U N I V E R S I D A D E N O V A D E L I S B O A

F a c u l d a d e e C i ê n c i a s e T e c n o l o g i a

Sumários Comentados

das

Aulas Teóricas de Linguagens de Programação I

A r t u r M i g u e l D i a s

2 0 0 4 / 2 0 0 5

Índice geral

Teórica 1

Paradigma

Um **paradigma** é um **modelo interpretativo** (ou **conceptualização**) duma realidade. Permite organizar as ideias com vista:

- Ao entendimento dessa realidade;
- À determinação de qual a melhor forma de actuar sobre essa realidade.

Pode dizer-se que um paradigma é um **ponto de vista** que determina como uma realidade é entendida e como se actua sobre ela.

Exemplos de tipos de paradigma

- **Paradigmas políticos (ideologias)**
  - Liberalismo
  - Fascismo
  - Comunismo
  - Socialismo

    [Sabemos que as diferentes ideologias interpretam distintamente a realidade social e estabelecem planos de acção diferentes.]
- **Paradigmas económicos (teorias)**
  - Keynesianismo (Keynes)
  - Monetarismo (Milton Friedman)
- **Paradigmas científicos (teorias)**
  - Mecânica de Newton
  - Mecânica de Einstein
- **Paradigmas militares**
  - Estratégia Medieval (armas brancas, fortificações)
  - Guerra Fria (armas nucleares, ameaça/contenção)
  - Gerrilha (tentativa de obter máximos resultados com recursos muitos escassos)

Algumas características dos paradigmas

- Os paradigmas são simplificações da realidade.
- Cada paradigma é caracterizado por um conjunto de **conceitos de base**.
- Descobre-se com frequência que dois paradigmas sobre uma mesma realidade são irreconciliáveis. Por exemplo, os vários paradigmas subjacentes à mentalidade Budista são de tal forma distintos dos paradigmas subjacentes à mentalidade Ocidental que a tradução de livros entre essas duas culturas é uma tarefa muitíssimo difícil e de resultados sempre muito discutíveis.
- A evolução das ideias e da tecnologia vai conduzindo ao aparecimento de novos paradigmas e, acima de tudo, à mudança do *paradigma consensual*.

Paradigma de programação

Um **paradigma de programação** consiste numa **conceptualização da computação**.

Computação = processamento mecânico de informação

Cada paradigma de programação é caracterizado por um conjunto particular de **conceitos de base**.

Exemplos de paradigmas de programação e respectivos conceitos de base

- **Paradigma imperativo**
  - Conceitos: estado, atribuição, sequenciação
  - Linguagens: Basic, Pascal, C, Assembler.
- **Paradigma funcional**
  - Conceitos: função, aplicação, avaliação
  - Linguagens: Lisp, ML, OCaml, Haskell.
- **Paradigma lógico**

- Conceitos: relação, dedução
- Linguagens: Prolog.

- **Paradigma orientado pelos objectos**
  - Conceitos: objecto, mensagem
  - Linguagens: C++, Java, Eiffel.

- **Paradigma concorrente**
  - Conceitos: processo, comunicação (síncrona ou assíncrona)
  - Linguagens: Occam, Ada, Java.

Aspectos práticos

- Cada paradigma de programação determina uma forma particular de abordar os problemas e de formular as respectivas soluções. De facto, diferentes paradigmas de programação representam muitas vezes visões irreconciliáveis do processo de resolução de problemas.
- O grau de sucesso dum programador depende em parte da colecção de paradigmas que domina e da sua arte em escolher o modelo conceptual (paradigma) mais indicado para analisar e resolver cada problema.

Uma linguagem de programação pode combinar mais do que um paradigma

- C++ --- Paradigma imperativo + paradigma orientado pelos objectos.
- Java --- Paradigma imperativo + paradigma orientado pelos objectos + paradigma concorrente.
- Ocaml --- Paradigma funcional + paradigma orientado pelos objectos.
- Ada --- Paradigma imperativo + paradigma concorrente.

Nota: Alguns autores separam o conceito de *paradigma de computação* (referindo-se ao modelo básico de computação) do conceito de *paradigma de programação* (referindo-se às construções e metodologias concretas colocadas à disposição do programador). Para evitarmos entrar em demasiados detalhes, a nossa ideia de *paradigma de programação* já incorpora as duas facetas.

Teórica 2

Programação imperativa *versus* programação funcional

Programação imperativa

A computação baseia-se na existência de um "estado" que vai evoluindo ao longo do tempo. O estado é modificado usando um "comando de atribuição" e existe um "operador de sequenciação". (Linguagens: Basic, Pascal, C).

Programação funcional

A computação consiste na avaliação de "expressões" nas quais podem ocorrer chamadas de funções. O objectivo da avaliação duma expressão é a produção dum "resultado" ou "resposta". Não há "atribuição" nem variáveis cujo valor possa ser alterado. Cada função limita-se a produzir um resultado a partir dos seus argumentos. (Linguagens: Lisp, ML, Haskell).

Estratégias de avaliação de expressões

Uma expressão da forma `f exp` pode ser avaliada usando pelo menos duas estratégias diferentes:

- **Estratégia *call-by-value*** (ou *eager evaluation*) --- Avalia primeiro o argumento antes de aplicar a função. Esta é a estratégia usada na linguagem funcional ML (e também nas linguagens Pascal, C, Java, etc). Exemplo de utilização desta estratégia:

```
(fun x -> x+1)(2+3)
= (fun x -> x+1)5
= 5+1
= 6
```

- **Estratégia *call-by-name*** (ou *lazy evaluation*) --- Aplica imediatamente a função ao argumento, adiando para mais tarde a avaliação desse argumento. Esta é a estratégia usada na linguagem funcional Haskell e outras. Exemplo de utilização desta estratégia:

```
(fun x -> x+1)(2+3)
= (2+3)+1
= 5+1
= 6
```

Diferente poder destas duas estratégias de avaliação

Estas duas estratégias dão sempre o mesmo resultado desde que as sequências de avaliação terminem. Mas pode acontecer que a estratégia *call-by-name* consiga terminar uma avaliação e a estratégia *call-by-value* não a consiga terminar. Por isso se diz que a estratégia do Haskell é mais poderosa do que a estratégia do ML.

Eis um exemplo que mostra o diferente poder das duas estratégias:

Vamos avaliar a expressão `(fun x -> 5)(loop 3)` usando ambas as estratégias. `loop` é a função assim definida:

```
let rec loop x = loop x ;;
```

Estratégia *call-by-value*:

```
(fun x -> 5)(loop 3)
= (fun x -> 5)(loop 3)
= (fun x -> 5)(loop 3)
= (fun x -> 5)(loop 3)
= ...  não termina
```

Estratégia *call-by-name*:

```
(fun x -> 5)(loop 3)
= 5   termina
```

Funções como valores de primeira classe

Nas linguagens funcionais as funções têm o estatuto de **valores de primeira classe**. Isso significa que as funções têm um estatuto tão importante como o dos inteiros, reais, e outros tipos predefinidos.

Concretamente, numa linguagem funcional as funções podem ...

- Ser passadas como argumento para outras funções;
- Podem ser retornadas por outras funções;
- Podem ser usadas como elementos constituintes de estruturas de dados;
- Têm uma representação literal própria. Exemplo em ML: `(fun x->x+1)`

Elementos essenciais da linguagem ML

- O ML é uma linguagem funcional.
- Um programa é uma sequência de funções. As funções podem ser recursivas.
- São dois os principais mecanismos que podem ser usados na escrita do corpo das funções em ML:
  - APLICAÇÃO (aplicação duma função a argumentos). Ex: `sqrt 9`
  - IF-THEN-ELSE. Ex: `if prime x then x else x/2`

Exemplo de função que usa os dois mecanismos:

```
let rec fact x =
    if x = 0 then 1 else x * fact (x-1)
;;
```

Tipos

Um **tipo** representa uma colecção de valores e tem um conjunto de *literais* e *operações* associadas.

Tipos básicos

TIPO	LITERAIS	OPERAÇÕES MAIS USADAS
int	5 -456	+ - * / mod min_int max_int
float	3.14e-21	+. -. *. /. sqrt exp log sin ceil floor float_of_int
string	"" "ola"	^ String.length String.sub String.get
bool	false true	not || &&
char	'a' '$'	int_of_char char_of_int
unit	()	ignore

Tipos compostos

TIPO	LITERAIS	OPERAÇÕES MAIS USADAS
'a->'b	(fun x -> x+1)	aplicação
'a*'b	(5, 5.6)	fst snd *emparelhamento de padrões*
'a list	[] [3;5;7]	*emparelhamento de padrões*
tipos produto	*diversos*	*emparelhamento de padrões*
tipos soma	*diversos*	*emparelhamento de padrões*

Inferência de tipos

Note que o tipo dos argumentos e do resultado das funções não se declaram em ML. A implementação faz **inferência de tipos**: ela infere para cada função o tipo mais geral que é possível atribuir a essa função.

Qual o tipo das seguintes funções?

```
fun x -> x+1                    : int -> int
fun x -> x +. 1.0               : float -> float
fun x -> x ^ x                  : string -> string
fun (x,y) -> x + y              : (int * int) -> int
fun (x,y) -> (y,x)              : ('a*'b) -> ('b*'a)
```

As quatro primeiras funções dizem-se **monomórficas** porque só aceitam argumentos de tipos fixos. A última função diz-se **polimórfica** pois aceita argumentos de tipos diversos.

