FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Departamento de Engenharia Electrotécnica

Sistemas de Telecomunicações

2012/2013

Introdução ao desenvolvimento de aplicações que funcionam em rede usando a linguagem Java (Versão 3)

Mestrado Integrado em Engenharia Electrotécnica e de Computadores

http://tele1.dee.fct.unl.pt

Paulo Pinto
Luis Bernardo
Rodolfo Oliveira

ÍNDICE

1. OBJETIVO

Familiarização com a linguagem Java e com o desenvolvimento de aplicações utilizando *sockets* UDP e TCP. Este documento fornece uma primeira introdução à linguagem Java e descreve as classes mais relevantes para a disciplina da biblioteca do Java para a programação de aplicações que comunicam numa rede TCP/IP.

Os parágrafos sombreados contêm informações adicionais. Numa primeira leitura pode passar sem ler. Estes conceitos são importantes para o último trabalho.

2. A LINGUAGEM JAVA

A linguagem Java é uma linguagem orientada por objetos que tem uma sintaxe semelhante às linguagens C e C++, com uma grande exceção: não são definidos apontadores. As aplicações desenvolvidas em Java são vulgarmente compiladas para *bytecode*, que é depois interpretado em máquinas virtuais Java (ou compilado para código nativo num compilador JIT). A grande vantagem da plataforma Java é precisamente a possibilidade de correr em qualquer arquitetura.

O trabalho proposto nesta disciplina vai ser desenvolvido para a versão 6 ou 7 de Java SE Development Kit (JDK). Para o ambiente de desenvolvimento propõe-se a utilização do NetBeans IDE, totalmente desenvolvido em Java, que pode correr em qualquer sistema operativo. Para o obter os alunos devem começar pela ligação http://www.oracle.com/technetwork/java/index.html e instalar o NetBeans IDE Java SE.

Para realizar o trabalho proposto vai ser necessário recorrer a um subconjunto reduzido das classes da biblioteca Java. Este documento contém uma pequena introdução ao Java para um programador de C e apresenta as classes necessárias de uma forma simples. Deve servir como referência ao longo do semestre. Os alunos podem, no entanto, usar qualquer outra classe disponível na biblioteca.

2.1. Primeiras diferenças entre C e Java

Na linguagem C os dados das aplicações são geralmente declarados em estruturas (*struct*) e os algoritmos são realizados em funções.

Na linguagem Java as *classes* incluem ambas as funcionalidades – tanto guardam dados como definem as funções que os manipulam. Mas classes são apenas a descrição dos dados e funcionalidades. Têm de ser instanciadas para se executar qualquer coisa.

A instância de uma classe designa-se por *objeto*. Tem memória própria e estado. Pode ser pensado como uma variável dessa classe

O exemplo seguinte ilustra o cálculo da diferença entre dois números complexos em C (estrutura `Complex`) e em Java (classe `Complex`).

Vamos então comparar o C com o Java:

Em C é definido um tipo chamado de `Complex` que é uma estrutura. A variável deste tipo é criada no `main` e a variável só é conhecida no `main` (ou a quem o `main` a der).

Em Java é definida uma classe `Complex` que contém as variáveis que podem ser públicas ou privadas (`public` ou `private`) para indicar que estão acessíveis do exterior ou só o objeto as pode manipular (e neste caso nem as pode dar a ninguém).

Em C a função `distance` é uma função do programa. Pode ser chamada da `main` ou de outra qualquer função.

Em Java o objeto tem as suas funções, umas `public` que podem ser chamadas de fora, outras `private` que só podem ser chamadas de dentro do objeto.

Em C as variáveis foram criadas no `main` e foram-lhes dados valores iniciais.

Em Java o objeto foi criado com a instrução `new` e nessa instrução foram indicados os valores iniciais. O código que é corrido quando se invoca `new` chama-se *construtor* e tem o mesmo nome do que o da classe.

```c
// C
#include <math.h>
#include <stdio.h>

typedef struct Complex {
  double re;   // Parte real
  double im;   // Parte imaginária
} Complex;

// Função
double distance(Complex c1, Complex c2) {
  return sqrt(pow(c1.re-c2.re,2)+
              pow(c1.im-c2.im,2));
}

int main() {
  Complex a, b;
  double dist;
  a.re= 1.0;
  a.im=b.re=b.im= 0.0;
  dist= distance(a, b);
  printf("Distancia= %lf\n",dist);
  return 1;
}
```

```java
// Java
public class Complex {
  private double re;   // Parte real
  private double im;   // Parte imaginária

// Construtor
  Complex(double r, double i) {
    re= r;     // ou this.re= r;
    im= i;     // ou this.im= i;
  }
// Função da classe Complex
  public double distance(Complex other) {
    return Math.sqrt(Math.pow(re-other.re,2)+
                     Math.pow(im-other.im,2));

  public static int main() {
    Complex a= new Complex(1.0, 0.0);
    Complex b= new Complex(0.0, 0.0);
    double dist;
    dist= a.distance(b);
    System.out.print("Distancia="+dist+"\n");
    return 1;
  }
}
```

Antes de se continuar a descrever este exemplo é preciso estudar mais alguns conceitos…

2.2. Tudo são objetos

Em Java só existem objetos! No entanto, podemos distinguir três situações: tipos primitivos, objetos, e objetos estáticos (`static`).

2.2.1. Tipos Primitivos

Os tipos primitivos têm um tratamento especial do Java. Pense neles como os tipos básicos de C. Os tipos primitivos de Java e o seu tamanho são (repare que um `char` tem 16 bits):

Tipo primitivo	Tamanho
boolean	--
char	16 bits
byte	8 bits
short	16 bits
int	32 bits
long	64 bits
float	32 bits
double	64 bits
void	--

São declarados como em C, inicializados também como em C, e usados como em C. Como os tamanhos são fixos, não existe o operador `sizeof ()` em Java.

2.2.2. Classes e Objetos

Para além dos tipos primitivos existem os outros tipos/classes que têm de ser definidos (os seus dados e métodos/funções). Imagine uma classe designada `AnyClass`. Ao defini-la ficamos a saber que ela existe. Como é que a podemos usar?

O Java só trabalha com referências para objetos. Vamos criar então uma referência para objetos da classe `AnyClass`.

```
AnyClass   a;
```

Esta referência não está "ligada" a nenhum objeto. Ainda não existe nenhum objeto. Não existe memória associada. Para criar um objeto tem de se utilizar a instrução `new`. Com a instrução abaixo, já temos um objeto e a referência "`a`" indica/aponta para este novo objeto.

```
a = new AnyClass ();
```

Tirando os tipos primitivos, tudo em Java são objetos e usam-se como em cima.

Uns objetos interessantes são os objetos "*wrapper*" (envolventes) dos tipos primitivos. São os seguintes:

Tipo primitivo	**Wrapper Type (tipo envolvente)**
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
void	Void

Como os tipos envolventes já são objetos normais, tem de se usar a instrução `new`. Repare nas instruções abaixo (em que uma delas é de um tipo primitivo):

```
char      c  = 'x';
Character c1 = new Character ('x');
Character c2 = new Character (c);
```

Os tipos envolventes servem para "trazer" os tipos primitivos para o mundo orientado a objetos. A utilidade maior vai ser a possibilidade de colocar tipos primitivos em listas, conjuntos, mapas, tabelas, etc., que foram criados para conter qualquer tipo de objetos (mas apenas objetos e não tipos primitivos). O Java só permite tipos primitivos em vetores. Estas classes envolventes têm depois um conjunto de métodos para tratamento mais fino dos dados. Por exemplo acesso a bits, rodar bits, etc. Um aspeto interessante é o uso destes métodos, mesmo que não haja objeto (ver a próxima secção).

No entanto, a liberdade não é completa com os tipos envolventes. Isto porque ao se atribuir um valor para um objeto de um tipo envolvente, esse valor não pode mudar. Assim, ao colocarmos um tipo envolvente numa lista com um certo valor, esse valor não pode mudar. É evidente que isto pode não ser um problema para certos propósitos, mas se se pretender mudar o valor tem mesmo de se criar um objeto "normal", que contenha um `int`, por exemplo.

2.2.3. Objetos estáticos

Existem objetos que podem ter a característica de estáticos, dada pela palavra-chave `static`. São usados para duas situações:

i. podem ser acedidos sem se criar o objeto (o Java tem de garantir isso);
ii. um dado estático é partilhado por todos os objetos dessa classe (não é muito interessante nesta altura da explicação).

Três objetos estáticos que vão ser muito úteis desde o início deste trabalho são os objetos da classe `System` chamados de `out`, `in`, e `err`. Estes objetos estão relacionados com o ecrã e o teclado. Lembre-se do `stdout`, `stdin`, e `stderr` do C.

Os métodos dos tipos envolventes apresentados acima têm a característica de `static`. Isto quer dizer que podem ser usados mesmo que não existam objetos. O exemplo seguinte mostra como se pode passar uma `String` para um inteiro (`int`), mesmo não se tendo criado nenhum objeto `Integer`.

```
int n;
String str= "123";
n= Integer.parseInt(str);
```

2.2.4. Constantes

Em Java, usa-se o modificador `final` para indicar que algo não pode ser mudado. Consoante se está a tratar de tipos primitivos, objetos, argumentos, métodos ou classes, os significados são diferentes. Neste documento só se aborda os dois primeiros casos:

- Dados (tipos primitivos): o valor de um dado com o modificador `final` nunca se altera. Se se pretender que use apenas um local na memória usa-se também o `static`. Se os tipos primitivos forem variáveis de uma classe são acedidos com "`NomeClasse.Variavel`". Atenção que é nome de classe e não nome de objeto. Por exemplo, se a declaração abaixo estivesse numa classe chamada de `SomeClass`, a variável `MIN` seria acedida como "`SomeClass.MIN`":

```
public static final int MIN= 1;
```

- Dados (objetos): o que nunca se altera é a referência e não o objeto. Uma vez inicializada a apontar para um objeto, não pode ser alterada. Mas o objeto pode ser alterado.

2.3. Primeiras diferenças entre C e Java (revisitado)

Voltando aos programas que calculam a distância de dois complexos pode-se ver que o modo de chamar um método é usando a referência seguida de um ponto e do nome da operação. No caso da distância foi decidido que o outro valor vai como parâmetro: `a.distance(b)`. I.e. invoca-se o método `distance` sobre os dados do objeto "`a`", com argumento "`b`". Na linguagem C são passadas ambas as variáveis por argumento.

A linguagem Java simplifica muito a gestão de memória, pois liberta automaticamente toda a memória alocada para um objeto quando este deixa de estar referenciado. Por exemplo, no fim de uma função ou quando a referência tem o valor `null` (i.e. objeto não inicializado, ou colocado com esse valor de propósito). O importante em Java é que o programador não tem de se preocupar com a gestão de memória.

Repare que a função `main` tem a palavra-chave `static`. Segundo o que foi dito atrás, é como se o programa fosse chamado sem o objeto (que é o programa) ser criado.

O exemplo ilustra também a utilização das funções externas, neste caso de matemática – `sqrt` e `pow`.

Em C é preciso incluir o ficheiro de definições da biblioteca de matemática "`math.h`".

Em Java usou-se um objeto chamado de `Math`. Notar que este objeto não foi criado (é estático), e que não foi preciso incluir a biblioteca. Se fosse preciso, ter-se-ia de colocar uma instrução de importação. Por exemplo, se quiséssemos um objeto de uma classe parecida com a de vetores (ver adiante) colocaríamos "`import java.util.ArrayList;`". Se quiséssemos todas as classes de utilitários poderíamos escrever "`import java.util.*;`"

2.4. Operadores

Quase todos os operadores só funcionam para tipos primitivos.

Existem as exceções de '=', '==' e '!=', que funcionam para todos os objetos (e são um foco de confusão) e o operador +' e '+=' que funciona também para a classe `String`.

A atribuição '=' pode ser confusa pois se "`A`" e "`B`" forem referências para objetos de uma certa classe e se se fizer "`A = B`", então ambas as referências "`A`" e "`B`" ficam a apontar para o mesmo objeto (não se criou um novo objeto apontado por "`A`" para o qual foram copiados valores iguais aos do objeto apontado por "`B`").

Os operadores relacionais '==' e '!=' trabalham com as referências. É como se estivéssemos a comparar ponteiros! Se quisermos comparar os objetos, devemos usar o método `equals()` que existe para todos os objetos – `n1.equals(n2)`. Atenção que este método só deve ser usado para classes já existentes, caso contrário (numa classe feita por nós) compara outra vez apenas os ponteiros...

Atenção que quando se passa um objeto numa lista de parâmetros, está-se a passar a referência. Lembre-se do C e de se usarem apontadores nas listas de parâmetros.

O operador +' (e '+=') quando aplicado a objetos da classe `String` significa concatenação. De um modo geral, se uma expressão começar com um `String`, todos os operandos que se seguem têm de ser `String`, ou são convertidos para `String`. Todos os objetos (os não primitivos) têm um método `toString()` definido por omissão para fazer a conversão.

2.5. Um primeiro projeto

Arranque agora o NetBeans e escolha um projeto novo. Escolha a categoria "*Java*" e um projeto "*Java Application*". Chame-lhe *HelloWorld* e escolha a diretoria onde o colocar. Ao ciar o novo projeto é criada a diretoria e alguns ficheiros, e aparece um código na janela grande à direita.

As classes estão incluídas num pacote, que neste caso tem um nome parecido – *helloworld*. Foi criada a classe *HelloWorld* e dentro dela foi também criada uma função `public` e `static` chamada `main`. Note que *HelloWorld* não tem outros métodos nem atributos. É uma classe vazia. Não faz sentido fazer `new`.

