras ocorrem com maior frequência antes ou depois de uma certa palavra.

Desenvolva uma nova solução para contagem de bigramas (`CountBigrams2`) baseada num dicionário que usa cada palavra como chave e o valor associado é uma tabela das palavras que lhe sucedem e respetivas contagens. Após processamento dos textos, o programa deve apresentar a informação no seguinte formato:

```
...
has -> {(been, 2), (access, 1), (appropriate, 1)}
...
to -> {(them, 1), (using, 1), (produce, 4), (my, 1), (the, 14), ... }
...
```

A seguir, o programa deve pedir uma palavra ao utilizador e indicar a palavra que mais frequentemente surge a seguir a essa nos textos. (Pode reutilizar a função `mostFrequent` feita no exercício 11.2 ou 12.3.)

Exercício 12.5

O ficheiro `groups1.csv` tem uma lista de alunos que fizeram a primeira prova de avaliação de Programação 2 no ano de 2017-2018. Cada linha contém o número de um aluno e o computador onde fez a prova, separados por um TAB (`"\t"`). Como a prova foi feita em grupo, vários alunos partilharam o mesmo computador e o nome do computador serve de identificador do grupo respetivo. Para a segunda avaliação, os grupos tinham de ser diferentes.

a. Dado um ficheiro nesse formato, o programa `CheckGroups` deve pedir números de alunos dois-a-dois e verificar se estiveram no mesmo grupo ou não. Complete a função de leitura do ficheiro e a função principal para atingir esse objetivo.

```
aula12$ java -ea -jar CheckGroups.jar groups1.csv
Student1? 71904
Student2? 84672
Students 71904 and 84672 were NOT in the same group!

Student1? 75762
Student2? 88888
Students 75762 and 88888 were NOT in the same group!

Student1? 75762
Student2? 76442
Students 75762 and 76442 WERE in the same group!

Student1? <Ctrl+D>
```

b. Escreva um programa `ListGroups` que leia um ficheiro de grupos e mostre os alunos de cada grupo, um grupo por linha. Sugestão: convém criar um dicionário que associe cada grupo à respetiva lista de alunos. Pode criar uma nova função de leitura ou reutilizar a função de leitura da alínea anterior e acrescentar outra função para converter o dicionário obtido.

```
aula12$ java -ea -jar ListGroups.jar groups1.csv
l040106-ws10: [88970, 88826]
l230213-ws03: [88750, 88998]
l230213-ws04: [84983]
l040106-ws03: [89189, 88906]
l230213-ws06: [76442, 88932, 75762]
...
```

c. O ficheiro `groups2.csv` tem os pares (aluno, computador) da segunda prova de avaliação. Escreva um programa `FindGroupCollisions` que, dados os ficheiros de duas avaliações, descubra se na segunda avaliação algum aluno fez grupo com outro aluno com quem já tivesse feito grupo na primeira. Note que pode ter havido grupos de 1, 2 ou mais alunos em qualquer das avaliações e que alguns alunos foram apenas a uma das avaliações. Teste o programa também com os ficheiros `groups1.csv` e `groupsX.csv` que têm algumas "colisões".

```
aula12$ java -jar FindGroupCollisions.jar groups1.csv groups2.csv

aula12$ java -jar FindGroupCollisions.jar groups1.csv groupsX.csv
In groupsX.csv, group l230204-ws06: [88857, 85123]
includes students that were in same group
in groups1.csv: {(l230214-ws02, [88857, 85123])}

In groupsX.csv, group l230214-ws04: [80248, 89280, 79996]
includes students that were in same group
in groups1.csv: {(l230211-ws02, [80248, 79996]), (l230205-ws10, [89280])}
```

Aula Prática 13

Resumo:
- Dicionários;
- Árvores binárias de procura na implementação de dicionários;
- Árvores Binárias.

Exercício 13.1

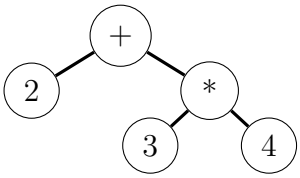
Implemente o tipo de dados abstrato de um dicionário na classe `BinarySearchTree`, o qual se deve basear numa árvore binária de procura. Experimente a classe com os exercícios da aula anterior ou usando o programa `TestBST` fornecido.

Exercício 13.2

Podemos escrever expressões aritméticas de três formas distintas, consoante a posição do operador:[1]

- *Notação infixa*: `2 + 3` ou `2 + 3 * 4` ou `3 * (2 + 1) + (2 - 1)`
- *Notação prefixa*: `+ 2 3` ou `+ 2 * 3 4` ou `+ * 3 + 2 1 - 2 1`
- *Notação sufixa*: `2 3 +` ou `2 3 4 * +` ou `3 2 1 + * 2 1 - +`

Independentemente da notação utilizada, uma expressão aritmética pode ser representada por uma árvore binária em que os nós representam as (sub)expressões existentes. Os números são expressões constantes e, portanto, serão as folhas da árvore. Por exemplo, a expressão (prefixa) `+ 2 * 3 4` é expressa pela árvore binária:



A geração desta árvore binária a partir de uma expressão em notação *prefixa* é bastante simples (quando feita recursivamente):

[1] No exercício 10.4 já utilizámos a notação sufixa como forma de simplificar o cálculo de expressões.

```
createPrefix() {
   if (in.hasNextDouble()) { // next word is a number
      // leaf tree with the number
   }
   else { // next word is the operator
      // tree with the form: operator leftExpression rightExpression
      // leftExpression and rightExpression can also be created with createPrefix
   }
}
```

a. Implemente uma classe – `ExpressionTree` – que cria uma árvore binária a partir de uma expressão em notação *prefixa* para os 4 operadores aritméticos elementares (`+`, `-`, `*` e `/`). Considere que cada número e operador é uma palavra a ser lida no *standard input*.

b. Implemente um novo serviço na classe – `printInfix` – que escreve a expressão (já lida) na notação *infixa*. Execute o programa `ParsePrefixExpression.jar` para ver o formato desejado.

c. Implemente a classe de uma forma robusta por forma a detectar expressões inválidas (note que a responsabilidade por uma expressão inválida é exterior ao programa pelo que deve fazer uso de uma abordagem defensiva).

d. Implemente um novo serviço na classe – `eval` – que calcula o valor da expressão.

Nota: A criação da árvore binária com base numa expressão em notação *sufixa* é um pouco mais complexa. Se tiver essa curiosidade pode tentar fazer essa implementação.