Não há sobreposição (overloading)

A linguagem ML não suporta sobreposição (overloading) de nomes ou operadores.

Pelo contrário, a linguagem C++ suporta sobreposição. Por exemplo, em C++ o símbolo "+" é usado para denotar simultaneamente diversas operações: soma de inteiros, soma de reais, concatenação de strings, etc.

Por exemplo, em ML o operador "+" denota apenas a soma inteira.

Teórica 3

Funções com múltiplos argumentos

Em OCaml há duas formas de escrever funções com mais do que um argumento: a **forma não-curried** e a **forma curried**. A segunda forma é mais elaborada do que a primeira mas tem algumas vantagem técnicas.

Forma não-curried

Os vários argumento agrupam-se num único tuplo ordenado. Exemplos:

```
let nAdd (x,y) = x+y ;;
nAdd: (int * int) -> int

let nAdd3 (x,y,z) = x+y+z ;;
nAdd3: (int * int * int) -> int

nAdd (3,4) = 7
nAdd3 (2,8,1) = 11
```

Tecnicamente, a função `nAdd` tem apenas um parâmetro (do tipo `int*int`). Mas claro, através desse único parâmetro conseguimos passar duas unidades de informação.

Forma curried

Os argumentos ficam separados. Exemplos:

```
let cAdd x y = x+y ;;
cAdd: int -> int -> int

let cAdd x y z = x+y+z ;;
cAdd: int -> int -> int -> int

cAdd 3 4 = 7
cAdd 2 8 1 = 11
```

Escrever *funções curried* não tem nada que saber: basta usar espaços em branco a separar os parâmetros, na cabeça de cada função. Contudo o tipo destas funções afigura-se surpreendente para quem o observa pela primeira vez. Isso é resultado da representação interna das funções em OCaml. Segue-se a explicação...

Representação interna das funções em OCaml

Por uma questão de regularidade de implementação, o OCaml só usa internamente funções anónimas com um único argumento. Uma função escrita com múltiplos argumentos acaba sempre convertida para um formato interno especial - chamado **forma interna** - envolvendo apenas funções anónimas com um único argumento.

Por exemplo, a função

```
let cAdd x y = x+y ;;
```

é internamente convertida em:

```
let cAdd = fun x -> (fun y -> x+y) ;;
```

Repare como é engenhosa a ideia que está por detrás do esquema de tradução. Trata-se duma ideia matematicamente correcta que, ao mesmo tempo, está na base da implementação simples e regular do OCaml.

Vejamos mais alguns exemplos de tradução, lado a lado:

```
let cAdd x y z = x+y+z ;;
```

```
let cAdd = fun x -> (fun y -> (fun z -> x+y+z)) ;;
```

```
let f x = x+1 ;;
```

```
let f = fun x -> x+1 ;;
```

```
let nAdd (x,y) = x+y ;;
```

```
let nAdd = fun (x,y) -> x+y ;;
```

Avaliação de expressões envolvendo funções com múltiplos argumentos

Compare a avaliação das duas seguintes expressões:

```
nAdd (2,3)
```

```
= 2 + 3
= 5

cAdd 2 3
= (fun y -> 2+y) 3
= 2 + 3
= 5
```

Aplicação parcial

As funções curried têm a vantagem de poderem ser **aplicadas parcialmente** ou seja, poderem ser invocadas omitindo alguns dos argumentos do final. Eis um exemplo de aplicação parcial:

```
let succ = cAdd 1 ;;
succ: int -> int
```

Associatividades

- O **operador de aplicação** é associativo à esquerda. (Note que este operador é *invisível*, pois nunca se escreve).
- O **contructor de tipos funcionais ->** é associativo à direita.

Portanto, a expressão `f a b` deve ser interpretada como `(f a) b`.

Portanto, o tipo `int -> int -> int` deve ser interpretado como `int -> (int -> int)`.

Formas equivalentes de escrever a mesma função

As seguintes quatro declarações são equivalentes, no sentido em que declaram a **mesma** função `f:int->int->int`.

```
let f x y = x + y ;;
let f x = (fun y -> x+y) ;;
let f = (fun x y -> x+y) ;;
let f = (fun x -> (fun y -> x+y)) ;;
```

Todas são convertidas para a mesma forma interna.

Teórica 4

Listas homogéneas em OCaml

Apresentam-se aqui os três aspectos essenciais relativos as listas em OCaml: como se escrevem listas literais; como se constroem novas listas a partir de listas mais simples; como se analisam listas e se extraem os seus elementos constituintes.

Listas literais

Exemplos de listas literais:

```
[]                                  : 'a list
[2;4;8;5;0;9]                       : int list
["ola"; "ole"]                      : string list
[[1;2]; [4;5]]                      : int list list
[(fun x -> x+1); (fun x -> x*x)]    : (int->int) list
```

Construtores de listas

Estão disponíveis dois construtores de listas que, como o nome indica, servem para construir listas novas:

```
[] : 'a list
:: : 'a -> 'a list -> 'a list
```

- O construtor `[]` chama-se "lista vazia" e representa a lista vazia.
- O construtor `::` chama-se "cons" e serve para construir listas não vazias.

O operador :: é associativo à direita.

Exemplos de utilização de `cons`:

```
2::[3;4;5] = [2;3;4;5]
1::2::3::[] = [1;2;3]
[]::[] = [[]]
[1;2]::[3;4] = ERRO
4::5 = ERRO
[1;2]::[[3;4]] = [[1;2];[3;4]]
```

Emparelhamento de padrões

O processamento de listas efectua-se por *análise de casos*, usando a construção **match** e *padrões*. Exemplo:

```
let rec len l =
    match l with
        [] -> 0
      | x::xs -> 1 + len xs
;;

len: 'a list -> int
```

A função anterior trata dois casos, cada um dos quais tem um padrão diferente associado. Os vários casos são analisados sequencialmente, de cima para baixo.

A utilização de padrões torna as funções mais fáceis de escrever e de entender. Os padrões são, portanto, bons amigos do/a programador/a. As linguagens funcionais antigas (eg. Lisp) não usavam padrões, mas as linguagens funcionais modernas (eg. ML, Haskell) não os dispensam.

Método indutivo

Redução de problemas a problemas mais simples

Considere as seguintes duas funções recursivas, escritas em ML:

```
let rec fact n =
    if n = 0 then 1
    else n * fact (n-1)
;;

let rec len l =
    match l with
```

```
        [] -> 0
      | x::xs -> 1 + len xs
;;
```

Analisando estas duas funções recursivas `len` e `fact`, verificamos que quando lidam com o caso não-trivial - o chamado **caso geral** - ambas efectuam **reduções de problemas a problemas mais simples**. Concretamente, a função `len` reduz o problema `len (x::xs)` ao problema mais simples `len xs`, e a função `fact` reduz o problema `fact n` ao problema mais simples `fact (n-1)`.

Além de efectuarem *redução de problemas*, as duas funções lidam também, separadamente, com o caso trivial de cada problema - o chamado **caso base**. Concretamente, a função `len` trata separadamente o caso base `len []` e a função `fact` trata separadamente o caso base `fact 0`.

Repare que quando uma destas duas funções é aplicada a um argumento, inicia-se uma sucessão de *reduções a problemas mais simples* que só termina quando o caso base é alcançado. Por exemplo, a sequência de *reduções* produzida pela aplicação `len [7;6;5;4]` é a seguinte:

```
len [7;6;5;4]              <-- problema inicial
= 1 + len [6;5;4]          <-- problema um pouco mais simples
= 1 + 1 + len [5;4]        <-- problema ainda mais simples
= 1 + 1 + 1 + len [4]      <-- problema ainda mais simples
= 1 + 1 + 1 + 1 + len []   <-- caso base
= 1 + 1 + 1 + 1 + 0
= 4
```

As funções recursivas `len` e `fact` são representativas do que é programar em ML. Em ML programa-se essencialmente *reduzindo problemas a problemas mais simples*.

Técnica do método indutivo

O método indutivo serve para ajudar a programar funções recursivas que efectuam *reduções de problemas a problemas mais simples*.

O método indutivo consiste na seguinte técnica:

- Para programar o caso geral G duma função recursiva, assume-se como PONTO DE PARTIDA (para começar a raciocinar) que o resultado S de aplicar a função a argumentos *um pouco mais simples do que os iniciais* já se encontra disponível.
- O problema que é então preciso então resolver é o seguinte: COMO SE PASSA DE S PARA G?

Exemplo de utilização do método indutivo

Problema: Programar uma função `len` que determine o comprimento de listas.

Resolução: Para começar, a função `len` recebe uma lista, a qual terá de ser processada usando emparelhamento de padrões (não há outra maneira, em ML). Aplicando o método indutivo ao caso geral `G=len (x::xs)`, vamos começar por assumir que já temos o problema resolvido para o caso, mais simples, `S=len xs`.

Desde já podemos ir escrevendo a seguinte estrutura incompleta:

```
let rec len l =
    match l with
        [] -> ...
      | x::xs -> ... len xs ...
;;
```

O problema que temos para resolver é o seguinte: *Como é que se obtém o valor de `G=len (x::xs)` a partir do valor de `S=len xs`?* Mas este problema é simples! De facto, basta adicionar uma unidade a S para se obter G!