Escreva agora como única instrução dentro do `main`, a instrução seguinte que escreve uma linha (com o "`enter`" no final) no ecrã.

```
System.out.println ("Hello World");
```

e corra o programa premindo na tecla triangular verde.

Foi escrita numa janela em baixo à direita a frase "*Hello World*".

Parabéns pois executou o seu primeiro programa!

Só que este programa não é bem um programa em Java pois não tem objetos. Tente comparar este programa com o programa sobre complexos, mostrado no início.

No caso do *HelloWorld* o `main` começou, escreveu uma frase e acabou. Não criou objetos nenhuns, nem invocou operações em objetos nenhuns.

O código do programa deve estar assim:

```
package helloworld;

/**
 *
 * @author paulopinto
 */
public class HelloWorld {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
        System.out.println("Hello World");
    }
}
```

2.6. Tratamento de Erros e Exceções

Uma exceção é algo que acontece (e.g., divisão por zero, apontar para uma zona de memória proibida) que impede que o programa corra como se nada tivesse acontecido. Normalmente não existe informação suficiente para tratar do problema nesse ponto. Então, tem de se parar nesse momento e alguém, algures, tem de perceber o que fazer. Se não houver esse alguém, o programa é parado pelo sistema operativo.

O tratamento de erros e exceções em C é baseado em convenções: definem-se valores especiais que as funções devolvem e o programador deve testar esses valores depois da função retornar. O que acontece é que a maioria dos programadores nunca testa esses valores e tudo o que se convencionou falha. Uma das razões é que o código começa a ficar ilegível.

O Java introduziu o tratamento de erros e exceções na própria linguagem e tem uma abordagem muito sistemática. As partes do código que tratam as exceções (*exception handlers*) estão bem identificadas não complicando o código "normal". A ideia é que não é preciso fazer a pergunta depois da função retornar. Se aconteceu uma exceção "A", existe uma parte do código que trata da exceção "A".

Um aspeto interessante é que o programador pode também gerar exceções. É criado um objeto exceção (tal como qualquer outro objeto em Java) a execução do programa é parada e passada ao mecanismo de tratamento de exceções que vai procurar a parte de código que trata esta exceção. Se não a encontrar o programa é parado. No código seguinte a exceção `NullPointerException` é levantada se a referência "`t`" for `null`. A ideia é o programador ter a seguinte atitude: "Não quero pensar mais nisto. Alguém que resolva".

```
if (t == null)
   throw new NullPointerException ();
```

Existem dois construtores de exceções: o primeiro é o normal, como mostrado em cima; o outro aceita como argumento uma *string*, para podermos dar mais alguma informação.

```
if (t == null)
   throw new NullPointerException ("t = null");
```

As exceções são apanhadas numa parte do código que poderia ser designada por "zona guardada". A zona guardada é um bloco precedido pela palavra-chave `try`.

```
try {
     // Code that might generate exceptions
}
```

Nesta zona podem-se chamar os métodos que se quiser (ou o mesmo muitas vezes) sem ter de testar condições de erro. As exceções são tratadas nos *exception handlers*, em que existe um por cada tipo de exceção, e são colocados depois do `try`. Eles têm a palavra-chave `catch` como está mostrado em baixo:

```
try {
     // Code that might generate exceptions
}
} catch(Type1 id1) {
  // Handle exceptions of Type1
} catch(Type2 id2) {
  // Handle exceptions of Type2
} catch(Type3 id3) {
  // Handle exceptions of Type3
}
```

Este tipo de estrutura (com o `catch`) significa as exceções em Java são interpretadas como tão graves que o código normal tem de deixar de ser corrido. O programa pode, ou não, acabar.

Se o programador quiser criar a sua própria exceção, tem de herdar (ver próxima secção) de outra classe. O código seguinte mostra como se cria uma exceção chamada `SimpleException`:

```
class SimpleException extends Exception {}

public class SimpleExceptionDemo {
  public void f() throws SimpleException {
    System.out.println("Throwing SimpleException from f()");
    throw new SimpleException ();
  }
  public static void main(String[] args) {
    SimpleExceptionDemo sed = new SimpleExceptionDemo();
    try {
      sed.f();
    } catch(SimpleException e) {
      System.err.println("Caught it!");
    }
  }
}
```

As exceções que uma função "levanta" estão indicadas usando a palavra-chave `throws`, como está mostrado em baixo.

```
void func() throws TooBig, TooSmall, DivZero {  ...  }
```

É possível fazer um *exception handler* que apanhe qualquer exceção. Para isso, ele deve apanhar a classe base das exceções (deve talvez ser usado no final da lista de `catch`):

```
catch(Exception e) {
    System.err.println("Caught an exception");
}
```

Pode acontecer que se pretenda sempre fazer qualquer coisa quer o código corra bem, quer uma exceção tenha sido levantada. Note que quando se entra num `catch`, mais nenhuma instrução depois do `catch` é executada. O bloco `finally` serve para isso mesmo: para executar um código sempre, haja ou não haja exceção.

```
try {
  // The guarded region: Dangerous activities
  // that might throw A, B, or C
```

```
} catch(A a1) {
  // Handler for situation A
} catch(B b1) {
  // Handler for situation B
} catch(C c1) {
  // Handler for situation C
} finally {
  // Activities that happen every time
}
```

2.7. Duas classes importantes

2.7.1. *Strings*

Em Java as cadeias de carateres são memorizadas em objetos da classe `String`. A construção de uma cadeia pode ser feita por concatenação (operação '+') e até se pode usar o truque já explicado de que se se começar algo com uma `String`, tudo o resto é convertido para `String`. Assim, se "`i`" e "`j`" forem dois inteiros `int`, a expressão `("" + i + j)` tem os valores dos dois inteiros sem espaços no meio.

A classe `String` pode ter uma inicialização especial, não usada para os outros objetos, talvez devido ao modo como é feita em C. O exemplo abaixo mostra os dois modos possíveis, e também o método `charAt()` da classe para devolver o caracter que está numa certa posição.

```
String   str = new String ("678");
String   str2 = "123";
char c= str.charAt(0);          // Primeiro carater da string;
```

É possível localizar a posição de uma `String` ou carater noutra `String` utilizando os métodos `indexOf()` (primeira ocorrência) ou `lastIndexOf()` (última ocorrência).

```
int posicao= str.indexOf("Java");   // Posição de "Java" em "Java is great" é 0
```

A classe `String` tem o método `substring()` para selecionar uma parte da `String`:

```
String a= "Java is great";
String b= a.substring(5);             // b é a string "is great"
String c= a.substring(0, 4);          // c é a string "Java"
```

O método `trim()` remove os espaços e tabulações no início e fim da `String`.

A comparação entre `String` é feita com os métodos `equals()`, ou `equalsIgnoreCase()` se se quiser ignorar maiúsculas e minúsculas. A comparação "`a==b`" compara referências.

`String` e tipos envolventes

É importante a relação entre os tipos envolventes e a classe `String` devido aos vários métodos que eles têm para se retirar um valor (`int`, `short`, `float`) a partir de uma `String`. Para isso existem os métodos "`parse???`". Por exemplo `parseByte`, `parseShort`, etc., que se aplicam ao respetivo tipo, e podem também ser usados sem objeto, como se viu acima.

Caso a `String` contenha um caracter inválido é gerada a exceção `NumberFormatException`.

```
int n;
String str= "123";
try {
     n= Integer.parseInt(str);
```

```
}
catch (NumberFormatException e) {
      System.out.println("Número inválido"+ e);
}
```

A operação inversa (converter para `String`) é trivial como foi explicada acima.

Antes de ver a solução no código em baixo tente acrescentar duas variáveis ao `main` do *HelloWorld*: um inteiro e um objeto `String` inicializado a "2013". O `main` deve passar o valor para o inteiro e escrever no ecrã "*Hello World 2013*". Use o tratamento de exceções.

```
package helloworld;

/**
 *
 * @author paulopinto
 */
public class HelloWorld {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        int n = 0;
        String str = "2013";
        try {
            n = Integer.parseInt(str);
        } catch (NumberFormatException e) {
            System.out.println("Número inválido" + e);
        }
// TODO code application logic here
        System.out.println("Hello World " + n);
    }
}
```

Este programa ainda não é verdadeiramente um programa com pensamento de Java pois ainda não tem propriamente objetos…

EXERCÍCIO 1: Vamos dar um primeiro passo para termos objetos.

As variáveis que estão no `main` no *HelloWorld* (o inteiro e a `String`) vão ser agora variáveis da classe *HelloWorld*. Tem de as passar para cima, logo abaixo da declaração da classe. Considere-as `public` para poderem ser acedidas de fora do objeto.

Atenção que agora tem de escrever um construtor para inicializar as variáveis (o construtor pode receber os valores iniciais pela lista de argumentos).

Não se esqueça de criar o objeto no `main`.

O programa deve retirar o valor inteiro da `String` e escrever o mesmo que o programa acima escreve: "*Hello World 2013*".

O código com a solução está na secção seguinte. Tente mudar a partir do anterior (em cima) com o que já aprendeu para ver se chega ao mesmo resultado.

2.7.2. Datas

A linguagem Java utiliza vulgarmente a classe `Java.util.Date` para representar datas, com uma precisão de até dezenas de milissegundos.

É possível obter a data atual criando um novo objeto:

```
Date dNow = new Date();              // Obtém a data+hora atual
```

A escrita formatada de datas para uma `String` é realizada utilizando um objeto da classe `java.text.SimpleDateFormat`. Em primeiro lugar temos de criar o objeto indicando-lhe

o tipo de formato que queremos que ele execute. Depois usa-se o método `format()` desse objeto.

```
Date dNow = new Date();                 // Obtém a data+hora atual
SimpleDateFormat formatter = new SimpleDateFormat("E hh:mm:ss 'em' dd.MM.yyyy");
System.out.println("A data actual é " + formatter.format(dNow));
```

Uma variável do tipo `Date` também pode ser representada utilizando o tipo `long`. Utilizando o método `getTime()`, obtém-se o número de milissegundos em relação a uma data de referência. O construtor da classe `Date` aceita argumentos do tipo `long`, conseguindo-se, desta forma, criar uma data a partir de um valor do tipo `long`.

```
long t= dNow.getTime(); // t = data dNow no formato long (n° de milisegundos)
Date nData= new Date(t);// nData = data dNow novamente no formato Date
```

Utilizando a representação no formato `long` é possível calcular diferenças entre datas.

EXERCÍCIO 2: Continuando com a classe *HelloWorld* acrescente mais um dado à classe que seja da classe `Date`. O construtor mantém a mesma assinatura (só dois argumentos), mas quando o objeto é criado o objeto da classe `Date` fica com a hora da criação. Quando tentar definir a classe `Date`, uma lâmpada há-de sugerir-lhe que tem de importar uma classe.

O programa, depois de escrever "*Hello World 2013*", deve escrever a hora da criação no formato mostrado no exemplo em cima. Quando tentar fazer isso vá aceitando as sugestões da lâmpada.

A solução está mostrada na secção seguinte. Tente conseguir o objetivo, mesmo que não consiga logo à primeira. Se vir simplesmente a solução perdeu uma boa oportunidade de começar a pensar "à Java".

2.8. Classes para conter objetos

O Java tem muitos objetos que servem para agregar outros objetos. Eles foram feitos de modo a serem muito genéricos e aceitarem todas as classes. Apenas os vetores podem, para além disso, aceitar tipos primitivos. Vão-se descrever três casos: vetores ou *arrays*, uns contentores especiais, e como fazer *arrays* de tamanho variável.

2.8.1. Vetores ou *Arrays*

Unidimensionais

Um vetor é uma sequência de objetos, ou tipos primitivos. No caso unidimensional, a sua definição pode tomar uma de duas formas:

```
int []  vect;
int vect [];        // Esta é mais para os saudosistas do C
```

A definição indica simplesmente a referência para o vetor (por isso é que não é necessário o tamanho). Ainda não existe memória. Para haver memória tem de se inicializar o vetor.

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

Todos os vetores têm um elemento por omissão chamado de `length` que contém o valor do seu tamanho (de zero a *length*-1). Por exemplo, `vect.length`.

Se se quiser alocar um certo tamanho de memória para um vetor sem ter de o inicializar, tem de se usar a instrução `new`.

```
int [] vect;
vect = new int [100];
```

Pode-se fazer tudo na mesma instrução.

```
int [] vect = new int [100];
```

Quando se tem vetores de objetos (não de tipos primitivos) tem de se usar sempre o `new`. Isto significa que não se fez a inicialização. O vetor tem simplesmente referências (apontadores) para o objeto, e tem de se criar o objeto posteriormente (pode-se, de facto, criar tudo na mesma instrução, isto é, declarar e inicializar, mas deixa-se isso para o futuro).

```
Complex [] array = new Complex [10];  // Primeiro aloca array com apontad. a null
for (int i= 0; i<10; i++)
    array[i]= new Complex(0,0);       // Aloca os elementos do array, um a um
```

Os vetores têm uma dimensão fixa, que só pode ser modificada através da criação de um novo vetor e da cópia integral dos dados. Repare como se copiam vetores.

```
byte[] aux;           // Declaração de novo vector; não aloca memória
aux= new byte[80];    // inicialização da memória para o vector
System.arraycopy(arrayBytes, 0, aux, 0, arrayBytes.length); // copia array
arrayBytes= aux;      // substitui array, a memória antiga é libertada pelo sistema
```

Multidimensionais

Os vetores podem ser multidimensionais e aplicam-se as noções anteriores. O exemplo seguinte mostra a definição com a inicialização

```
int[][] a1 = {
      { 1, 2, 3, },
      { 4, 5, 6, },
    };
```

O segundo exemplo mostra o uso do `new`

```
int[][][] a2 = new int [2] [4] [5];
```

O terceiro exemplo usa a função `pRand(i)` para gerar um valor aleatório inferior a "`i`". Usa depois o `new` para ir alocando memória. No final coloca valores em sequência. Repare no uso dos membros `length`.

```
int valor = 0;
int[][][] a3 = new int[pRand(7)][][];
for(int i = 0; i < a3.length; i++) {
  a3[i] = new int[pRand(5)][];
  for(int j = 0; j < a3[i].length; j++) {
    a3[i][j] = new int[pRand(5)];
    for(int k = 0; k < a3[i][j].length; k++)
       a3[i][j][k] = valor++;
  }
}
```

O quarto exemplo mostra a construção de uma matriz de objetos `Integer` feita também peça a peça. No final é atribuído o valor `i*j` a cada inteiro. Consegue perceber qual a dimensão da matriz que foi criada?

```
Integer[][] a5;
a5 = new Integer[3][];
for(int i = 0; i < a5.length; i++) {
  a5[i] = new Integer[3];
  for(int j = 0; j < a5[i].length; j++)
    a5[i][j] = new Integer(i*j);
}
```

Sempre que se acede a um vetor, o ambiente Java testa se o índice está dentro dos limites, gerando em caso de falha a exceção `ArrayIndexOutOfBoundsException`.

EXERCÍCIO 3: Acrescente um vetor de `int` à classe *HelloWorld* que tenha 5 posições, e tenha os valores 0, 2, 4, 6, e 8 que lhe são colocados com um ciclo `for`. Este vetor vai ser `private`. Isto implica que temos de ter métodos para podermos aceder aos seus valores. Defina um método que escreva o vetor todo, e chame esse método no final do `main`.

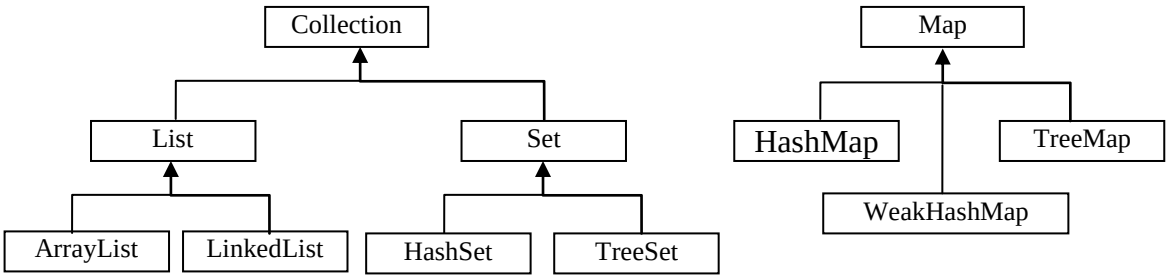
Solução na secção seguinte como é habitual.

2.8.2. Contentores – *HashMap*

Para além dos vetores, o Java tem outras classes para agrupar objetos, que trabalham sobre referências para objetos, e portanto não suportam tipos primitivos (tem de se usar os tipos envolventes caso se pretendesse usar tipos primitivos, com a limitação de que os seus valores não podem ser mudados). Estas classes dividem-se em dois grandes grupos:

- *Collection* (coleção/conjunto) – agrupam elementos individuais usando alguma regra: as **listas** (`List`) têm uma certa sequência, e os **conjuntos** (`Set`) não podem ter elementos repetidos. Para colocar elementos usa-se o método `add()`.
- *Map* (mapa/dicionário/tabela) – é um grupo de pares "chave-valor". As chaves têm de ser todas diferentes pois são o índice do contentor (em vez de um número). Para colocar elementos usa-se o método `put()`.

O esquema abaixo é um pouco simplificado, mas mostra algumas das classes existentes nas bibliotecas do Java.



Num ambiente tão genérico, existe o problema de como se mover ao longo de uma sequência e conseguir selecionar um objeto nessa sequência. O objeto iterador faz este papel. O que se pode fazer com ele é muito simples:

- Pedir ao contentor um iterador, `Iterator`, usando o método `iterator()`, do contentor. Este iterador está pronto a devolver o primeiro elemento ao se chamar o seu método `next()`.
- Obter o elemento seguinte pelo método `next()`.
- Perguntar se existe ainda um elemento seguinte pelo método `hasNext()`.
- Remover o último elemento retornado pelo método `remove()`.

Para as listas existe um iterador mais completo da classe `ListIterator`.

Deixando os contentores em geral e focalizando-nos agora na classe `HashMap`, ela tem os seguintes métodos principais:

Método	**Descrição**
put(Object key, Object value)	Adiciona o *value* e associa-o à *key*
get(Object key)	Retorna o *value* associado à *key*
containsKey( )	Testa se contém um elemento com a chave *key*
containsValue( )	Testa se contém o elemento com valor *value*

Como se pode ver, tanto os valores como as chaves podem ser de qualquer classe. Esta flexibilidade traz um problema com os iteradores para `HashMap` pois não existe uma sequência ou estrutura que se possa seguir. O problema ainda fica mais complicado porque se pode ter iteradores sobre o espaço de chaves e iteradores sobre o espaço dos valores. Sobre o espaço de chaves o que se fez foi gerar primeiro um conjunto (`Set`) (com a operação `keySet()` do `HashMap`) e depois fazer um iterador sobre a estrutura do conjunto. O iterador de chaves de um `HashMap`, `hmp`, cria-se assim:

```
hmp.keySet().iterator();
```

Atenção que este conjunto (`Set`) tem a ver com o `HashMap`. Se o `HashMap` mudar enquanto estiver a haver uma iteração, os resultados do iterador ficam indefinidos. As únicas operações de alteração que se podem fazer são os métodos de remoção do iterador.

Para o caso dos valores, eles podem ter elementos repetidos e o que se forma primeiro é uma coleção (`Collection`) (com a operação `values()` do `HashMap`) e depois um iterador sobre esta coleção. A instrução é a seguinte, e mais uma vez só se deve modificar o `HashMap` através de métodos do iterador.

```
hmp.values().iterator();
```

O código em baixo mostra a parte de um programa onde foi definido um `HashMap` em que a chave é da classe "*GPS*" (que contém as coordenadas GPS) e os elementos são da classe "*CAPACITY*" (que contém quantos quartos, salas de conferência, piscinas, etc., tem o hotel nessas coordenadas). Note a existência de *cast* pois o `HashMap` é completamente geral e trabalha com `Object`.

```
public class Region {
    public HashMap regOffer = new HashMap();
    Region () {     /*    something   */     }
    }

    void printElem(GPS loc, CAPACITY cap ) {
         /*   prints the key GPS and element (rooms, conference rooms, etc)  */
    }

    public static void main(String[] args) {
        Region reg = new Region ();
        Iterator  it;
        GPS       this_location;
        CAPACITY  this_capacity;

        for (it = (Iterator) reg.regOffer.keySet().iterator(); it.hasNext();) {
            this_location = (GPS) it.next();
            this_capacity = (CAPACITY) reg.regOffer.get(this_location);
            reg.printElem(this_location, this_capacity);
        }
    }
}
```

Se colocarmos outros tipos no `HashMap` mostrado no código em cima, o programa aceita pois tudo deriva de `Object`. Quando os formos buscar e fizermos o *cast* para os tipos que estamos à espera (*GPS* ou *CAPACITY*) temos uma exceção em *run-time*. É desagradável.

O que se deve fazer é usar um mecanismo chamado de `type parameters`, que parametrizam os tipos do `HashMap`, iteradores, etc., e que permitem a verificação durante a compilação, evitando erros em *run-time*. Usando este mecanismo, escusa-se de fazer *cast*. O código anterior ficaria então assim (repare nas declarações do `HashMap` e iterador, e na ausência de *casts*):

```
public class Region {
    HashMap <GPS, CAPACITY> regOffer = new HashMap <GPS, CAPACITY> ();
    Region () {     /*    something   */     }
    }

    void printElem(GPS loc, CAPACITY cap ) {
         /*   prints the key GPS and element (rooms, conference rooms, etc)  */
    }

    public static void main(String[] args) {
        Region reg = new Region ();
        Iterator <GPS> it;
        GPS            this_location;
        CAPACITY       this_capacity;

        for (it = reg.regOffer.keySet().iterator(); it.hasNext();) {
            this_location = it.next();
            this_capacity =  reg.regOffer.get(this_location);
            reg.printElem(this_location, this_capacity);
        }
    }
}
```

Atenção que os dois códigos anteriores não funcionam pois como as classes usadas são novas, o Java não sabe fazer algumas funções. O problema existe para a classe para a qual se decidiu ter o iterador pois usa os métodos `hashCode()` e `equals()`. Para a classe em que decidimos não ter o iterador não existe problema algum

O `hashCode()` existe em todos os objetos tal como o `equals()` e o `toString()`. Se estivermos a usar uma classe nova e não fornecermos estes métodos, são usados os métodos de `Object` que usam o endereço para fazer o *hash* e comparam os endereços para fazer a igualdade. Ora isto provoca problemas com o iterador. Outro método que também tem de ser fornecido é o `entrySet()` da classe `Map` que é usado pelo iterador para fazer o conjunto. O modo como se fornecem métodos está explicado na secção seguinte: "Reusar Objetos".

Assim, tem de se ter em atenção que classe se escolhe para a chave. A classe `String` já existe e é segura. As classes envolventes também são seguras, mas não se esqueça que mal atribua um valor a um objeto dessas classes, não o pode mudar… Isto pode não ser um problema…

EXERCÍCIO 4: Acrescente mais um dado à classe *HelloWorld* que é uma tabela `HashMap` com uma sequência de signos chineses e respetivas horas de regência, como está mostrado na tabela em baixo. Neste caso, tanto as chaves como os elementos são `String`. A tabela é privada.

O construtor de *HelloWorld* constrói a tabela.

Acrescente um método para escrever a tabela toda e outro para escrever as horas (valor) de um certo signo (chave) que é indicado como parâmetro.

O programa, antes de terminar, manda escrever as horas da SERPENTE e da CABRA, e depois manda escrever a tabela toda.

Solução na secção seguinte, como é habitual.

Signo	Período de 2 horas correspondente à regência deste signo
RATO	23:00 à 1:00
BOI	1:00 às 3:00
TIGRE	3:00 às 5:00
COELHO	5:00 às 7:00
DRAGAO	7:00 às 9:00
SERPENTE	9:00 às 11:00
CAVALO	11:00 às 13:00
CABRA	13:00 às 15:00
MACACO	15:00 às 17:00
GALO	17:00 às 19:00
CÃO	19:00 às 21:00
PORCO	21:00 às 23:00

2.8.3. Fazer vetores de tamanho variável com listas: *ArrayList*

A classe `ArrayList` é uma lista que permite executar vetores de tamanho variável – "um vetor que se expande a si mesmo". O seu uso é muito simples: cria-se a lista, colocam-se elementos com `add()` e retiram-se elementos com `get(i)`, usando um índice (tal como num vetor, mas não usando parêntesis retos).

Para a `ArrayList` funcionar com qualquer classe, ela funciona com a classe base de todas, a classe `Object`. Assim, quando se retira um elemento tem de se fazer um *cast* para a classe real. Por exemplo, se tivermos um `ArrayList` de `pandas`, tem de se fazer `((pandas) pand.get(i))` (Considerando que a lista é o objeto `pand`). A exceção a este *cast* acontece para a classe `String` (o compilador chama automaticamente o método `toString()` sempre que quer um `String`).

Também pode utilizar os `type parameters` apresentados em cimas…

A seguinte tabela tem alguns dos métodos da classe `ArrayList`:

Método	Descrição
add`(Object o)`	Adicionar objeto ao fim da lista
add`(int i, Object o)`	Adicionar objeto na posição *i* da lista
clear`()`	Remove todos os elementos da lista
get`(int i)`	Retorna referência para objeto na posição *i*
remove`(int i)`	Elimina objeto na posição *i*
Object [] toArray`()`	Retorna um *array* com todos os objetos da lista

2.9. Reusar objetos

Uma das grandes vantagens de linguagens orientadas a objetos é poder-se reusar objetos. Esta secção descreve os conceitos mais importantes de três mecanismos: composição e herança de classes; polimorfismo; e interfaces.

2.9.1. Composição e Herança

Existem dois modos de usar classes já existentes:

(a) Composição – fazer uma nova classe, onde se criam objetos de outras classes já existentes. "Normalmente quer-se usar certas funcionalidades dos objetos, mas ter uma interface diferente". Por exemplo, um "carro" tem um "motor", quatro "portas", etc. A relação que traduz a composição é "*has-a*".

(b) Herança (*inheritance*) – Cria uma nova classe como um tipo de uma nova classe: "Usa a forma existente e adiciona código (uma funcionalidade, uma especialização) sem modificar a classe existente". Por exemplo, um "jipe" é um "carro" pois tem tudo o que um "carro" tem mais os métodos de andar fora de estrada. A relação que traduz a herança é a "*is-a*".

A composição é bastante trivial: basta criar objetos como membros na nova classe e usá-los.