Preenchendo o que falta no esquema anterior, surge a solução final:

```
let rec len l =
    match l with
        [] -> 0
      | x::xs -> 1 + len xs
;;
```

NOTA1: Neste exemplo, reduzimos o problema `G=len (x::xs)` ao problema `S=len xs`. Mas esta não é a única redução possível... Existem muitas outras... Voltaremos a este assunto noutras aulas.

NOTA2: A função `len` é tão simples que faz o método indutivo parecer óbvio e desinteressante. No entanto,

este método tem imensas virtualidades:

- Ajuda-nos a organizar o pensamento quando queremos resolver problemas complicados (ver as funções da terceira aula prática: `power`, `sort`, etc).
- Simplifica a resolução de problemas. [Repare que já não temos de resolver "o problema completo": passamos a ter de resolver "apenas" o problema da passagem de S para G, o que costuma ser "muito mais fácil".]
- Ajuda-nos a descobrir soluções simples e compactas.

Mais funções sobre listas programadas usando o método indutivo

Comprimento de lista:

```
let rec len l =
    match l with
        [] -> 0
      | x::xs -> 1 + len xs
;;
```

Soma dos elementos de lista:

```
let rec sum l =
    match l with
        [] -> 0
      | x::xs -> x + sum xs
;;
```

Concatenação de listas:

```
let rec append l1 l2 =
    match l1 with
        [] -> l2
      | x::xs -> x::append xs l2
;;
```

Inversão de lista:

```
let rec rev l =
    match l with
        [] -> []
      | x::xs -> (rev xs)@[x]
;;
```

Aplicação de função a todos os elementos de lista:

```
let rec map f l =
    match l with
        [] -> []
      | x::xs -> (f x)::map f xs
;;
```

Selecção dos elementos duma lista que verificam uma dada propriedade:

```
let rec filter b l =
    match l with
        [] -> []
      | x::xs ->
            if b x then x::filter b xs
            else filter b xs
;;
```

Exemplos de avaliações

```
sum [1;2;3]
= 1 + sum [2;3]
= 1 + 2 + sum [3]
= 1 + 2 + 3 + sum []
= 1 + 2 + 3 + 0
= 6

append [1;2] [3;4]
```

```
= 1::append [2] [3;4]
= 1::2::append [] [3;4]
= 1::2::[3;4]
= [1;2;3;4]

rev [1;2;3]
= (rev [2;3]) @ [1]
= (rev [3]) @ [2] @ [1]
= (rev []) @ [3] @ [2] @ [1]
= [] @ [3] @ [2] @ [1]
= [3;2;1]
```

A função @

A função `append` está predefinida em OCaml, sendo denotada pelo operador binário `@`. Assim, podemos escrever:

```
[1;2;3] @ [4;5;6] = append [1;2;3] [4;5;6] = [1;2;3;4;5;6]
```

No entanto, a função `@` é pouco eficiente e deve ser usada com critério. Por exemplo, para acrescentar um zero à cabeça duma lista `list` é má ideia escrever `[0]@list`: escreve-se sempre `0::list`. No entanto, para acrescentar um zero ao final duma lista `list` não temos alternativa: escreve-se mesmo `list@[0]`.

Teórica 5

Padrões

Padrão

Um **padrão** é uma expressão especial que representa um **conjunto de valores**.

Exemplos:

```
Padrão          Conjunto de valores representados
~~~~~~          ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
[]              lista vazia
[x]             listas com um elemento
[x;y]           listas com dois elementos
x::xs           listas não vazias
x::y::xs        listas com pelo menos dois elementos
5::xs           listas cujo primeiro elemento é 5
x               todos os valores (padrão universal)
_               padrão universal anónimo
(x,y)           todos os pares ordenados
(0,y)           todos os pares ordenados cuja 1ª componente é 0
8               inteiro 8
(x,y)::xs       lista não vazia de pares ordenados
'a'..'z'        letras de 'a' a 'z'
```

Numa expressão `match` podem ocorrer padrões em número variável (ver exemplos 1 e 2). Durante a avaliação, esses padrões são explorados sequencialmente, de cima para baixo.

Exemplo1:

```
let rec len l =
    match l with
        [] -> 0
      | x::xs -> 1 + len xs
;;
```

Exemplo2:

```
let rec count5 l =
    match l with
        [] -> 0
      | 5::xs -> 1 + count5 xs
      | x::xs -> count5 xs
;;
```

No cabecalho duma função anónima, no lugar do argumento, também pode ocorrer um padrão (ver exemplo 3).

Exemplo:

```
fun (x,y) -> (y,x)
```

Regra da seta ->

- No contexto que antecede a seta ->, as expressões são interpretadas como padrões (por exemplo, nesse contexto, x::xs representa o conjunto de todas as listas não vazias);
- Fora do contexto que antecede a seta ->, as expressões são interpretadas da forma habitual (por exemplo, fora desse contexto, x::xs representa a aplicação do constructor "cons" aos nomes x e xs).

Padrões disjuntos

Dois padrões dizem-se **disjuntos** se os conjuntos que eles representam são disjuntos. Numa expressão `match` a ordem dos casos só é irrelevante se os padrões forem disjuntos entre si.

Exemplos:

- Os dois padrões que ocorrem na função `len` **são disjuntos**:

```
let rec len l =
    match l with
```

```
          [] -> 0
        | x::xs -> 1 + len xs
;;
```

- Os dois padrões que ocorrem na função `fact` **não são disjuntos**:

```
let rec fact n =
     match n with
       0 -> 1
     | n -> n * fact (n-1)
;;
```

Emparelhamento

Como resultado do **emparelhamento dum valor com um padrão**, há duas consequências possíveis:

- Falhanço;
- ou Sucesso, ficando os nomes do padrão ligados a valores.

Num padrão, não se permitem repetições de nomes. Por exemplo o padrão `(x,x)::xs` é inválido.

A seguinte função aplica-se a uma lista de pares ordenados, e conta o número de pares com as duas componentes iguais:

```
let rec eqPairs l =
    match l with
      [] -> 0
    | (x,y)::xs ->
        if x=y then 1 + eqPairs xs
        else eqPairs xs
;;
```

Nomes locais

Construção let-in

A construção **let-in**:

```
let n = exp in
    exp1
```

associa o nome `n` ao valor da expressão `exp` no contexto da expressão exp_1.

O exercício do "parque de estacionamento", resolvido na aula prática 1, ajuda a mostrar como esta construção pode contribuir para a legibilidade de certas funções. Comparemos a versão original da função principal com uma nova versão que tira partido da construção **let-in**:

Versão original:

```
let parque (he, me) (hs, ms) =
    pagar (convHoras (permanencia (convMinutos he me) (convMinutos hs ms)))
;;
```

Nova versão:

```
let parque (he, me) (hs, ms) =
    let te = convMinutos he me in
        let ts = convMinutos hs ms in
            let perm = permanencia te ts in
                let h = convHoras perm in
                    pagar h
;;
```

A construção **let-in** também pode ser usada para aumentar a eficiência de certas funções. Por exemplo a segunda função é mais eficiente do que a primeira:

```
let f g x =
    g x + g x * g x
;;

let f g x =
    let r = g x in
```

```
        r + r * r
;;
```

Tradução

É interessante saber que, no caso geral, o OCaml traduz a construção

```
let n = exp in
    exp1
```

para a seguinte representação interna:

```
(fun n -> exp1) exp
```

Isto também mostra que se pode usar um padrão dentro dum `let`, na posição do nome `n`. Por exemplo:

```
let rec f l =
    match l with
        [] -> ([],[])
      | x::xs ->
          let (ys,zs) = f xs in
            if x >= 0 then (ys, x::zs)
            else (x::ys, zs)
;;
```

O que faz esta função?

Nomes mutuamente recursivos

Em ML a palavra reservada **and** serve para definir **nomes mutuamente recursivos**. Eis um exemplo curioso:

```
let rec even n =
     if n = 0 then true else odd (n-1)
and odd n =
     if n = 0 then false else even (n-1)
;;
```

O que fazem estas funções?

Teórica 6

Tipos produto (registos)

A maioria das linguagens de programação incluem nos seus sistemas de tipos uma construção específica que permite representar agrupamentos de dados heterogéneos. Independentemente da linguagem, o nome genérico dessa construção é **tipo produto**.

Para exemplificar, numa linguagem com tipos produto é possível definir um tipo de dados *pessoa* constituído por um nome (do tipo string), um ano de nascimento (do tipo int), uma morada (do tipo string), etc.

Os tipos produto do Pascal são os *registos*.
Os tipos produto do C são as *estruturas*.
Os tipos produto do Java, Smalltalk e C++ são as *classes*.

Como é em ML?

Tipos produto em ML

Tipos produto não etiquetados

Os produtos cartesianos do ML são exemplos de tipos produto, neste caso ditos **não-etiquetados**. Por exemplo, para representar *pessoas*, pode usar-se em ML o seguinte produto cartesiano:

```
string * int * string
```

Para exemplificar, eis um valor do tipo anterior:

```
("João", 1970, "Lisboa")
```

Tipos produto etiquetados

Mas o ML, também suporta **tipos produto etiquetados**. Eis dois exemplos de tipos produto etiquetados:

```
type pessoa = { nome:string ; anoNasc:int ; morada:string  } ;;
type tempo = { hora:int ; minuto:int } ;;
```

Eis dois literais destes tipos:

```
type pessoa = { nome:string ; anoNasc:int ; morada:string  } ;;
type tempo = { hora:int ; minuto:int } ;;
```

Como se processam os elementos dum tipo produto etiquetado? De duas formas:

- Usando **emparelhamento de padrões**:

```
let getName p =
    match p with
      { nome = x ; anoNasc = y  ; morada = z } -> x
;;
```

- Ou usando a **operação de acesso** "." e que funciona em ML exactamente como em Pascal ou C:

```
let getHora t = t.hora ;;
```

Vejamos uma função que soma dois tempos:

```
let somaTempos t1 t2 =
    { hora = t1.hora + t2.hora + (t1.minuto + t2.minuto) / 60
    ; minuto = (t1.minuto + t2.minuto) mod 60 }
;;
```

Tipos soma (uniões)

Muitas linguagens de programação incluem uma construção específica para a definição de tipos cujos valores podem assumir formas diversas. Essa construção designa-se genericamente por **tipo soma**.

Por exemplo, numa linguagem com suporte para tipos soma é possível definir um tipo de dados *pessoa* cujos valores assumem as duas seguintes *variantes*: *Homem* e *Mulher*.

Um tipo soma tenta conciliar o *geral* com o *particular*: por um lado um Homem ou uma Mulher são *pessoas*, e é

possível associar determinados dados a todas as *pessoas*; mas por outro lado são *variantes*, e cada uma delas pode ter dados específicos associados. Por exemplo, faz sentido colocar nas *Mulheres*, mas não nos *Homens*, a data da última licença de maternidade.

Os tipos soma do Pascal são os *registos com variante*.
Os tipos soma do C são as *uniões*.
Os tipos soma do Java, Smalltalk e C++ são as *classes abstractas* (imagine por exemplo uma classe abstracta *pessoa* com duas subclasses concretas: *Mulher* e *Homem*). [Em rigor, num aspecto as classes abstractas são um bocadinho diferentes dos tipos soma: os tipos soma são entidades fechadas enquanto que as classes abstractas são infinitamente extensíveis.]

Como é em ML?

Tipos soma em ML

O tipo soma *pessoa*, atrás referido, pode ser definido em ML da seguinte forma, usando os tipos auxiliares pessoaM e pessoaH:

```
type pessoaM = { nomeM:string ; anoNascM:int ; moradaM:string ; licenMater: string } ;;
type pessoaH = { nomeH:string ; anoNascH:int ; moradaH:string } ;;

type pessoa = Mulher of pessoaM | Homem of pessoaH ;;
```

Eis dois literais do tipo pessoa. Repare como o nome de cada variante se usa para marcar os respectivos literais.

```
let a = Mulher { nomeM="Maria" ; anoNascM=1960 ; moradaM="Lisboa" ;
licenMater="25Out2000" } ;;
let b = Homem { nomeH="José" ; anoNascH=1960 ; moradaH="Lisboa" } ;;
```

Outro exemplo de tipo soma: O tipo predefinido *'a list* é um tipo soma!

Repare que as listas podem assumir duas formas distintas - *listas vazias* e *listas não vazias* - e que determinadas operações só se podem definir para uma das variantes: por exemplo, só se pode definir a operação de *obtenção de cabeça* para *listas não vazias*. Pode programar-se assim:

```
let head (x::xs) = x ;;
```

Sabemos que o tipo *'a list* está predefinido em ML, o que é muito prático. No entanto, se não estivesse predefinido, nós mesmos conseguiríamos definir um tipo lista equivalente, usando a seguinte definição:

```
let head (x::xs) = x ;;
```

Para exemplificar, eis uma constante do nosso tipo lista:

```
Node(5, Node(7, Node(-1, Nil)))
```

Os elementos de qualquer tipo soma são processados usando **emparelhamento de padrões**. Eis como se escreve uma função que determina o comprimento duma lista das nossas:

```
let rec len l =
    match l with
        Nil -> 0
      | Node(x,xs) -> 1 + len xs
;;
```

Etiquetas dum tipo soma

As etiquetas dum tipo soma tem três papéis:

- Identificam os vários ramos do tipo soma.
- Denotam os *construtores* que o sistema cria automaticamente para o tipo em causa. Um *construtor* é uma função especial que gera valores dum tipo soma.
- São elementos constituintes de novos padrões que a linguagem passa a reconhecer automaticamente.

Por exemplo, no tipo *'a list* por nós definido, a etiqueta `Node`...

- Identifica um ramo do tipo *'a list*;
- Denota um *construtor*, de tipo `('a * 'a list) -> 'a list`, que o sistema gera automaticamente;
- Pode ser usada em padrões: por exemplo, o padrão `Node(x,xs)` representa o conjunto das listas não vazias.

Árvores binárias

O útil tipo soma **árvore binária** não está predefinido em ML. Define-se assim usando um tipo soma:

```
type 'a tree = Nil | Node of 'a * 'a tree * 'a tree ;;
```

Eis uma constante do tipo "int tree":

```
Node(1, Node(2,Nil,Nil), Node(3,Nil,Nil))
```

Eis uma função que determina o número total de nós duma árvore binária:

```
let rec size t =
    match t with
        Nil -> 0
      | Node(x,l,r) ->
            1 + (size l) + (size r) ;;
```

[Qual é o tipo desta função "size"?]

Eis um exemplo de avaliação duma expressão envolvendo uma chamada da função "size":

```
  size (Node(1, Node(2,Nil,Nil), Node(3,Nil,Nil)))
= 1 + size (Node(2,Nil,Nil)) + size (Node(3,Nil,Nil))
= 1 + (1 + size Nil + size Nil) + (1 + size Nil + size Nil)
= 1 + (1 + 0 + 0) + (1 + 0 + 0)
= 1 + 1 + 1
= 3
```

Método indutivo aplicado a árvores binárias

No caso das árvores binárias, o método indutivo consiste na seguinte técnica:

- Para programar o caso geral G duma função recursiva, assume-se como PONTO DE PARTIDA PARA COMEÇAR A RACIOCINAR que o resultado L de aplicar a própria função à subárvore da esquerda e o resultado R de aplicar a própria função à subárvore da esquerda já se encontram disponíveis. O problema que é então preciso resolver é o seguinte: COMO É QUE A PARTIR DE L E R SE CHEGA A G?

Exemplo de utilização do método indutivo

Problema: Programar uma função "size" que determine o número total de nós duma árvore binária:

Resolução: A função "size" aplica-se a uma árvore, a qual será, naturalmente, processada usando emparelhamento de padrões. Aplicando o método indutivo ao caso geral "G=Node(x,l,r)", vamos começar por assumir que já temos o problema resolvido para os casos "L=size l" e "R=size r". Obtemos assim o seguinte PONTO DE PARTIDA PARA COMEÇAR A RACIOCINAR:

```
 let rec size t =
    match t with
         Nil ->  ...
       | Node(x,l,r) -> ... size l ... size r ...
;;
```

O problema que temos para resolver é o seguinte: *Como é que se obtém o valor de "G=size (Node(x,l,r))" a partir dos valores "L=size l" e "R=size r"?* Mas este problema é simples. De facto, basta adicionar uma unidade a L+R para se obter G!

Preenchendo o que falta no esquema anterior, surge a solução final:

```
let rec size t =
    match t with
        Nil -> 0
      | Node(x,l,r) ->
            1 + (size l) + (size r) ;;
```

Mais uma função sobre árvores binárias

Altura duma árvore:

```
(* height: 'a tree -> int *)

let rec height t =
    match t with
```

```
        Nil -> 0
      | Node(x,l,r) ->
        1 + max (height l) (height r)
;;
```

Árvore ao espelho:

```
(* mirror: 'a tree -> 'a tree *)

let rec mirror t =
    match t with
      Nil -> Nil
    | Node (x,l,r) ->
        Node (x,mirror r,mirror l)
;;
```

Teórica 7

Tratamento de excepções em ML

Durante a execução de um programa, em situações geralmente anómalas são geradas **excepções**.

As excepções podem ser geradas:

- Ao nível do hardware. Por exemplo, em virtude duma divisão por zero.
- Ao nível do sistema operativo. Por exemplo, devido à impossibilidade de abrir um ficheiro inexistente, ou devido à tentativa de continuar a ler dum ficheiro que já chegou ao fim.
- Usando explicitamente a construção **raise**, nos programas. Por exemplo, para lidar explicitamente com **argumentos proibidos**:

```
let rec fact x =
        if x = 0 then 1
        else if x>0 then x * fact(x-1)
        else raise(Invalid_argument "fact")
;;
```

Captura e tratamento de excepções

Quando uma excepção é gerada, o programa entra num modo de funcionamento especial, dito **modo de tratamento de excepção**.

Por defeito, a geração de qualquer excepção faz o programa terminar imediatamente, produzindo uma mensagem de erro. Exemplos:

```
# 4/0 ;;
Exception: Division_by_zero.
# open_in "fdsg" ;;
Exception: Sys_error "fdsg: No such file or directory".
```

Mas este comportamento por defeito pode ser alterado se o programador quiser **capturar e tratar as excepções** no seu próprio programa. Para isso usa-se a construção **try-with** aqui apresentada esquematicamente:

```
try
        exp
with Sys_error _ -> exp1
   | Division_by_zero -> exp2
   | End_of_file -> exp3
   ...
;;
```

Descrição da expressão **try-with**:

- A expressão anterior tem o mesmo valor de `exp`, se nenhuma excepção for gerada durante a avaliação de `exp`.
- Mas se for gerada alguma excepção durante a avaliação de `exp`, então o **try-with** usa emparelhamento de padrões para descobrir qual das expressões `exp1`, `exp2`, `exp3`, etc., deverá ser avaliada em substituição de `exp`.
- Se este emparelhamento de padrões falhar, então a excepção é propagada para o **try-with** prévio, em termos de ordem de activação.
- Se não existir qualquer **try-with** então é activado o comportamento por defeito, sendo produzida uma mensagem de erro.

Efeitos laterais

O OCaml usa um modelo de input/output imperativo, no qual as operações de input/output são tratadas como *efeitos laterais*.

O que é um **efeito lateral**? Um *efeito lateral* é qualquer actividade que uma função desenvolva para além de calcular um resultado a partir dos argumentos. Por exemplo: escrever num ficheiro, escrever no écran do computador, ler dum ficheiro, etc.

Como sabe, no paradigma funcional é suposto uma função limitar-se a calcular um resultado a partir dos seus

argumentos. O facto é que a linguagem OCaml ultrapassa os limites estritos do paradigma funcional, já que admite efeitos laterais nas funções.

Operador de sequenciação

Para sequenciar os efeitos laterais, o OCaml dispõe dum *operador de sequenciação* que se escreve ";" (como em Pascal!). Assim, por exemplo, para escrever no écran a string "ola" seguida da string "ole" poderíamos usar a seguinte função:

```
let hello () =
    print_string "ola" ; print_string "ole"
;;
```

Tecnicamente, ";" é o nome duma função com o seguinte tipo:

```
";": unit -> 'b -> 'b
```

e aproximadamente a seguinte definição:

```
";" x y = y ;;
```

(Esta definição funciona porque a linguagem usa a estratégia de avaliação "call-by-value".)

Tipo unit e valor ()

Por vezes gostaríamos de poder definir funções sem argumentos ou sem resultados: isso faria sentido para funções só com efeitos laterais. Mas o ML não o permite. Ele obriga todas as funções a ter pelo menos um argumento e um resultado. O tipo **unit** foi inventado para ser usado nessas situações.

O tipo **unit** é um tipo básico com apenas um valor, que se escreve **()**. [Para efeitos de comparação, note que o tipo **bool** é um tipo com apenas dois valores, que se escrevem **true** e **false**.]

Exemplos de utilização de unit e ().

```
print_string: string -> unit
read_int: unit -> int
print_newline: unit -> unit

let x = read_int () in
    print_int (x+1)
```

Input-Output em ML

Canais

As operações sobre ficheiros são efectuadas através de **canais**, que são valores dos tipos predefinidos *in_channel* e *out_channel*.

Existem três canais predefinidos: *stdin*, *stdout*, *stderr*. Estes três canais são automaticamente abertos quando o programa começa a correr.

Para ajudar a perceber: os "canais" do ML correspondem aos "ficheiros internos" do Pascal.

Primitivas de input/output

```
(* Primitivas para escrita em stdout *)
    print_char: char -> unit
    print_string: string -> unit
    print_int: int -> unit
    print_float: float -> unit
    print_newline: unit -> unit

(* Primitivas para leitura de stdin *)
    read_line: unit -> string
    read_int: unit -> int
    read_float: unit -> float

(* Primitivas para abertura e fecho de canais *)
    open_in: string -> in_channel
    open_out: string -> out_channel
    close_in: in_channel -> unit
    close_out: out_channel -> unit
```

```
(* Primitivas para escrita em canais *)
    output_char: out_channel -> char -> unit
    output_string: out_channel -> string ->unit
    flush: out_channel -> unit

(* Primitivas para leitura de canais *)
    input_char: in_channel ->char
    input_line: in_channel -> string
```

Esta lista de primitivas não é exaustiva. Há mais primitivas listadas no manual de referência.

Exemplo de função sobre ficheiros programada usando o método indutivo

Cópia de ficheiros:

```
(* copyChannel: copia o canal de input ci para o canal de output co *)
let rec copyChannel ci co =
    try
        let s = input_line ci in
            output_string co s ;
            output_string co "\n" ;
            copyChannel ci co
    with End_of_file -> () ;;

(* copyFile: abre os ficheiros ni e depois usa a funcao copyChannel *)
let copyFile ni no =
    let ci = open_in ni in
        let co = open_out no in
            copyChannel ci co ;
            close_in ci ;
            close_out co ;;
```

Esquema geral da utilização do método indutivo no tratamento de ficheiros, linha a linha:

```
let rec f ci =
    try
        let s = input_line ci in
            ... f ci ...
    with End_of_file -> ... ;;
```

A construção `try-with` é explicada na secção seguinte.

Teórica 8

Teste sobre os aspectos essenciais da linguagem ML.

Teórica 9

Funções parciais

Uma **função parcial** é uma função que só está definida em parte do seu domínio. (Não confundir com *aplicação parcial*, que é outra coisa.)

Por exemplo, a função `fact` é parcial porque só está definida para valores não-negativos:

```
let rec fact n = (* pre: n >= 0 *)
    if n = 0 then 1
    else n * fact (n-1)
;;
```

Outro exemplo: A função `maxList` é parcial porque só está definida para listas não-vazias.

```
let rec maxList l = (* pre: l <> [] *)
    match l with
        [x] -> x
      | x::xs -> max x (maxList xs)
;;
```

Quando se escreve uma função parcial, espera-se que essa função seja sempre aplicada a argumentos válidos. Mas os programas podem ter erros. Por isso, convém garantir que a função não produzirá qualquer resultado, quando aplicada a argumentos inválidos (por outras palavras, temos de obrigar o resultado a ser realmente *indefinido*). Há duas formas de impedir uma função de produzir resultado:

- Deixar a função entrar em *ciclo*, para que nunca termine.
- Deixar que seja gerada uma excepção.

Na prática, como se deve programar uma função parcial?

Programa pequeno

Num programa pequeno programa-se uma função parcial *pensando pela positiva*, sem perder muito tempo a ver no que acontecerá nos casos proibidos. Isto funciona bem no OCaml porque a linguagem trata automaticamente os argumentos proibidos.

É esta a atitude subjacente às duas funções atrás definidas. Confirmemos que, efectivamente, elas não produzem resultado ao serem aplicadas a argumentos inválidos.

```
# fact (-1)
Stack overflow during evaluation (looping recursion?).
```

Neste caso tudo bem. A função entrou num ciclo infinito e acaba por abortar por falta de memória. Não produziu resultado.

```
# maxList [];;
Uncaught exception: Match_failure ("", 276, 330)
```

Neste caso também está tudo bem. A função não consegue emparelhar a lista vazia pelo que aborta. Não produziu resultado. (Quando se compila a função, ela dá um *warning*, mas isso não é problema desde que percebamos a sua razão de ser).

Programa grande

Num programa grande, para eliminar possíveis fontes de confusão, é muito importante que todas as situações anormais sejam imediatamente detectadas e claramente assinaladas. Consegue-se isso tratando explicitamente os argumentos proibidos e produzindo mensagens de erro claras.

De acordo com esta atitude, as funções anteriores seriam assim reescritas:

```
let rec fact n =  (* pre: n >= 0 *)
    if n = 0 then 1
    else if n > 0 then n * fact (n-1)
    else raise (Invalid_argument "fact: numero negativo")
;;

let rec maxList l =  (* pre: l <> [] *)
    match l with
```

```
          [] -> raise (Invalid_argument "maxList: lista vazia")
        | [x] -> x
        | (x::xs) -> max x (maxList xs)
;;
```

Estas funções também não produzem resultado, se aplicadas a argumentos inválidos.

Resumo

As funções que se preocupam em tratar explicitamente argumentos proibidos têm duas vantagens:

- Produzem mensagens de erro claras;
- Não geram warnings, quando compiladas.

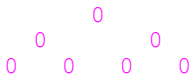
Mas têm duas desvantagens:

- São maiores e mais trabalhosas de escrever;
- Fazer-nos perder tempo com casos anormais que geralmente nunca acontecem (e se acontecerem já são tratados automaticamente pela linguagem).

Mais problemas sobre árvores (alguns complicados)

Problema 1

Função `zeros: int -> int tree` para criar uma árvore do tipo `int tree` com n níveis, cheia de zeros. Por exemplo, para n = 3 a árvore produzida deverá ser:



Solução indutiva directa:

```
type 'a tree = Nil | Node of 'a * 'a tree * 'a tree;;

let rec zeros n =
  if n = 0 then Nil
  else Node(0, zeros (n-1), zeros (n-1))
;;
```

Problema 2

Função `zerosx: int -> int -> int tree` para criar uma árvore semelhante à do problema anterior, mas com uma pequena diferença: a i-ésima folha fica com o número "1" (começando a contagem das folhas em zero). Por exemplo, para n = 3 e i = 2 a árvore produzida deverá ser:

Solução indutiva directa:

```
let rec leaves t =
    match t with
        Nil -> 0
      | Node(x,Nil,Nil) -> 1
      | Node(x,l,r) -> leaves l  + leaves r
;;

let rec changeLeaf t i v =
    match t with
       Nil -> Nil
     | Node(x,Nil,Nil) -> Node((if i = 0 then v else x),Nil,Nil)
     | Node(x, l, r) ->
                  Node(x, changeLeaf l i v, changeLeaf r (i-(leaves l)) v)
;;

let zerosx n i =
    let t = zeros n in
        changeLeaf t i 1
;;
```

Discussão

A função `changeLeaf` tem de lidar com folhas. Por isso, não admira que o padrão que representa as folhas - `Node(x,Nil,Nil)` - tenha de ser usado.

Como habitualmente, na função `changeLeaf` faz-se redução de problemas a problemas mais simples. A parte mais complicada da função reside no seguinte problema: se uma folha tem índice `i` na árvore completa, qual é o índice dessa folha em cada uma das duas subárvores? Resposta: o índice na subárvore da esquerda não muda; para obter o índice na subárvore da direita é preciso subtrair o número de folhas da árvore da esquerda (trata-se duma espécie de transformação de coordenadas).

Problema 3

Função `ints: int -> int tree` para criar uma árvore do tipo `int tree` com n níveis, preenchida com números inteiros em sequência a começar em 1. Por exemplo, para n = 3 a árvore produzida deverá ser:

Solução indutiva indirecta:

```
let rec intsAux n r =
  if n = 0 then Nil
  else Node (r, intsAux (n-1) (2*r), intsAux (n-1) (2*r+1))
;;

let ints n =
  intsAux n 1
;;
```

Discussão

Neste problema desistimos de usar o método indutivo directamente. Começamos a pensar indutivamente, a partir da árvore `ints (n-1)` mas, ao fim de algum tempo, desistimos de encontrar maneira de produzir a árvore `ints n` a partir dela.

Assim vamos procurar uma solução baseada na seguinte propriedade: se um nó tem o valor X, o seu filho imediato da esquerda tem o valor 2*X e o seu filho imediato da direita tem o valor 2*X+1. A solução é uma função em que uma certa informação circula de "cima para baixo"; mais concretamente, em que cada pai que diz a cada filho (usando o parâmetro `r`) qual é o valor que este deve ter.

Voltando a olhar para o que fizémos, vemos que, afinal, a função `intsAux` corresponde a uma solução indutiva directa para um problema um pouco diferente do original. O *novo* problema envolve uma regra que relaciona directamente pais com filhos, e a expressão natural deste problema conduz à introdução dum segundo argumento.

O nosso problema original acabou por ser resolvido escrevendo uma função não recursiva que tira partido da solução do outro problema.

Problema 4

Outra solução para o problema anterior.

Solução indutiva directa, mas pouco óbvia

```
let rec addt t n =
    match t with
       Nil -> Nil
    | Node(x,l,r) -> Node(x+n, addt l (2*n), addt r (2*n))
;;

let rec ints n =
  if n = 0 then Nil
  else Node(1, addt (ints (n-1)) 1, addt (ints (n-1)) 2)
;;
```

Discussão

Afinal sempre existe uma solução indutiva directa para o nosso problema! Esforçando-nos um pouco mais, lá conseguimos descobri-la...

Realmente, é possível produzir a árvore `ints n` a partir da árvore `ints (n-1)`. A árvore `ints (n-1)` pode ser transformada por duas vezes (usando a função `addt`), e os resultados constituem as duas subárvores da árvore `ints n`.

Teórica 10

Método dos parâmetros de acumulação

O método dos parâmetros de acumulação permite programar funções recursivas **enfatisando os aspectos operacionais**. Ou seja, **permite programar em ML usando raciocínios imperativos**.

Cada problema programa-se usando duas funções: uma função principal não recursiva, e uma função auxiliar recursiva.

- A **função auxiliar** tem todos os parâmetros da função principal e ainda um parâmetro extra chamado de **parâmetro de acumulação**: esse parâmetro **acumula, ao longo de sucessivas chamadas recursivas, o resultado final**. O resultado final é retornado nos *casos base* da função, quando esta termina a sua execução.
- A **função principal** limita-se a invocar a função auxiliar, passando para esta todos os seus parâmetros mais o valor inicial para o parâmetro de acumulação (muitas vezes esse valor é 0 ou []).

Exemplo1: Programação da função `len` usando o método dos parâmetros de acumulação

Comecemos por identificar a função principal e a função auxiliar:

- A função principal, a que vamos chamar `len`, tem um parâmetro de tipo lista e um resultado de tipo inteiro:

```
len: 'a list -> int
```

- A função auxiliar, a que vamos chamar `lenA`, tem um parâmetro normal de tipo lista e um parâmetro de acumulação de tipo inteiro (visto o resultado final da função `len` ser de tipo inteiro). Eis o tipo de `lenA`:

```
lenA: 'a list -> int -> int
```

A função principal `len` não é recursiva e limita-se a passar o seu argumento para a função `lenA`, além de fornecer um valor inicial para o parâmetro de acumulação:

```
let len list = lenA list 0 ;;
```

A função auxiliar `lenA` é recursiva e define-se por casos, como é habitual nas funções recursivas. Eis a sua definição:

```
let rec lenA l a =
    match l with
        [] -> a
      | x::xs -> lenA xs (a+1)
;;
```

Existem três fases distintas no funcionamento desta função, e repare como esta explicação tem um sabor imperativo:

- **Inicialização:** A função auxiliar é invocada com um valor inicial no parâmetro de acumulação.
- **Processamento:** Ao longo de sucessivas chamadas recursivas, o caso geral da função `lenA` é activado e a função conta mais um elemento da lista.
- **Finalização:** Na última invocação de `lenA`, o caso base da função é activado e a função retorna o valor do parâmetro de acumulação, o qual conterá na altura o número total de elementos da lista.

Exemplo de avaliação

```
len [10;20;30]
= lenA [10;20;30] 0
= lenA [20;30] 1
= lenA [30] 2
= lenA [] 3
= 3
```

Exemplo2: Função `map`

Usando o método indutivo:

```
let rec map f l =
```

```
    match l with
        [] -> []
      | x::xs -> (f x)::map f xs
;;
```

Usando o método dos parâmetros de acumulação:

```
let rec mapA f l a =
    match l with
        [] -> a
      | x::xs -> mapA f xs (a@[f x])
;;
let map f list = mapA f list [] ;;
```

Repare que a função `map` programada usando o método dos parâmetros de acumulação fica **menos eficiente** do que se fosse programada usando o método indutivo.

Exemplo3: Função `rev`

Usando o método indutivo:

```
let rec rev l =
    match l with
        [] -> []
      | x::xs -> (rev xs)@[x]
;;
```

Usando o método dos parâmetros de acumulação:

```
let rec revA l a =
    match l with
        [] -> a
      | x::xs -> revA xs (x::a)
;;
let rev list = revA list [] ;;
```

A função `rev` programada usando o método dos parâmetros de acumulação fica **mais eficiente** do que se fosse programada usando o método indutivo.

Exemplo4: Função de Fibonnaci

Usando o método indutivo:

```
let rec fib n =
    match n with
        0 -> 1
      | 1 -> 1
      | x -> fib (x-1) + fib (x-2)
;;
```

Usando o método dos parâmetros de acumulação:

```
let rec fibA n a b =
    match n with
       0 -> a
     | 1 -> b
     | x -> fibA (n-1) b (a+b)
;;
let rec fib n = fibA n 1 1 ;;
```

A função `fib` programada usando o método dos parâmetros de acumulação fica **mais eficiente** do que se fosse programada usando o método indutivo.

Comparação entre o método indutivo e o método dos parâmetros de acumulação

Existem três fortes razões que tornam o método indutivo preferível ao método dos parâmetros de acumulação:

- O método indutivo produz funções mais fáceis de compreender [por exemplo não faz muito sentido de dizer que lenA [] 5 = 5, mas já faz todo o sentido dizer que len [] = 0]. Realmente, dada uma função programada usando o método dos parâmetros de acumulação, para se perceber o que essa função faz é

necessário executar mentalmente a função (ou seja, temos de ser nós a fazer de computador).

- No método indutivo há a vantagem de se considerem apenas as propriedades lógicas do problema a resolver. No método dos parâmetros de acumulação temos também de considerar aspectos de natureza operacional, os quais conduzem muitas vezes a soluções mais complicadas.
- Muitas vezes as funções programadas usando o método indutivo são mais eficientes do que as funções programadas usando o método dos parâmetros de acumulação. Compare a versão indutiva da função `map` com a versão de acumulação e repare que a versão indutiva tem complexidade linear enquanto que a versão de acumulação tem complexidade quadrática (devido ao uso da operação @).

No entanto, em certas circunstâncias, o uso do método dos parâmetros de acumulação pode ser recomendável ou mesmo obrigatório. Isto acontece se...

- [Obrigatório] Se o problema envolver uma noção de "estado corrente". Neste caso temos de usar parâmetros de acumulação para representar esse estado.
- [Recomendável] Se o enunciado do problema for baseado em informação operacional. Se o enunciado do problema contiver a descrição de muitos algoritmos imperativos, pode ser mais simples de usar directamente o método dos parâmetros de acumulação para codificar directamente esses algoritmos.
- [Recomendável] Se o método dos parâmetros de acumulação conduzir a uma solução muito mais eficiente. Compare as duas versões da função `fib`, a indutiva e a de acumulação, e note que a de acumulação tem complexidade linear enquanto que a indutiva tem complexidade exponencial.

NOTA:

- Diz-se que uma função tem **complexidade temporal linear** se o seu tempo de avaliação for proporcional à dimensão do seu argumento.
- Diz-se que uma função tem **complexidade temporal exponencial** se o seu tempo de avaliação é proporcional à exponencial da dimensão do seu argumento.

[Na cadeira de AED1 será dada muita atenção ao estudo da complexidade temporal e da complexidade espacial de algoritmos.]

Teórica 11

Funções recursivas

Uma função recursiva bem definida faz sempre uma *análise de casos* sobre os seus parâmetros. Por exemplo, a função recursiva "fib" considera três casos relativamente ao seu parâmetro "n":