No caso da herança, existe a palavra-chave `extends`, que indica que esta classe herda de outra classe. O exemplo abaixo em que a classe *Motorbike* herda da classe *Bicycle* mostra as facilidades mais importantes:

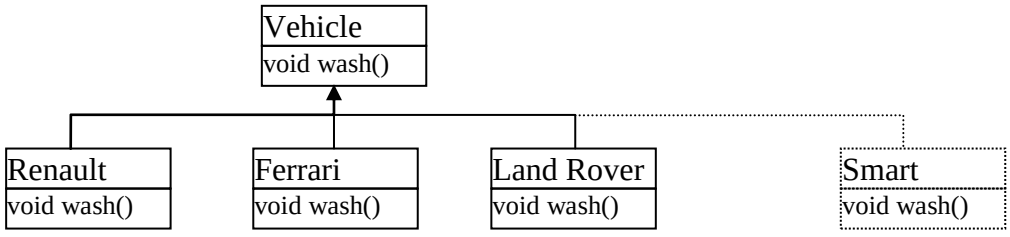
- Público ou privado: Como regra deve-se ter na classe base os dados como `private` e os métodos como `public`.
- `main`: Notar que existem dois `main`. O que é chamado é o da classe que se invoca, mas assim escusa-se de retirar o `main` da classe base (que até pode continuar a ser usada).
- Métodos: Repare que a classe *Motorbike* tem os seguintes métodos: `append()`; `colour()`; `wheel()`; `print()`; `horsepower()`.
- *Override*: O método `wheel()` foi modificado. Simplesmente escreveu-se outra vez esse método. Notar que se quis invocar o método `wheel()` de *Bicycle*. Para isso teve de se usar a expressão `super.wheel()`.
  É deste modo que se criariam novos métodos de `hashCode()`, `equals()` e `toString()` para novas classes.
- Ficheiro: Repare que os `main`, chamam simplesmente os seus métodos. Corra o programa para ver o que ele escreve no ecrã. Como se deve chamar o ficheiro? *Bicycle* ou *Motorbike*?
- Construtor: Repare que *Motorbike* não tem um construtor. É usado o de *Bicycle*. Se tivesse, seria chamado primeiro o construtor de *Bicycle* e depois o de *Motorbike*. Se os construtores tiverem argumentos (por exemplo um `int`), têm de ser chamados explicitamente com a instrução `super` (por exemplo `super (2)`) como primeira instrução no construtor de uma subclasse.
- Invocar uma função `static`: Repare que no `main` de *Motorbike* foi chamado o `main` de *Bicycle* com o nome da classe, em vez de se usar o `super`. Podia-se usar o `super`? Porque é que não?

```
class Bicycle {
  private String s = new String("Bicycle ");
  public void append(String a) { s += a; }
  public void colour() { append(" colour()"); }
  public void wheel() { append(" wheel()"); }
  public void print() { System.out.println(s); }
  public static void main(String[] args) {
    Bicycle bic = new Bicycle ();
    bic.colour();
    bic.wheel();
    bic.print();
  }
}

public class Motorbike extends Bicycle {
  // Change a method:
  public void wheel() {
    append(" Motorbike.wheel()");
    super.wheel(); // Call base-class version
  }
  // Add methods to the interface:
  public void horsePower() { append(" horsePower()"); }
  // Test the new class:
  public static void main(String[] args) {
    Motorbike mtbk = new Motorbike();
    mtbk.colour();
    mtbk.wheel();
    mtbk.horsePower();
    mtbk.print();
    System.out.println("Testing base class:");
    Bicycle.main(args);
  }
}
```

2.9.2. Polimorfismo

Polimorfismo significa um conjunto de conceitos que giram à volta de adaptações a realidades dinâmicas. Considere a imagem seguinte em que a classe base chamada de *Vehicle* tem um método `wash()` que tem as instruções para lavar um carro. Considere três classes mais específicas que têm procedimentos mais eficientes (por exemplo tirando partido da forma de cada marca). Estas classes herdam da classe *Vehicle*, e re-escrevem o método `wash()` (*override*).



Considere o seguinte pseudo código que pede o próximo carro e lava-o.

```
  public static void main(String[] args) {
    Vehicle   car;
    car = nextCar ();
    car.wash();
  }
```

Repare que `nextCar()` devolve o próximo carro da classe *Vehicle*, pois não se sabe de que marca exatamente ele é. O código foi propositadamente escrito deste modo muito embora nos interessasse que o método mais eficiente (da marca) fosse invocado, em vez do mais genérico da classe *Vehicle*.

É isso que acontece mesmo. Devido ao polimorfismo do Java, dependendo da classe específica do próximo carro, o método `wash()` respetivo é chamado. Isto é, se o próximo carro for um Ferrari, o método `wash()` da classe Ferrari é chamado em vez de ser o método `wash()` da classe *Vehicle*.

Esta faceta do polimorfismo é bastante boa pois o `main` pode trabalhar com a classe *Vehicle* e mesmo assim o método respetivo é invocado.

Outra faceta do polimorfismo é se novas classes são acrescentadas (representada na figura com a inclusão da classe *Smart*). Devido ao polimorfismo, a parte do programa que não se mudou não tem de ser mudada, e começa a trabalhar com a nova classe.

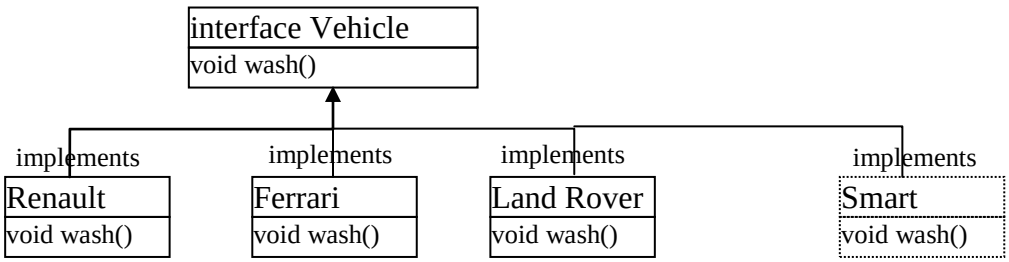
Notar que o método `wash()` numa subclasse tem sempre a mesma assinatura (tipos na lista de argumentos e tipo de retorno) do método da classe base. O método da subclasse toma o lugar do método `wash()` da classe base (a isto chama-se *override*).

Outra coisa seria o método ter o mesmo nome mas uma assinatura diferente. A isto chama-se *overload*, e o Java permite isso. O que acontece é que é simplesmente um novo método, que apenas tem o mesmo nome que o outro. Não o substitui e pode gerar muita confusão.

Repare que polimorfismo só funciona para os métodos das subclasses que também existem na classe base. Isto é, se a subclasse tem mais métodos (especialização) estes métodos não estão contemplados no polimorfismo, como é evidente.

2.9.3 Interfaces e *inner classes*

Quando se usa a palavra-chave `interface` em lugar da palavra-chave `class` isso produz uma classe que tem apenas as assinaturas dos métodos, e não os seus corpos (ou código). Se tiver dados, eles têm de ser `static` e `final`. É como dizer: "Isto é como se parece quaisquer classes que *implementam* esta interface". Aproveitando o tema (e só o tema) do exemplo acima, a classe *Vehicle* poderia ser apenas uma interface sem o código para o método `wash()`.

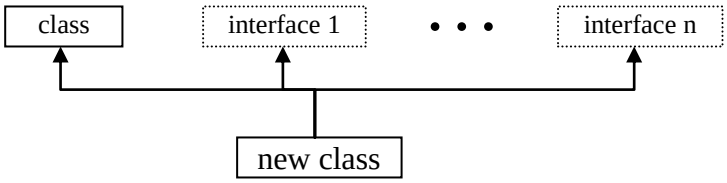


A classe que implementa é uma classe normal que tem todas as características de qualquer outra classe de Java: pode ser base de outras, etc. O código seguinte mostra a sintaxe usada para as interfaces e implementações.

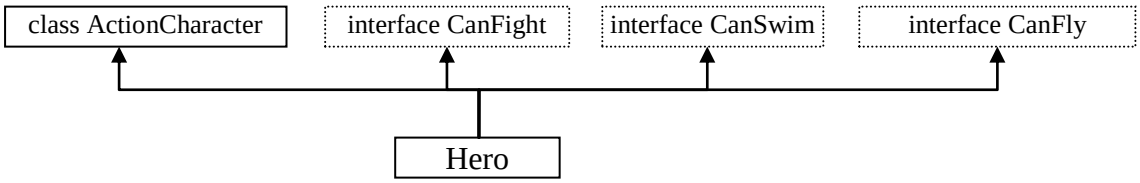
```
interface Instrument {
    // Compile-time constant:
    int i = 5; // static & final
    // Cannot have method definitions:
    void play(); // Automatically public
    String what();
    void adjust();
}

class Wind implements Instrument {
    public void play() {
      System.out.println("Wind.play()");
    }
    public String what() { return "Wind"; }
    public void adjust() {}
}
```

Um uso muito importante é poder definir uma classe que herda de outra classe e que implementa "*n*" interfaces. Esta classe pode não herdar nenhuma classe normal e apenas implementar "n" interfaces, mas se herdar classes normais só pode herdar uma.



O código seguinte tem o tema de super-heróis. Repare que a classe *Hero* implementa três interfaces e herda de outra classe.



A classe *Adventure* tem quatro métodos em que cada um recebe como argumento um objeto de uma classe diferente. O programa começa por criar um herói "*h*" e chamar cada um dos métodos de *Adventure*, em que o herói é tratado como a respetiva classe base (seja ela interface ou classe).

Repare no método `fight()` na classe *Hero*. Não está lá pois é herdado de *ActionCharacter*.

Este exemplo dos super-heróis mostra a grande vantagem do uso de interfaces: poder-se fazer o *upcast* de um objeto para mais do que uma classe.

```
import java.util.*;

interface CanFight {
  void fight();
}
interface CanSwim {
  void swim();
}
interface CanFly {
  void fly();
}
class ActionCharacter {
  public void fight() {  /* something  */  }
}

class Hero extends ActionCharacter
    implements CanFight, CanSwim, CanFly {
  public void swim() {  /* something  */  }
  public void fly() {  /* something  */  }
}

public class Adventure {
  static void t(CanFight x) { x.fight(); }
  static void u(CanSwim x) { x.swim(); }
  static void v(CanFly x) { x.fly(); }
  static void w(ActionCharacter x) { x.fight(); }
  public static void main(String[] args) {
    Hero h = new Hero();
    t(h); // Treat it as a CanFight
    u(h); // Treat it as a CanSwim
    v(h); // Treat it as a CanFly
    w(h); // Treat it as an ActionCharacter
  }
}
```

"*Inner classes*" são classes definidas dentro de outras classes. Existem razões para o seu uso como o acesso a dados e métodos, ou a possibilidade de fazer "*upcasting*" na cadeia de herança para uma classe que não é possível ser acedida de outro modo. Este assunto não vai ser mais explorado neste documento.

2.10. Temporizadores

A biblioteca de classes da linguagem Java inclui várias classes que podem funcionar como temporizadores. O funcionamento é simples (mas a sintaxe de escrita é um pouco complicada): o temporizador é ativado com um certo intervalo e quando o tempo expirar é chamado um método, que se designa normalmente por *callback*.

Uma das classes que tem uma interface mais simples é a classe `javax.swing.Timer`. Para usar esta classe, têm de se importar as classes debaixo da diretoria `javax.swing` e `java.awt.event` utilizando o seguinte código:

```
import javax.swing.*;
import java.awt.event.*;
```

O modo como se define o método de *callback* é que ficou complicado. O modo consiste em criar um objeto de uma classe chamada `java.awt.event.ActionListener` (ouvidor de

ações) e definir no ato da sua criação o método de *callback* (com o código). O método de *callback* tem de ter o nome de `actionPerformed`.

Os procedimentos são os seguintes.

Primeiro tem de se criar uma referência para um objeto `Timer`.

```
javax.swing.Timer timer;
```

O código em baixo mostra os procedimentos para se definir o método de *callback* e a criação do objeto da classe `Timer` que ficou com um tempo inicial igual ao valor de um inteiro com o nome de *period*. Este código pode ficar dentro do construtor do objeto principal do programa.

```
// Define the timer's callback function and creates timer object
  java.awt.event.ActionListener act;
  act = new java.awt.event.ActionListener() { // define função corrida
      public void actionPerformed(java.awt.event.ActionEvent evt) {
        // Código executado quando o temporizador disparar
      }
    };
  timer = new javax.swing.Timer(period /*ms*/, act);// Cria objeto timer
```

Pode-se depois estabelecer um tempo para o temporizador e lançá-lo (por exemplo no `main`):

```
timer.setDelay (3000);          // Set a time of 3 seconds
timer.start();                  // Launches the timer
```

O temporizador pode ser interrompido usando o método `stop()`.

```
timer.stop();
```

A classe `javax.swing.Timer` oferece outros métodos que permitem:

- reiniciar o temporizador (método `restart()`),
- definir se corre apenas uma vez ou se funciona continuamente (método `setRepeats()`). **Atenção que por omissão o valor `setRepeats` está a `true`.**
- modificar o tempo de espera (`setDelay`),
- etc.

EXERCÍCIO 5: Como o código já vai bem grande, use o código do exercício 1. A classe *HelloWorld* tinha o inteiro e a `String` e vai ter mais uma referência para a classe `Timer`.

No construtor faça a definição da *callback* (que escreve qualquer coisa como "*Timer expired*") e crie o objeto temporizador com um valor de tempo qualquer. No `main` estabeleça o valor de 3 segundos para o temporizador, arranque-o e acabe o programa.

5.1: Vai ver que o programa acaba e o temporizador não chegou a expirar. Acabou com o programa. Como o temporizador estava dentro do objeto e ele acabou, acabou tudo.

5.2: Não deixe o programa acabar. Para isso faça com que ele adormeça por um período muito grande. Coloque a instrução `Thread.sleep(200000)` no `main`. Agora deve acontecer que a cada três segundos é escrita a mensagem "*Timer expired*".