```
let rec fib n =
     if n = 0 then 1
     else if n = 1 then 1
     else fib (n-1) + fib (n-2)
;;
```

Existem sempre dois géneros de casos que uma função recursiva tem sempre de tratar: *casos base* e *casos gerais*. Os **casos base** são aqueles que não conduzem, nem directa nem indirectamente, a chamadas recursivas da função; os **casos gerais** são aqueles que conduzem, directa ou indirectamente, a chamadas recursivas da função. A função "fib" trata dois casos base, n=0 e n=1, e trata um caso geral, n>=2.

Toda a função recursiva deve tratar pelo menos um caso base e um caso geral. Além disso todas as chamadas recursivas que estão associadas aos casos gerais devem corresponder **problemas mais simples** do que o problema que a função, no seu todo, resolve (*problema mais simples* significa *problema mais próximo de algum dos casos base*). Apenas este conjunto de condições garante terminação.

Usando o método indutivo

O **método indutivo**, introduzido na aula teórica 6, ajuda a programar os casos gerais das funções recursivas. Concretamente, dá alguma orientação sobre como fazer a *redução de problemas a problemas mais simples*.

Quando se usa o método indutivo, **devem-se ter em mente** apenas as propriedades lógicas do problema a resolver. **Não se devem ter em mente** quaisquer preocupações relacionadas com as propriedades operacionais da função que está a ser programada. Misturar do uso do método indutivo com preocupações de natureza operacional é uma receita para o desastre.

De qualquer forma, depois de programada a função, já não há problema em tentar compreendê-la operacionalmente. Pode mesmo valer a pena escrever uma avaliação da função para se ganhar confiança nela. Exemplo: avaliação de (len [1;5;3;8]):

```
len [1;5;3;8] =
    1 + len [5;3;8] =
    1 + 1 + len [3;8] =
    1 + 1 + 1 + len [8] =
    1 + 1 + 1 + 1 + len [] =
    1 + 1 + 1 + 1 + 0 =
    4
```

Categorias de funções recursivas

Vamos agora estudar 4 categorias de funções recursivas, todas programadas usando o método indutivo: funções de tipo 1, de tipo 2, de tipo 3 e de tipo 4.

Na avaliação da qualidade duma função, um dos critérios que surge em primeiro lugar é o da **facilidade em compreender a função**. As funções de tipo 1 e 2 tendem a ser as mais simples de perceber, mas nem todos os problemas podem ser resolvidos directamente usando apenas funções de tipo 1 e 2.

Funções de tipo 1

Quando se tenta resolver um problema, convém começar por procurar construir uma função de tipo 1 já que estas são as mais fáceis de escrever e compreender, na maioria dos casos.

As **funções de tipo 1** são programadas usando o método indutivo e caracterizam-se pela seguinte propriedade:

- A estrutura da função baseia-se num dos esquemas rígidos predefinidos, indicados a seguir.

Nestes esquemas, o PONTO DE PARTIDA PARA COMEÇAR A RACIOCINAR aparece sublinhado.

Inteiros

```
let rec f n =
    if n = 0 then ...
    else ... f (n-1) ...
```

```
;;
```

Listas

```
let rec f l =
    match l with
        [] -> ...
      | x::xs -> ... f xs ...
;;
```

Árvores binárias

```
let rec f t =
    match t with
        Nil -> ...
      | Node(x,l,r) -> ... f l ... f r ...
;;
```

Ficheiros

```
let rec f ci =
   try
       let s = input_line ci in
           ... f ci ...
   with End_of_file -> ...
;;
```

Strings

```
let cut s = (String.get s 0, String.sub s 1 ((String.length s)-1)) ;;
let rec f s =
    if s = "" then ...
    else
        let (x,xs) = cut s in
            ... f xs ...
;;
```

Exemplos de funções de tipo 1: fact, len, append, rev, belongs, union, power, height, size, zeros, treeToList, balanced, subTrees, etc. (em suma, a maioria das funções estudadas até ao momento).

Mais exemplos de funções de tipo 1:

```
(* stringAsList: converte string em lista de caracteres *)
    let rec stringAsList s =
        if s = "" then []
        else
            let (x,xs) = cut s in
                x::stringAsList xs
    ;;

(* sortList: ordena lista *)
    let rec insOrd v l =
        match l with
            [] -> [v]
          | x::xs ->
              if v <= x then v::x::xs
              else x::insOrd v xs
    ;;
    let rec sortList l =
        match l with
            [] -> []
          | x::xs ->
              insOrd x (sortList xs)
    ;;

(* halfHalf: reparte os elementos duma lista por duas listas, alternadamente *)
    let rec halfHalf l =
        match l with
            [] -> ([],[])
          | x::xs ->
                let (a,b) = halfHalf xs in
```

```
                (x::b,a)
    ;;
```

Funções de tipo 2

Quando se tenta escrever uma função de tipo 1, por vezes descobre-se que é necessário ou conveniente fazer alguns pequenos ajustamentos ao esquema rígido de base. Desta forma somos levados a inventar uma função de tipo 2.

As **funções de tipo 2** são programadas usando o método indutivo e caracterizam-se pelas duas seguintes propriedades:

- Os argumentos das chamadas recursivas são subpadrões dos padrões que representam os argumentos da função.
- Fogem aos esquemas rígidos das funções de tipo 1.

Nas funções de tipo 2, o PONTO DE PARTIDA PARA COMEÇAR A RACIOCINAR envolve um pouco de descoberta. Mas, em geral, é fácil de descobrir esse ponto de partida pois os argumentos das chamadas recursivas são subpadrões dos padrões que representam os argumentos da função.

Exemplos de funções de tipo 2: maxList, fall, fib.

Mais dois exemplos de funções de tipo 2:

```
(* halfHalf: reparte os elementos duma lista por duas listas, alternadamente
   esta versão é programada com base na ideia de processar os elementos da lista dois a dois
*)
    let rec halfHalf l =
        match l with
            [] -> ([],[])
          | [x] -> ([x],[])
          | x::y::xs ->
                let (us,vs) = halfHalf xs in
                    (x::us, y::vs)
    ;;

(* addEvenPos: soma todos os elementos duma lista que estão em posições pares (0, 2, 4, ...)
*)
    let rec addEvenPos l =
        match l with
            [] -> 0
          | [x] -> x
          | x::_::xs ->
                x + addEvenPos xs
    ;;
```

Teórica 12

Categorias de funções recursivas (cont.)

Funções de tipo 3

As funções de tipo 3 surgem geralmente quando se tenta resolver um problema aplicando uma estratégia previamente pensada, em vez de se tentar encontrar uma solução baseada nos esquemas predefinidos que caracterizam as funções de tipo 1.

As **funções de tipo 3** são programadas usando o método indutivo e caracterizam-se pelas seguinte propriedade:

- A chamada recursiva envolve expressões que *não são* subpadrões dos padrões que representam os argumentos da função. As expressões que constituem os argumentos da chamada recursiva são *calculados*.

Escolher programar uma função de tipo 3 em vez duma de tipo 1 ou 2 é, quase sempre, complicar desnecessariamente. Além disso, as funções de tipo 3 são mais difíceis de perceber do que as funções de tipo 1 ou 2.

Em todo o caso, ocasionalmente surge uma boa razão para se escrever uma função de tipo 3: por exemplo, o aumento da eficiência, se for caso disso.

Exemplos de funções de tipo 3:

```
(* sillyLen: determina comprimento de lista de forma tresloucada *)
    let rec sillyLen l =
        match l with
            [] -> 0
          | [x] -> 1
          | list ->
              let (us,vs) = halfHalf list in
                  sillyLen us + sillyLen vs
    ;;

(* minSort: ordena lista usando o algoritmo de selecção directa *)
    let rec removeFromList v l =
        match l with
            [] -> []
          | x::xs ->
                if x = v then xs
                else x::removeFromList v xs
    ;;
    let rec minList l =
        match l with
            [x] -> x
          | x::xs -> min x (minList xs)
    ;;
    let rec minSort l =
        match l with
            [] -> []
          | list ->
              let m = minList list in
                  m::minSort (removeFromList m list)
    ;;

(* quickSort: ordena lista eficientemente (reduz-se o tratamento de
              uma lista ao tratamento de duas secções dessa lista) *)
    let rec partition p l =
        match l with
            [] -> ([],[])
          | x::xs ->
              let (a,b) = partition p xs in
                  if x <= p then (x::a,b) else (a, x::b)
    ;;
    let rec quickSort l =
        match l with
            [] -> []
          | x::xs ->
              let (us,vs) = partition x xs in
```