5.3: Mude a função *callback* para que ela acabe com o programa, com a instrução `System.exit (1);`

Solução na secção seguinte como é habitual.

2.11. Entradas e Saídas (I/O): Consola, Ficheiros, Rede

As operações de entrada e saída seguiram a filosofia do C e Unix em ter um conceito que fosse geral para todos os casos. Basicamente pretende-se comunicar com ficheiros, a consola, e a rede. Para complicar, pretende-se ainda ter a possibilidade de usar modos diferentes: sequencial, acesso aleatório, *buffered*, binário, caracter, por linhas, por palavras, comprimido, etc.

O Java usa também o conceito de *stream*, e as classes estão divididas em entradas e saídas.

- Entradas – existem as classes base `InputStream` (`Reader`) que definem o método `read()` para ler um byte (caracter) ou um vetor de bytes (caracteres).
- Saídas – existem as classes base `OutputStream` (`Writer`) que definem o método `write()` para escrever um byte (caracter) ou um vetor de bytes (caracteres).

No entanto, estas classes não se usam diretamente. Definem-se subclasses, ou envolvem-se as classes noutras de modo a fornecer interfaces mais úteis. Os procedimentos acabam por ser mais ou menos complicados em virtude do número de classes envolvidas.

É útil pensar que no Java existem duas raízes para as classes dos *streams*:

- a raiz orientada ao byte com o `InputStream` e
- a raiz orientada ao caracter (16 bits) com o `Reader`.

No caso das saídas são o `OutputStream` e `Writer`.

> Ao se envolver um objeto da classe `InputStream` com a classe `InputStreamReader`, os bytes são passados para caracteres (o `InputStream` é convertido num `Reader`). Isto é, o método `read()` do `InputStreamReader` lê um caracter.
> O mesmo acontece para as saídas com `OutputStream`, `Writer`, e `OutputStreamWriter`.
> Este assunto é explicado a seguir.

`InputStream/OutputStream`

Para o caso do `InputStream` existem as seguintes subclasses que se especializaram em alternativas para a origem dos dados (memória, objeto `String`, ficheiro, *pipes*, múltiplos `InputStreams`, ou outros como a Internet) e na possibilidade de ter *decorators*.

O Java usa a palavra "*decorator*" para significar o envolver classes noutras para lhes dar mais funcionalidades.

	Class	Function
1	`ByteArrayInputStream`	Allows a buffer in memory to be used as an `InputStream`
2	`StringBufferInputStream`	Converts a `String` into an `InputStream`
3	`FileInputStream`	For reading information from a file.
4	`PipedInputStream`	Produces the data that's being written to the associated `PipedOutputStream`. Implements the "piping" concept.
5	`SequenceInputStream`	Converts two or more `InputStream` objects into a single `InputStream`.
6	`FilterInputStream`	Is an interface for decorators that provide useful functionality to the other `InputStream` classes

As cinco primeiras linhas representam que se pode ter um `InputStream` a partir de um *buffer* de memória, de um `String`, de um ficheiro, etc., do mesmo modo que se tem um `InputStream` do teclado, por exemplo. Isto é, ao criar um objeto de uma dessas classes fica-

se com os métodos do `InputStream` para ler do *buffer* de memória, do `String`, de um ficheiro, etc.

A subclasse 6 é apenas uma interface que permite ter os *decorators*. Com ela, pode-se colocar um *decorator* sobre um `InputStream` e ler-se do teclado de um certo modo, ou sobre um `ByteArrayInputStream` e ler-se de um *buffer* de memória desse mesmo modo, etc. Lembre-se que se pode herdar simultaneamente de uma classe normal e de uma interface…

Para o `OutputStream` existem subclasses equivalentes (menos o 2 e o 5).

Os *decorators* possíveis são:

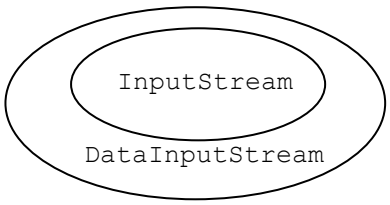
Class	Function
`DataInputStream`	Used in concert with `DataOutputStream`, so you can read primitives (`int`, `char`, `long`, etc.) from a stream in a portable fashion.
`BufferedInputStream`	Use this to prevent a physical read every time you want more data. You're saying "Use a buffer."
`LineNumberInputStream`	Keeps track of line numbers in the input stream; you can call `getLineNumber()` and `setLineNumber(int)`.
`PushbackInputStream`	Has a one byte push-back buffer so that you can push back the last character read.

O `DataInputStream` permite fazer duas coisas importantes:

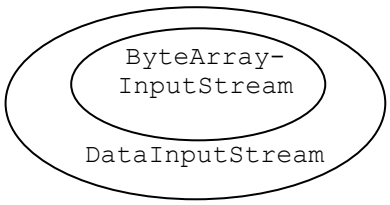
1. Permite ler diretamente tipos primitivos e `String`. Todos os métodos começam com "`read`" tal como `readByte()`, `readFloat()`, etc.
2. Com o seu "companheiro" `DataOutputStream` permite mover tipos primitivos de um sítio para outro via um *stream*. Ver mais adiante.

As outras classes modificam o modo como o `InputStream` se comporta: se tem uma fila (*buffered*); se tem registo das linhas que lê, ou se permite que se volte a recolocar um caracter já lido.

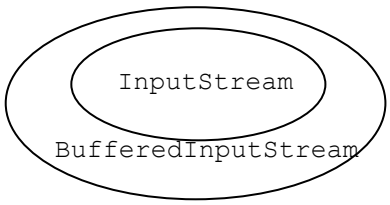
A figura seguinte mostra três casos. No primeiro, ficam definidos métodos simples para ler tipos primitivos (`int`, `short`, etc.) e `String`. O segundo é equivalente mas o objeto foi criado sobre um *buffer* de memória e portanto está-se a ler `int`, `short`, etc. a partir de um *buffer* de bytes. No terceiro, foi criado um *buffer* onde são guardados os bytes lidos e o programa lê desse *buffer*, em vez de efetuar uma leitura física.



Tem métodos para ler `int`, `short`, `float`, etc.



Tem métodos para ler `int`, `short`, `float`, etc.



Usa um *buffer*. Cada leitura não é uma leitura física.

Em termos de saída existem os seguintes *decorators*:

Class	Function
`DataOutputStream`	Used in concert with `DataInputStream` so you can write primitives (`int`, `char`, `long`, etc.) to a stream in a portable fashion.
`PrintStream`	For producing formatted output. While `DataOutputStream` handles the storage of data, `PrintStream` handles display.
`BufferedOutputStream`	Use this to prevent a physical write every time you send a piece of data. You're saying "Use a buffer." You can call `flush()` to flush the buffer.

O `DataOutputStream` tem métodos simples para escrever tipos primitivos ou `String` no `OutputStream`. Todos os métodos começam com "`write`", tal como `writeByte()`, `writeFloat()`, etc.

Um aspeto interessante do `DataInputStream` e do `DataOutputStream` é que o modo como eles fazem a conversão é independente do modo como os inteiros, os float, etc., estão representados nas várias máquinas. Assim um inteiro escrito numa máquina Windows por `DataOutputStream` é lido numa máquina Linux por `DataInputStream` e o valor é o mesmo. Se se colocasse bit a bit e o formato dos inteiros fosse diferente, o valor do inteiro ficará diferente, como é óbvio.

O propósito original do `PrintStream` é o de escrever os tipos primitivos e o `String` num formato legível. Tente perceber como isto é diferente do propósito do `DataOutputStream`. A classe `PrintStream` tem os métodos `print()` e `println()`[1].

O `BufferedOutputStream` usa uma fila de espera para evitar fazer uma escrita física cada vez que é chamado.

Reader/Writer

Basicamente, as classes `Reader` e `Writer`, orientadas ao caracter, são as classes a usar em programação dita "normal" e vão-se descrever de seguida. As classes orientadas ao byte têm utilidade quando temos operações de compressão de dados ou quando se transmite dados pela rede, com no caso desta disciplina.

As subclasses seguintes são as equivalentes às subclasses dos `InputStream` e `OutputStream` (tente fazer a relação do que foi explicado para estes casos):

`Reader` adapter: `InputStreamReader`	`Writer` adapter: `OutputStreamWriter`
`FileReader`	`FileWriter`
`StringReader`	`StringWriter`
`CharArrayReader`	`CharArrayWriter`
`PipedReader`	`PipedWriter`

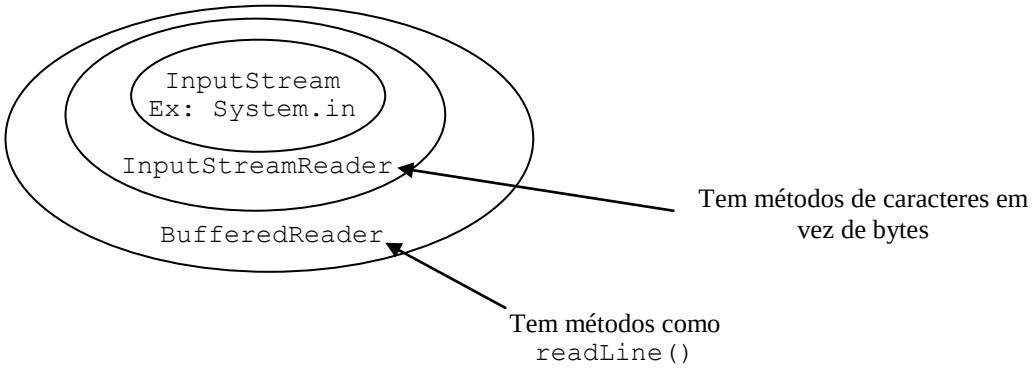
A lista seguinte é parcial e é a correspondente aos *decorators*, mas agora para o `Reader` e o `Writer`. Do ponto de vista de organização de classes as coisas não são bem iguais, mas este pormenor sai fora do âmbito desta descrição.

[1] O `PrintStream` pode ser problemático pois apanha todas as exceções `IOExceptions` (tem de se testar o estado do erro com `checkError()`, que retorna `true` se ocorreu um erro). Para além disso o `PrintStream` não é muito internacional (usa bytes em vez de caracteres) e não trata os *line breaks* de um modo independente das diferentes máquinas. Estes problemas foram resolvidos com `PrintWriter`.

`BufferedReader` (also has `readLine()`)	`BufferedWriter`
	`PrintWriter`
`LineNumberReader`	
`PushBackReader`	

À laia de receita, quando se quiser usar `readLine()` deve-se usar o `BufferedReader`. O `PrintWriter` tem a opção de fazer o esvaziamento do buffer sempre que se usa `println()`.

A figura abaixo mostra como se parte do `System.in` que é um `InputStream` (orientado ao byte) e se consegue ler uma linha de caracteres.



Acha que conseguia imaginar as classes envolvidas para ler uma linha de caracteres a partir de um *buffer* de bytes em memória?

2.11.1. Consola

Leitura

Do que já foi abordado a descrição é óbvia.

O *stream* `System.in` é um objeto da classe `InputStream`. Primeiro deve-se envolver com o `InputStreamReader` para termos caracteres e depois com o `BufferedReader` para termos o *buffer*, e evitar fazer leituras físicas.

Em baixo está mostrado uma parte de código para se ler uma linha de caracteres (`String`):

```
String s = null;
BufferedReader bufferRead = new BufferedReader(new InputStreamReader(System.in));
try {
    s = bufferRead.readLine();
} catch (IOException ex) {
    Logger.getLogger(HelloWorld.class.getName()).log(Level.SEVERE, null, ex);
}
```

Para se ler um `double` pode-se usar o código seguinte. Tente perceber o que se fez.

```
public static void main(String[] args) {
    String s;
    double numberFromConsole;
    DecimalFormat df;
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

    System.out.println("Enter a number.");
    try {
        s = br.readLine();
        numberFromConsole= Double.parseDouble(s);
    } catch (IOException e) {
        numberFromConsole = 0;
    } catch (NumberFormatException e) {
        numberFromConsole = 0;
    }
}
```

Escrita

Tanto o *stream* `system.out` como o `System.err` já estão envolvidos como `PrintStream` e como tal já têm métodos úteis como o prova este documento desde o início com métodos como `System.out.println()`. O que se pode ainda fazer é envolvê-los com `PrintWriter`, como mostra o código

```
PrintWriter out = new PrintWriter(System.out, true);
    out.println("Hello, world");
```

Deve-se usar o construtor com dois argumentos e o segundo deve ser `true` para fazer o esvaziamento do buffer sempre que se chamar o `println()`.

2.11.2. Ficheiros

A classe `File` representa um ficheiro ou uma diretoria, e define um conjunto de métodos para os manipular. O construtor recebe o nome completo de um ficheiro. Os métodos da classe permitem: saber se o ficheiro existe; se é ficheiro ou diretoria; o comprimento do ficheiro; apagar o ficheiro; ou marcar o ficheiro para ser apagado quando o programa terminar. Inclui ainda métodos para criar ficheiros temporários com nomes únicos.

```
File f= new File ("/home/pc40/xpto.txt");   // Associa-se a ficheiro
long len= f.length();                       // Comprimento do ficheiro
if (f.exists()) …                           // Se existe
if (f.isFile()) …                           // Se é ficheiro
if (f.canRead()) …                          // Se é legível
f.delete();                                 // Apaga ficheiro

File temp= File.createTempFile("proxy", ".tmp"); // Cria ficheiro temporário com
                                                 //  nome único
temp.deleteOnExit();                        // Apaga ficheiro quando a aplicação
                                            //termina
File.separator                              // '/' ou '\\' dependendo do sistema
                                            //operativo
```

A leitura de ficheiros de texto é geralmente realizada através da classe `FileInputStream`.

```
FileInputStream f= new FileInputStream ( file );
```

Recomenda-se que a escrita de ficheiros de texto seja feita através da classe `BufferedWriter`, por esta permitir definir o conjunto de carateres ("ISO-8859-1" por omissão). Como sabe, o conjunto "ISO-8859-1" é um dos da série de conjuntos ASCII, em que os últimos caracteres (menos de uma centena) são tais que suportam integralmente as línguas ocidentais incluindo a Portuguesa. Caso o mesmo tipo de carateres seja usado nos canais associados a *sockets* (canais de comunicação através da rede) e a ficheiros, o Java nunca faz conversão de tipos, permitindo transmitir dados arbitrários (imagens, aplicações, etc.).

```
FileOutputStream fos = new FileOutputStream(file);
OutputStreamWriter osr= new OutputStreamWriter(fos, "8859_1");
BufferedWriter os= new BufferedWriter(osr);

// Permitem ler e escrever 'char []' com os métodos 'read' e 'write'
// usando o método 'getBytes()' é possível converter um 'char []' em 'byte []'
```

2.11.3. Rede

O uso de entradas e saídas para a rede vai ser o tema forte de Sistemas de Telecomunicações, pelo que não se adianta nada nesta secção.

2.12. Programação multi-tarefa

O Java, ao contrário do C, suporta de raiz o paralelismo entre tarefas (*threads*). Em Java é muitas vezes necessário lançar vários objetos a correr em paralelo para receber eventos de várias fontes. Esta funcionalidade é vulgarmente realizada criando classes que estendem a classe `Thread` (isto é, que são subclasses da classe `Thread`), e que têm a função `run`, que é corrida pelo sistema.

O código seguinte exemplifica a declaração de uma classe que realiza uma tarefa. A classe deve incluir um construtor, que inicia todas as variáveis locais da classe, e o método `run`. O método `run` é vulgarmente um ciclo controlado por uma variável de controlo (no caso `keepRunning`), onde se invoca uma operação bloqueante (e.g. leitura de dados de um *socket*).

Embora a classe `Thread` disponibilize um método `stop`, este não deve ser usado para parar uma tarefa pois foi classificado na versão de Java 1.4 como DEPRECATED, isto é, desatualizado. Em sua substituição, deve ser criada uma função na classe (e.g. `stopRunning`), que modifique a variável de controlo do ciclo principal da tarefa.

```
public class Daemon extends Thread {

    volatile boolean keepRunning= true;
    // Parametros adicionais …

    public Daemon(/*argumentos*/) {       /** Creates a new instance of Daemon */
        // Inicialização de parametros a partir de argumentos
    }

    public void run() {                                    /** Runned function */
        // Inicializações antes do ciclo
        while (keepRunning) {
             // Código do ciclo
             this.yield();     // Passar controlo para outra thread
        }
    }

    public void stopRunning() {               /** Stops thread running safely */
        keepRunning= false;
```

```
    }
}
```

Uma tarefa pode ser criada e arrancada em qualquer função utilizando o seguinte excerto de código:

```
Daemon daemon= new Daemon(/*argumentos*/);              // Cria objeto thread
daemon.start();                                        // Arranca a tarefa
```

O facto de existirem várias tarefas a correr em paralelo pode levantar alguns problemas de sincronismo no acesso a objetos partilhados por diversas tarefas. Por exemplo, várias caixas de texto da interface gráfica, ou elementos de um vetor. O problema é que uma tarefa pode estar a ler ou a escrever e antes de finalizar todo o procedimento outra tarefa modifica algo. Este é um problema complexo que os alunos estudarão no terceiro ano do curso. Uma solução simples para Sistemas de Telecomunicações reside em usar o mecanismo da linguagem Java para garantir que o acesso a uma função ou objeto só é realizado por uma tarefa de cada vez. O método mais simples é a classificação de métodos de classes com a palavra-chave `synchronized`, que bloqueia todos os métodos do objeto enquanto uma tarefa estiver a usá-los.

2.13. Bibliografia adicional

Este documento resume uma pequena parte das especificações da linguagem Java e do ambiente de desenvolvimento NetBeans, necessárias para a realização do trabalho prático. Caso necessite de mais informação do que a fornecida por este documento recomenda-se a consulta de:

"Thinking in Java Second Edition", de Bruce Eckel, 2001, Prentice Hall. Disponível na web (http://www.mindview.net/Books). GRATIS

"Thinking in Java Third Edition", de Bruce Eckel, 2002. Disponível na web (http://www.mindview.net/Books). GRATIS

"Thinking in Java Forth Edition", de Bruce Eckel, Prentice Hall, 2006, ISBN: 0131872486.

Documentação sobre o JDK e NetBeans disponível na web (http://java.sun.com/).

"Java Cookbook, Second Edition", de Ian F. Darwin, O'Reilly & Associates, Inc., 2004, ISBN: 0596007019.

"Java Network Programming, Third Edition", de Elliotte R. Harold, O'Reilly & Associates, Inc., 2004, ISBN: 0596007213.

"Java in a Nutshell, 5th Edition", de David Flanagen, O'Reilly & Associates, Inc., 2005, ISBN: 0596007736.

3. SOLUÇÕES DOS EXERCÍCIOS

O código do exercício 1 é o seguinte. Acertou? Repare no que aconteceu às duas variáveis que agora são atributos do objeto. Nesta solução a *string* `str` fica a apontar para onde `str1` está a apontar. Não se criou memória nova com `new`. Podia-se tê-lo feito…

```
package helloworld;

/**
 *
 * @author paulopinto
 */
public class HelloWorld {

    public int n;
    public String str;

    /**
     * @param args the command line arguments
     */
    // Construtor
    HelloWorld(int n1, String str1) {
        n = n1;          // ou this.n = n1;
        str = str1;      // ou this.str = str1;
    }

    public static void main(String[] args) {
        HelloWorld ob = new HelloWorld(0, "2013");
        try {
            ob.n = Integer.parseInt(ob.str);
        } catch (NumberFormatException e) {
            System.out.println("Invalid Number" + e);
        }
// TODO code application logic here
        System.out.println("Hello World " + ob.n);
    }
}
```

O código do exercício 2 encontra-se aqui

```
package helloworld;

import java.text.SimpleDateFormat;
import java.util.Date;

/**
 *
 * @author paulopinto
 */
public class HelloWorld {

    public int n;
    public String str;
    public Date  data_inicio;

    /**
     * @param args the command line arguments
     */
    // Construtor
    HelloWorld(int n1, String str1) {
        n = n1;         // ou this.n = n1;
        str = str1;     // ou this.str = str1;
        data_inicio = new Date ();
    }

    public static void main(String[] args) {
        HelloWorld ob = new HelloWorld(0, "2013");
        try {
            ob.n = Integer.parseInt(ob.str);
        } catch (NumberFormatException e) {
            System.out.println("Ivalid Number" + e);
        }
// TODO code application logic here
        SimpleDateFormat formatter = new SimpleDateFormat("E hh:mm:ss 'em' dd.MM.yyyy");
        System.out.println("Hello World " + ob.n);
        System.out.println("The current date is " + formatter.format(ob.data_inicio));
    }
}
```

Exercício 3 (foram retirados alguns comentários do início para encurtar o tamanho):

```
package helloworld;

import java.text.SimpleDateFormat;
import java.util.Date;

public class HelloWorld {

    public int n;
    public String str;
    public Date  data_inicio;
    private int[] vect;

    // Construtor
    HelloWorld(int n1, String str1) {
        int   i;
        n = n1;          // ou this.n = n1;
        str = str1;      // ou this.str = str1;
        data_inicio = new Date ();
        vect = new int [5];
        for (i=0; i<5; i++) {
            vect [i] = i*2;
        }
    }
    public void print_vect () {
        int i;
        System.out.println("Vetor:");
        for (i=0; i <5; i++) {
            System.out.println("Element " + i + " = " +  vect[i]);
        }
    }
    public static void main(String[] args) {
        HelloWorld ob = new HelloWorld(0, "2013");
        try {
            ob.n = Integer.parseInt(ob.str);
        } catch (NumberFormatException e) {
            System.out.println("Invalid Number" + e);
        }
// TODO code application logic here
        SimpleDateFormat formatter = new SimpleDateFormat("E hh:mm:ss 'em' dd.MM.yyyy");
        System.out.println("Hello World " + ob.n);
        System.out.println("The current date is " + formatter.format(ob.data_inicio));
        ob.print_vect();
    }
}
```

Exercício 4 (foram retirados alguns comentários do início para encurtar o tamanho):

```
package helloworld;

import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.HashMap;
import java.util.Iterator;

public class HelloWorld {

    public int n;
    public String str;
    public Date  data_inicio;
    private int[] vect;
    private String val;
    private HashMap<String,String> h;


    // Construtor
    HelloWorld(int n1, String str1) {
        int   i;
        n = n1;         // ou this.n = n1;
        str = str1;     // ou this.str = str1;
        data_inicio = new Date ();
        vect = new int [5];
        for (i=0; i<5; i++) {
            vect [i] = i*2;
        }
        h = new HashMap<String,String> ();  // Cria um Map vazio
        h.clear();                                                  // Limpa Map
        h.put("RATO", "23:00 à 1:00");
        h.put("BOI", "1:00 às 3:00");
        h.put("TIGRE", "3:00 às 5:00");
        h.put("COELHO", "5:00 às 7:00");
        h.put("DRAGAO", "7:00 às 9:00");
        h.put("SERPENTE", "9:00 às 11:00");
        h.put("CAVALO", "11:00 às 13:00");
        h.put("CABRA", "13:00 às 15:00");
        h.put("MACACO", "15:00 às 17:00");
        h.put("GALO", "17:00 às 19:00");
        h.put("CÃO", "19:00 às 21:00");
        h.put("PORCO", "21:00 às 23:00");
    }
```

```
    public void print_vect () {
        int i;
        System.out.println("Vector:");
        for (i=0; i <5; i++) {
            System.out.println("Element " + i + " = " + vect[i]);
        }
    }

    public void print_element (String val) {
        System.out.println(val + " has regency in hours " + h.get(val));
    }

    public void print_all_elements () {
        Iterator<String> it;
        String           key;
        for (it= h.keySet().iterator(); it.hasNext();) {
            key = it.next();
            System.out.println(key + " has regency in hours " + h.get(key));
        }
    }

    public static void main(String[] args) {
        HelloWorld ob = new HelloWorld(0, "2013");
        try {
            ob.n = Integer.parseInt(ob.str);
        } catch (NumberFormatException e) {
            System.out.println("Invalid Number " + e);
        }
// TODO code application logic here
        SimpleDateFormat formatter = new SimpleDateFormat("E hh:mm:ss 'em' dd.MM.yyyy");
        System.out.println("Hello World " + ob.n);
        System.out.println("The current date is " + formatter.format(ob.data_inicio));
        ob.print_vect();
        ob.print_element("SERPENTE");
        ob.print_element("CABRA");
        ob.print_all_elements();
    }
}
```

Exercício 5 (foram retirados alguns comentários do início para encurtar o tamanho):

```
package helloworld;

import java.util.logging.Level;
import java.util.logging.Logger;

public class HelloWorld {

    public int n;
    public String str;
    public javax.swing.Timer timer;

    // Construtor
    HelloWorld(int n1, String str1) {
        java.awt.event.ActionListener act;

        n = n1;         // ou this.n = n1;
        str = str1;     // ou this.str = str1;
        act = new java.awt.event.ActionListener() { // define callback function
            @Override
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                // Code to be run when timer expires
                System.out.println("Timer expired");
                System.exit (1);
            }
        };
        timer = new javax.swing.Timer(1000 /*ms*/, act);// The period is just dummy
    }

    public static void main(String[] args) {
        HelloWorld ob = new HelloWorld(0, "2013");
        try {
            ob.n = Integer.parseInt(ob.str);
        } catch (NumberFormatException e) {
            System.out.println("Invalid Number " + e);
        }
// TODO code application logic here
        ob.timer.setDelay(3000);
        ob.timer.start();
        System.out.println("Hello World " + ob.n);
        try {
            Thread.sleep(200000);
        } catch (InterruptedException ex) {
            Logger.getLogger(HelloWorld.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}
```

4. UM POUCO DE HISTÓRIA SOBRE PROGRAMAÇÃO PARA REDES TCP/IP

Os sockets (que se poderão traduzir por tomada[2]) foram inventados pela comunidade UNIX para possibilitar a comunicação entre quaisquer processos UNIX correndo na mesma máquina, ou em máquinas diferentes. Na disciplina de Sistemas de Telecomunicações vão-se usar o subgrupo (**domínio**) de sockets que permite a comunicação entre máquinas diferentes. São, assim, uma interface à rede. Houve duas preocupações na sua definição:

- que primitivas definir para não complicar a programação, mantendo o estilo de programação usado nas aplicações não distribuídas; e,
- onde colocar essa interface na pilha de protocolos TCP/IP que se adotou para o UNIX.

A primeira preocupação levou à definição dos *sockets* baseada nos acessos aos ficheiros do sistema UNIX, que é extremamente simples. Basicamente é necessário criar os *sockets*, dar-lhes um número (*handler*), torná-los operacionais e depois poder fazer operações de leitura e de escrita. Quando se lê informação a partir dos *sockets*, recebe-se informação que veio da rede e que está guardada no sistema operativo à espera que a aplicação a queira receber. Quando se escreve no *socket*, envia-se informação para o sistema operativo, que a seu tempo a enviará para a rede. Na linguagem Java, estes *sockets* básicos são usados através das classes da biblioteca `DatagramSocket`, `Socket` e `ServerSocket`.

A segunda preocupação, onde colocar a interface de interação com os *sockets* na pilha de protocolos, podia ter muitas hipóteses. Podia-se fazer deles uma interface ao nível IP (Internet Protocol), ou mesmo mais abaixo. Se tal fosse feito, as aplicações teriam de se preocupar ainda com problemas específicos de comunicação (controlo de fluxo, de erro, encaminhamento, congestão, etc.). Seria muito diferente de um acesso a um ficheiro... A opção foi colocá-los acima do nível Transporte (concretamente, TCP ou UDP). Acima deste nível já só existem preocupações de dados e não de comunicação. Outro ponto importante é que o Transporte deve esconder alterações (evoluções) da tecnologia de rede, e escudar as aplicações dessas alterações. Tal veio a suceder pois os *sockets* mantiveram a sua definição inalterada durante trinta anos, estando agora a sofrer pequenas alterações para possibilitar o uso de aplicações multimédia. Não é demais realçar que trinta anos em computadores é bastante tempo e só prova a clareza e simplicidade que houve no desenho inicial.

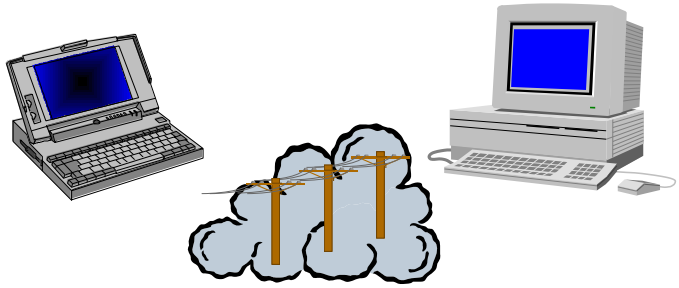
4.1. Tipos de *sockets*

O domínio de *sockets* de rede é designado de domínio Internet, e embora seja muito geral para permitir uma diversidade grande de tipos de *sockets* só têm existido dois tipos, na prática: os *sockets stream* e os *sockets datagram*.

Os *sockets stream* são os mais parecidos com os acessos aos ficheiros. As aplicações escrevem nos *sockets* a informação que querem enviar em feixes de octetos. Isto é, escrevem por exemplo 30 octetos, ou 123, ou 235. Não existe a noção de pacote da rede. O protocolo TCP, dentro do sistema operativo, é que decide se espera por mais octetos para formar um pacote de rede, se divide os dados em vários pacotes, ou se envia um pacote por chamada de escrita. Na leitura o processo é semelhante: a aplicação lê um número de octetos que quer (que faz sentido para ela) – 30, 57, 127, etc. O TCP vai recebendo pacotes e coloca os dados numa fila à espera de serem lidos pela aplicação. Como se vê, é tal e qual como escrever, ou ler, de um ficheiro. Outra caraterística deste tipo de *sockets* é que existe fiabilidade e sequencialidade. Enquanto o *socket* estiver ativo, a aplicação sabe que o que escreveu chega ao outro processo ***exatamente*** nos mesmos modos que escreveu (tudo e pela mesma ordem). Se houver problemas na rede que

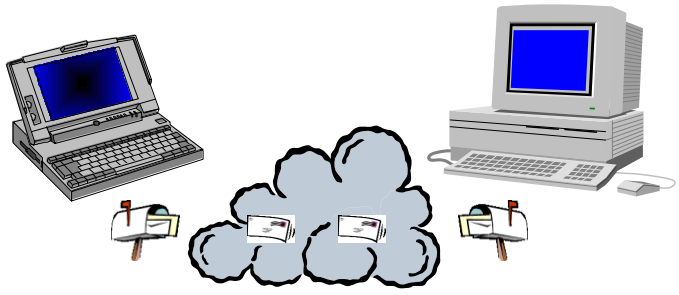
[2] Existiriam depois os *plugs* (fichas) nas aplicações, que se ligariam às tomadas. Com o tempo veio-se a verificar que não houve a necessidade de se definir este conceito de *plug* explicitamente.

o TCP não consiga resolver, a aplicação é informada da destruição do *socket*, e terá de fazer os procedimentos de recuperação necessários para se resincronizar com a aplicação remota novamente. Quando se usam *sockets stream*, ficam estabelecidos dois caminhos entre as aplicações, permitindo comunicação bidirecional. É, portanto, orientado à ligação. O funcionamento é muito parecido com a rede telefónica neste aspeto: estabelece-se uma ligação, troca-se informação com garantia de sequencialidade entre emissor e recetor, e desliga-se a ligação.



Os *sockets datagram* são muito mais simples internamente, e mais complexos de se usar pela aplicação. Na interface de *socket* não existe a noção de feixe (*stream*) como descrito anteriormente e a quantidade de dados que a aplicação escreve corresponde exatamente a um pacote datagrama que é enviado pela rede. Quando se lê a partir de um socket datagrama, recebe-se o número de octetos que o primeiro pacote da fila de espera tem. Pode-se ler um número arbitrário de octetos (ex. 30, 55, etc.). Cada pacote é enviado pela rede sem garantias de fiabilidade. Isto não significa que a rede perca pacotes uns atrás dos outros. Normalmente isso quase nunca acontece. O que não existe é garantia de recuperação e pode acontecer que um pacote pura e simplesmente se perca devido a erros ou congestão e nunca chegue ao destino. Assim, a aplicação tem de ser programada para esta eventualidade. Em vez do TCP, o protocolo de Transporte usado neste tipo de *sockets* é o UDP, que não é orientado à ligação. Neste tipo de *sockets* não faz sentido falar em comunicação bidirecional, pois cada datagrama é independente dos outros, e as aplicações que queiram interagir enviam, simplesmente datagramas umas às outras.

A analogia agora é com o serviço postal. Neste serviço são enviadas cartas, cada uma com o endereço completo e sem qualquer referência a outras cartas que foram enviadas anteriormente ou que serão enviadas posteriormente.



4.2. Identificação de *sockets*

Os *sockets* são um ponto de acesso de serviço (SAP – *Service Access Point*) entre a aplicação e o nível Transporte. Necessitam de uma identificação clara para o sistema operativo poder colocar as mensagens que vêm pela rede na fila de espera correspondente. A sua identificação consiste na concatenação do endereço de rede da máquina com um identificador de Transporte. O endereço de rede das máquinas é o endereço IPv4 – 32 bits, em que os primeiros designam a rede e sub-rede onde a máquina se encontra e os últimos, o número da máquina

nessa rede e sub-rede. Os endereços IPv4 são escritos normalmente representando cada octeto em decimal (0 a 255) separado por pontos. Por exemplo **136.124.0.23**

Por exemplo, o endereço do computador número um na rede do laboratório de telecomunicações é '172.16.54.1' ('172.16.54.' define a rede local e o último octeto '1' define o número da máquina).

Existe um conjunto de endereços especiais reservados. Por exemplo, o endereço '127.xx.yy.zz' (loopback) permite realizar testes de software sem estar ligado a uma rede – os pacotes são enviados para a própria máquina local.

A identificação de Transporte, que se funde no conceito do *socket* com a identificação do próprio *socket*, é designada por ***porto***. O porto é um identificador de 16 bits. Existem portos que foram reservados para aplicações muito importantes dos sistemas – World Wide Web, Correio Eletrónico, Terminal remoto, gestão, etc. Nunca devem ser usados por aplicações normais! Existe um grande intervalo de numeração de portos que é para uso geral e pode ser usado pelas aplicações normais. O programador pode escolher um número nesse intervalo, ou deixar o sistema escolher por ele. Quando um *socket* é criado, também é criado um objeto Java que é responsável por facilitar a comunicação através desse canal, e por gerar os eventos decorrentes de falhas na utilização do canal.

Quando se está a usar *sockets stream*, depois de criado o *socket* e ligado à aplicação remota, a simples escrita ou leitura no identificador local do *socket* basta para se comunicar com o interlocutor. Como os *sockets datagram* usam datagramas, cada escrita deve ser acompanhada com o endereço do *socket* para onde se pretende que essa informação seja enviada. Na leitura destes *sockets*, pode-se simplesmente ler e depois ver de onde veio a informação a partir dos campos de endereço associados, ou pode-se indicar ao sistema operativo que se pretende ler informação que seja proveniente de um determinado *socket* remoto, fornecendo essa informação de endereço na chamada de leitura.

4.3. Verificação da configuração

Na maior parte dos sistemas operativos (incluindo Windows XP, 7, 8, Linux, MacOs, etc.) é possível utilizar o comando (em modo de linha) '`netstat -a`' para obter a lista de portos ativos numa dada máquina para os protocolos UDP e TCP.

5. PROGRAMAÇÃO DE APLICAÇÕES PARA UMA REDE IPV4

O Java inclui um conjunto de classes (em `java.net.*`) que suporta o desenvolvimento de aplicações em rede. Nesta secção são apresentadas apenas as classes mais importantes para a programação de aplicações baseadas em *sockets*.

5.1. A classe `java.net.InetAddress`

A classe `InetAddress` permite lidar com endereços IPv4, suportando a conversão entre vários formatos. Suporta ainda a identificação do endereço IP da máquina local.

Um endereço IP pode ser representado por:

- uma *string* com um nome (e.g. "*tele1.dee.fct.unl.pt*");
- uma *string* com um endereço (e.g. "*193.136.127.217*");
- um *array* de 4 bytes com o endereço (e.g. byte[] addr = {193, 136, 127, 217};)

Qualquer um dos três formatos anteriores é suportado pela classe `InetAddress`, que internamente contém dois componentes privados: o nome (`String`) e o endereço (*array* de bytes).

Um endereço é vulgarmente inicializado a partir de uma *string*, através da função `getbyName()`:

```
String str= "193.136.127.217";
InetAddress netip;                          // Endereço IP
try {
    netip= InetAddress.getByName(str);      // converte 'string' em endereço IP
} catch (UnknownHostException e) {
    // Tratar excepção
}
```

Alternativamente, pode-se inicializar uma variável do tipo `InetAddress` com o endereço da máquina local utilizando o método estático `getLocalHost()` da classe `InetAddress`:

```
try {
    InetAddress addr = InetAddress.getLocalHost();  // Obtém endereço IP local
} catch (UnknownHostException e) {
    // tratar excepção
}
```

O método `getHostAddress()` permite obter o endereço em formato `String`. O nome associado ao endereço pode ser obtido com o método `getHostName()`.

5.2. *Sockets* datagrama

A interface para *sockets* datagrama é composta por duas classes: `DatagramPacket` e `DatagramSocket`.

5.2.1. A classe DatagramPacket

A classe `DatagramPacket` encapsula um pacote de dados. Esta classe inclui, como dados internos, o conteúdo da mensagem, um endereço IP e o número de porto, que podem conter o *socket* de destino ou de origem, conforme a mensagem seja enviada ou recebida.

Existem dois construtores para a classe `DatagramPacket`. O primeiro construtor limita-se a inicializar os dados da mensagem com o conteúdo de um *array* de bytes. O segundo construtor também inicializa o endereço IP e porto de destino. Observe-se que o valor do

endereço IP e número de porto pode ser modificado *a posteriori* usando os métodos `setAddress` e `setPort()`.

```
byte[] data= …;
DatagramPacket dp;
dp= new DatagramPacket(data, data.length);           // Construtor 1 – só define dados

OU

try {
    dp= new DatagramPacket(data, data.length, ip, porto); // Constr. 2 – define dados e
} catch (UnknownHostException e) {                      // destino do pacote (addr,porto)
    System.err.println("Erro a criar datagrama: "+ e);   // trata excepção
}
```

A classe define adicionalmente métodos para obter os valores dos vários campos internos:

- `InetAddress getAddress()` : devolve o endereço IP
- `int getPort()` : devolve o número de porto
- `byte[]getData()` : devolve um *buffer* com a mensagem
- `int getLength()` : devolve número de bytes contidos na mensagem

5.2.2. Composição e decomposição de mensagens

A mensagem que vai num pacote *datagram* é um *array* de bytes. O modo mais simples para o preencher é definir um `ByteArrayOutputStream` para se ter um `OutputStream` para memória, envolvido por um `DataOutputStream` para se escrever facilmente tipos primitivos ou `String`. O código baixo mostra isso. Repare que o `ByteArrayOutputStream` foi construído "no vazio", sendo no final alocado o *array* de bytes necessário.

```
ByteArrayOutputStream BAos= new ByteArrayOutputStream();
DataOutputStream dos= new DataOutputStream(BAos);
try {
    // Escrita sequencial dos componentes. Exemplos de funções:
    //     dos.writeBytes(str); - escreve string
    //     dos.writeShort(n);   - escreve short
} catch (IOException e) {
    // tratar excepção
}
byte [] buffer = BAos.toByteArray();        // Cria array de bytes com os dados
```

Na receção de dados o *buffer* onde está a mensagem que foi lida é constituído como `InputStream`, `ByteArrayInputStream`, e é envolvido com um `DataInputStream` para se ter os métodos de leitura de tipos primitivos e `String` (`readInt`, `readLong`, …).

```
ByteArrayInputStream BAis= new ByteArrayInputStream(buf, 0, len);
DataInputStream dis= new DataInputStream(BAis);
try {
    // Leitura sequencial dos componentes (exemplos de funções)
    //    byte [] aux= new byte [len];             // ler len bytes
    //    int n= dis.read(aux,0,len);              //   para uma
    //    String str= new String(aux, 0, len);     //    string
    //      em alternativa, poder-se-ia ler directamente de buf.
    //    int len_msg= dis.readShort();                  // Ler short
} catch (IOException e) {
    // tratar excepção
}
```

5.2.3. A classe DatagramSocket

A classe `DatagramSocket` permite criar *sockets* datagrama associados a um número de porto, e enviar e receber pacotes. A associação a um porto é realizada no construtor da classe. Existem três variantes de construtor, representadas em baixo:

- O construtor 1 inicia um *socket* num porto indefinido;
- O construtor 2 tenta iniciá-lo num porto definido;
- O construtor 3 define o porto e o endereço IP da interface, para o caso de existirem várias interfaces de rede.

```
public DatagramSocket() throws SocketException                    // Const 1
public DatagramSocket(int port) throws SocketException            // Const 2
public DatagramSocket(int port, InetAddress intf) throws SocketException // Con 3
```

Qualquer dos construtores pode gerar a exceção `SocketException`, indicando que não foi possível criar o *socket* porque, por exemplo, o número de porto pretendido não está livre. É possível obter o número de porto associado ao *socket* usando o método `getLocalPort()`. O porto associado ao *socket* é libertado usando o método `close()`.

Após iniciar um *socket*, é possível enviar pacotes usando o método `send()`. Caso falhe o envio do pacote é gerada a exceção `IOException`. Observe-se que o endereço IP e o número de porto são preenchidos no objeto `DatagramPacket`.

```
DatagramSocket ds= new DatagramSocket();
DatagramPacket dp = …                 // prepara pacote e define IP e porto
try {
    ds.send(dp);
} catch (IOException e) {
    System.err.println("Excepção :"+e);
}
```

Os pacotes são recebidos usando o método `receive()`. Caso não seja possível receber pacotes, é gerada a exceção `IOException`. O endereço IP e o número de porto de origem do pacote são guardados no objeto `DatagramPacket`.

```
DatagramSocket ds;
try {
    ds = new DatagramSocket(porto);          // Define porto
} catch (SocketException e) {
    System.err.println("Falhou criação de socket: "+e);
}
byte [] buf= new byte[PACKET_SIZE];  // Tamanho máximo teórico é 65535
DatagramPacket dp= new DatagramPacket(buf, buf.length);
try {
    ds.receive(dp);                  // Espera pela chegada de um pacote
    int len= dp.getLength();   // obtém dimensão do pacote recebido
    // buf contém len bytes de dados recebidos
    … processar pacote …
} catch (IOException e) {
    // Tratar erro
}
```

Por omissão, a operação de leitura é bloqueante, esperando indefinidamente por pacotes enquanto o *socket* for válido. No entanto, usando o método `setSoTimeout()` é possível definir um tempo máximo de espera por pacotes. Esta operação pode gerar a exceção `IOException` em caso de erro. Caso expire o tempo máximo de espera é gerada a exceção `SocketTimeoutException`.

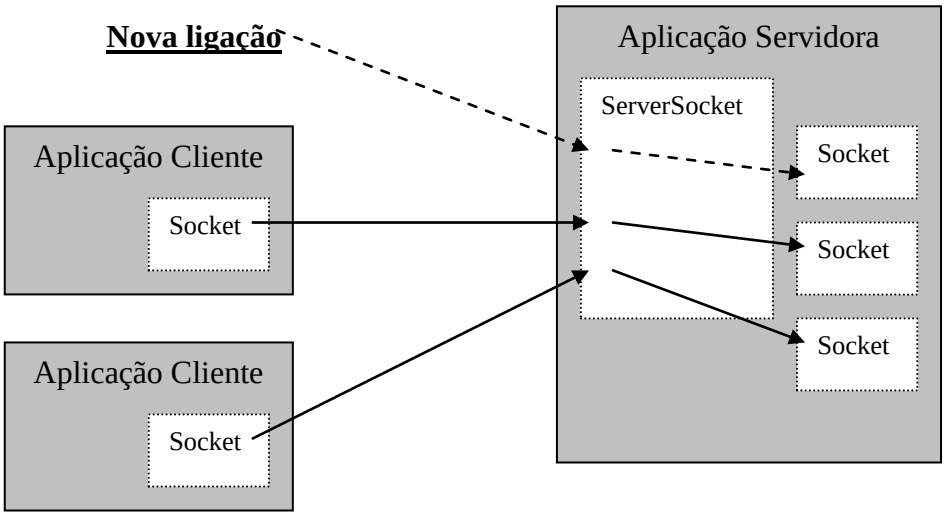
```
ds.setSoTimeout(3000);  // 3 segundos
try {
    ds.receive(dp);
    // processa o pacote
} catch (SocketTimeoutException e) {
    System.err.println("No packet within 3 seconds");
}
```

5.3 Sockets orientados à ligação

A interface para *sockets* orientados à ligação (TCP) é realizada através de duas classes: `ServerSocket` e `Socket`.

5.3.1. A classe ServerSocket

A classe `ServerSocket` define um objeto servidor que pode receber e manter várias ligações abertas. Quando se cria um objeto, define-se o porto onde ele vai escutar. O método `accept()` bloqueia o objeto até que seja recebida uma ligação a esse porto. Quando é recebida uma ligação é criado um objeto da classe `Socket` com outro número de porto para tratar da ligação.



Os construtores da classe `ServerSocket` permitem definir o porto, o número de novas ligações pendentes que são aceites pelo objeto e o endereço IP (a interface) a que se faz a associação. Existe ainda um quarto construtor sem parâmetros, que não inicializa o porto, permitindo usar a função `setReuseAddress()`, antes de definir o porto com a função `bind()`.

```
// Public Constructors
public ServerSocket(int port) throws IOException;
public ServerSocket(int port, int backlog) throws IOException;
public ServerSocket(int port, int backlog, InetAddress bindAddr) throws IOException;
public ServerSocket() throws IOException;
```

As suas duas funções principais permitem receber ligações e terminar o servidor.

```
public Socket accept() throws IOException;
public void close() throws IOException;
```

Outras funções permitem ter acesso a parâmetros e configurações do *socket*:

```
public InetAddress getInetAddress();
public int getLocalPort();
public synchronized int getSoTimeout() throws IOException;
public synchronized void setSoTimeout(int timeout) throws SocketException;
```

Para permitir comunicar com os clientes e aceitar novas ligações em paralelo é comum usar múltiplas tarefas (*threads*) para lidar com cada ligação.

5.3.2. A classe Socket

A classe `Socket` define um objeto de intercomunicação em modo feixe. Pode ser criado através de um construtor ou a partir da operação `accept()`. O construtor permite programar clientes: especifica-se o endereço IP e porto a que se pretende ligar, e o construtor estabelece a ligação.

```
public Socket(String host, int port) throws UnknownHostException, IOException;
public Socket(InetAddress address, int port) throws IOException;
public Socket(String host, int port, InetAddress localAddr,
                                            int localPort) throws IOException;
public Socket(InetAddress address, int port, InetAddress localAddr,
                                            int localPort) throws IOException;
```

As operações de escrita e leitura do *socket* são operações em *streams*. Foram descritas atrás e são relembradas na secção seguinte. Existem ainda funções para fechar o *socket* e para obter informações sobre a identidade da ligação.

```
public InputStream getInputStream() throws IOException;
public OutputStream getOutputStream() throws IOException;
public synchronized void close() throws IOException;
public InetAddress getInetAddress();
public InetAddress getLocalAddress();
public int getLocalPort();
public int getPort();
public isConnected();
public isClosed();
```

Várias operações de configuração dos parâmetros do protocolo TCP podem ser realizadas através de métodos desta classe. Caso esteja ativo, pode originar perda de dados não detetável pela aplicação.

```
public synchronized int getSoTimeout() throws SocketException;
public synchronized void setSoTimeout (int timeout) throws SocketException;
```

As funções `getSoTimeout()` e `setSoTimeout()` permitem configurar o tempo máximo que uma operação de leitura pode ficar bloqueada, antes de ser cancelada. Caso o tempo expire é gerada uma exceção `SocketTimeoutException`. Estas funções também existem para as classes `ServerSocket` e `DatagramSocket`.

5.3.3. Comunicação em sockets TCP

O método `getInputStream()` da classe `Socket` devolve um objeto da classe `InputStream` , e o método `getOutputStream()` um objeto da classe `OutputStream`. Deve-se envolver estes objetos como foi explicado atrás. Recordando:

- Para leitura é útil envolver com a classe `InputStreamReader` e depois envolver com a classe `BufferedReader`, para suportar o método `readLine()` para esperar pela receção de uma linha completa.
- Para a escrita é útil poder escrever `Strings`. Para isso deve-se envolver com `OutputStreamWriter` e depois com `PrintWriter` para se ter os métodos `print()` e `println()`.

Caso se pretenda dados num formato binário (não legível), deve-se usar as classes `DataInputStream` e `DataOutputStream`, apresentados anteriormente na secção 4.2.2. para composição de mensagens em *sockets* datagrama.

Um exemplo de utilização das classes `BufferedReader` e `PrintStream` para ler `String` é o seguinte:

```
try {          // soc representa uma variável do tipo Socket inicializada
      // Cria feixe de leitura
  InputStream ins = soc.getInputStream( );
  BufferedReader in = new BufferedReader(
      new InputStreamReader(ins, "8859_1" ));  // Tipo de caracter ISO-Latin-1
      // Cria feixe de escrita
  OutputStream out = soc.getOutputStream( );
  PrintStream pout = new PrintStream(out);
  // Em alternativa poder-se-ia usar PrintWriter:
  // PrintWriter pout = new PrintWriter(
  //        new OutputStreamWriter(out, "8859_1"), true);
      // Lê linha e ecoa-a  para a saída
  String in_string= in.readln();
  pout.writeln("Recebi: "+ in_string);
}
catch (IOException e ) { ... }
```

5.3.4. Exemplo de aplicação - TinyFileServ

No exemplo seguinte é apresentado um servidor de ficheiros parcialmente compatível com o protocolo HTTP (utilizável por browsers), que recebe o nome do ficheiro no formato (* *nome-completo* *) e devolve o conteúdo do ficheiro, fechando a ligação após o envio do último caracter do ficheiro. O servidor recebe como argumentos da linha de comando o número de porto a partir da linha de comando e o diretório raiz correspondente à diretoria "/" (e.g. *"java TinyFileServ 20000 /home/rit2/www"*). Para cada nova ligação, o servidor lança uma tarefa que envia o ficheiro pedido e termina após o envio do ficheiro. O servidor é parcialmente compatível com um browser: lê o segundo campo do pedido (e.g. GET / HTTP/1.1) e depois envia o ficheiro de resposta. Este exemplo usa algumas das classes que foram descritas anteriormente, mais outras que não vão ser usadas.

```
//file: TinyFileServd.java
import java.net.*;
import java.io.*;
import java.util.*;

// Classe thread que trata cada ligação individual
class TinyFileConnection extends Thread {
  private Socket client;
  private String root;
  TinyFileConnection ( Socket client, String root ) throws SocketException {
    this.client = client;
    this.root= root;
  }

  public void run( ) {
    try {
      BufferedReader in = new BufferedReader(
                new InputStreamReader(client.getInputStream( ), "8859_1" ));
      OutputStream out = client.getOutputStream( );
      PrintStream pout = new PrintStream(out, false, "8859_1");
      String request = in.readLine( );                    // Lê a primeira linha
      System.out.println( "Request: "+request );
// A primeira linha do pedido é da forma "GET /file.html HTTP/1.1"; do protocolo HTTP
// este código usa a classe "StringTokenizer" para separar nos 3 componentes
// Depois acrescenta a "root" ao caminho; se tiver apenas "/" acrescenta index.html
      StringTokenizer st= new StringTokenizer (request);
      if (st.countTokens() != 3)   // Se não tem 3 componentes não é válido
        return;  // Invalid request
      st.nextToken();                 // Salta primeiro componente
      String file= st.nextToken();    // obtém segundo componente
      // Ignora o terceiro componente
      // Cria nome de ficheiro a ser transferido
      String filename= root+file+(file.equals("/")?"index.htm":"");
      System.out.println("Filename= "+filename);
      FileInputStream fis = new FileInputStream ( filename );
      byte [] data = new byte [fis.available()]; // Aloca um array com tamanho do ficheiro
      fis.read( data );   // Lê todo o ficheiro para memória
      pout.print("http/1.1 200 OK\r\n\r\n"); // Escreve mensagem de texto para socket
      out.write( data );  // Escreve dados binários para o socket
      out.flush( );       // Força o envio de todos os dados para o ficheiro
      fis.close( );       // Fecha o ficheiro
    }
    catch ( FileNotFoundException e ) {
      System.out.println( "File not found" );
    }
    catch ( IOException e ) {
      System.out.println( "I/O error " + e );
    }
    catch ( Exception e ) {
      System.out.println( "Error " + e );
    }
    finally {       // É sempre corrido, mesmo que exista excepção
      try {
        client.close( );   // Fecha socket e todos os feixes associados
      } catch (Exception e) { /* Ignore everything */ }
    }
  }
} // end of class TinyFileConnection

// Classe principal - fica em ciclo à espera de ligações; lança uma tarefa por cada pedido
public class TinyFileServd {
  public static void main( String argv[] ) throws IOException {
    ServerSocket ss = new ServerSocket(
      Integer.parseInt(argv.length>0 ? argv[0] : "20000"));
    while ( true ) {
      Socket s= ss.accept();       // Recebe nova ligação
      TinyFileConnection th= new TinyFileConnection(s,(argv.length>1 ? argv[1] : ""));
      th.start();                  // Inicia nova thread e volta a espera por ligação
    }
  }
} // end of class TinyFileServd
```