```
                    (quickSort us) @ [x] @ (quickSort vs)
    ;;
```

(Repare como diferentes formas de reduzir um problema conduziram a diferentes funções recursivas: funções "sortList", "minSort" e "quickSort".)

Teórica 13

Categorias de funções recursivas (cont.)

Funções de tipo 4

Existem problemas que não podem ser resolvidos directamente usando o método indutivo. Nestes casos surge a necessidade de escrever uma função de tipo 4.

Que fazer quando surge um problema que não se consegue resolver usando o método indutivo de forma directa?

- A solução é a seguinte: INVENTA-SE um novo problema que já se possa resolver usando o método indutivo e que ajude a resolver o problema original.

Uma **função de tipo 4** é uma função não recursiva que se define à custa de funções auxiliares, que são geralmente funções recursivas de tipo 1, 2, ou 3.

Exemplos de funções de tipo 4:

```
(* prime: determina de um inteiro é ou não primo *)

    let rec hasDiv n a z =
        if a > z  then false
        else (n mod a = 0) or (hasDiv n (a+1) z)
    ;;
    let prime n =
       not (hasDiv n 2 (n-1))
    ;;

(* width: determina a largura duma árvore binária *)

    type 'a tree = Nil | Node of 'a * 'a tree * 'a tree ;;

    let rec maxList l =
        match l with
            [x] -> x
          | x::xs -> max x (maxList xs)
    ;;
    let rec sumLists l1 l2 =
        match l1,l2 with
            [],l -> l
          | l,[] -> l
          | x::xs, y::ys -> (x+y)::sumLists xs ys
    ;;
    let rec levels t =
        match t with
            Nil -> []
          | Node(x,l,r) -> 1::sumLists (levels l) (levels r)
    ;;
    let width t =
         if t = Nil then 0
         else maxList (levels t) ;;
```

Final da primeira parte da cadeira, relativa ao paradigma funcional

O que fizemos?

- Estudámos e treinámos o uso de técnicas recursivas/indutivas na resolução de problemas;
- A mudança de paradigma permitiu alargar horizontes e enriquecer a paleta das técnicas de programação;
- Absorvemos um pouco da "cultura da programação funcional". A programação funcional sempre teve numerosos praticantes e defensores, e consequentemente uma grande influência no curso da história da informática;
- Analisámos alguns conceitos relativos linguagens de programação de forma independente de qualquer linguagem particular, com vista a compreender um pouco melhor as linguagem de programação na sua diversidade. Concretamente, os conceitos analisados (com variável grau de detalhe) foram:
    - sistemas de tipos;

- inferência de tipos;
- sobreposição (overloading);
- tipo produto;
- tipo soma;
- funções e funções de ordem superior;
- polimorfismo (paramétrico);
- efeitos laterais;
- tratamento de excepções.

Teórica 14

Teste sobre o trabalho de ML.

Teórica 15

Introdução

A segunda parte da cadeira de LP1 é dedicada ao paradigma orientado pelos objectos e tem a duração de 6 semanas. Analisamos os conceitos essenciais do paradigma, justifica-se o interesse desses conceitos, e discutimos a melhor forma de organizar os programas usando esses conceitos.

Existem inúmeras linguagens que poderíamos usar como veículo para estudar o paradigma orientado pelos objectos: Smalltalk, Ocaml, Java, C++, C#, Beta, Objective-C, etc., etc. Em algumas das edições anteriores de LP1 foi adoptada a linguagem Smalltalk, uma linguagem orientada pelos objectos *pura* e muito simples. Na corrente edição, será usada a linguagem **Java**, uma linguagem orientada pelos objectos *quase pura* e não demasiado complicada. A linguagem Java resulta, largamente, duma simplificação da linguagem C++.

Para acompanhar a segunda parte de LP1, é faz falta ter passado pela cadeira de P2. Caso não tenha tido essa cadeira, é indispensável estudar imediatamente os capítulos 2, 3 e 4 no livro que se encontra na página da bibliografia de LP1.

O C++ não é para esquecer!!

Agora que você vai aprender Java, será que é altura para por de lado o C++? Afinal, o C++ é bastante mais complicado do que o Java...

A resposta é um rotundo NÃO! A linguagem C++ é uma linguagem muito importante, conhecida e usada pela maioria dos programadores profissionais. Aproveite pois esta oportunidade, não só para aprender Java, mas também para (indirectamente) melhorar o seu entendimento do C++.

Paradigma orientado pelos objectos

É um paradigma de programação no qual os dados, denominados por **objectos**, são entidades activas que interagem entre si através de troca de **mensagens**. Um objecto pode reagir a uma mensagem de várias formas: alterando o seu estado interno; criando uma cópia modificada de si próprio; criando outros objectos; enviando mensagens a outros objectos ou a si próprio.

- Abreviadamente, um **objecto** pode ser definido como um elemento de dados **activo** e com **individualidade própria**.

No paradigma orientado pelos objectos, um programa em execução consiste numa comunidade de objectos trocando mensagens entre si. **Quem controla a computação são, pois, os objectos!** O código executável subordina-se a estes, pois só se escreve código para especificar como é que um objecto reage a mensagens provenientes do exterior. [Numa linguagens orientada pelos objectos *pura* não se permite a escrita de funções ou procedimentos autónomos.]

Paradigmas procedimentais

Paradigmas procedimentais, ou *paradigmas orientados pelos procedimentos e funções*, é o nome genérico que se dá aos paradigmas imperativo e funcional. Nestes paradigmas, os dados, designados por **valores**, são entidades passivas, livremente manipuladas e transferidas entre as diversas partes do programa.

- Abreviadamente, um **valor** pode ser definido como um elemento de dados **inactivo** e sem **individualidade própria**.

Nos paradigma procedimentais, um programa em execução consiste numa comunidade de funções e procedimentos que se invocam mutuamente, trocando valores entre si. **Quem controla a computação são pois os procedimentos e funções!** Os dados subordinam-se a estes.

Objectos

Um **objecto** é uma entidade recursiva, com acesso a si própria através dum nome predefinido (**this** em Java e C++, **self** em Smalltalk e Self). Cada objecto caracteriza-se pelos seguintes elementos:

- Uma **identidade**, que torna o objecto distinguível de qualquer outro objecto (em Java é a operação `=` que permite verificar se dois objectos são o mesmo).
- Um conjunto de **variáveis de instância**, *públicas* e *privadas*, que, no seu conjunto, definem o chamado **estado do objecto**.
- Um conjunto de **métodos de instância**, *públicos* e *privados*, que determinam o comportamento do objecto, ou seja, a forma como ele reage a *mensagens* recebidas do exterior (cada mensagem recebida por um objecto activa neste objecto um determinado método público).

Ocultação da representação

Geralmente, recomenda-se que os objectos incluam apenas variáveis de instância *privadas*, com o objectivo de se **ocultar a representação interna do objecto**. As vantagens desta ocultação são diversas:

- Impede-se a corrupção dos objectos a partir do exterior.
- Simplifica-se a gestão dos objectos (já que a diversidade das interacções possíveis entre objectos se reduz).
- Simplifica-se a depuração dos programas.
- Permite-se que a representação interna dum objecto seja futuramente alterada, sem com isso se comprometer o resto do programa.

Criação dos objectos

Nas linguagens *baseadas em classes* - tal como as linguagens Java, C++, Smalltalk - os objectos são criados usando **classes**.

Nas linguagens *baseadas em protótipos* - tal como a linguagem Self - a maioria dos objectos são criados por duplicação de objectos existentes. Os objectos *iniciais*, ditos **protótipos**, definem-se separadamente por meio duma construção sintáctica própria.

Classes

Uma **classe** é uma **entidade geradora de objectos** (é, digamos, uma *maternidade de objectos*). Os objectos gerados por uma classe são designados por **instâncias** dessa classe.

Naturalmente que uma classe precisa de especificar, nos mais ínfimos detalhes, os objectos que gera. Por isso também se diz que uma classe **implementa** os objectos que gera.

A linguagem Java

O Java é uma **linguagem orientada pelos objectos**, **baseada em classes** e **tipificada**. Contudo, do ponto de vista da orientação pelos objectos, não é uma linguagem completamente *pura*... Só por uma razão: em Java, os dados de tipos primitivos - *int*, *boolean*, *char*, etc. - são *valores* e não *objectos*.

O Java também suporta concorrência, mediante a utilização de métodos *synchronized* e de objectos da classe predefinida *Thread*.

O Java resulta, em grande parte, duma simplificação do C++ (concretamente da primeira versão desta linguagem). O Java deixa de fora os seguintes elementos: apontadores, gestão explícita de memória (ou seja, a primitiva *delete*), preprocessador (#define, etc.), typedef, union, struct, funções globais, variáveis globais, funções-membro não-virtuais, goto, ficheiros ".h", sobreposição de operadores, herança múltipla, conversões automáticas definidas pelo programador.

Mas, relativamente ao C++, o Java adiciona dois importantes conceitos: interfaces e packages.

Não obstante derivar do C++, o Java adopta a nomenclatura da linguagem Smalltalk (que também é a nomenclatura padrão do paradigma orientado pelos objectos). Eis alguns termos da nomenclatura do Java: *método* (em vez de *função-membro*), *envio de mensagem* (em vez de *aplicação de função*), *subclasse* (em vez de *classe derivada*), *superclasse* (em vez de *classe base*).

O Java é uma linguagem interpretada, portanto **portável** a nível binário. Também é uma linguagem razoavelmente **eficiente**: as implementações antigas eram ineficientes, mas a partir da versão 1.2 a eficiência do Java deu um salto em frente. No entanto, deixa ainda algo a desejar, relativamente ao C++.

A descrição linguagem Java (versão 1.2) está disponível na página da cadeira, na coluna da esquerda, no item "Java Language 1.2".

A plataforma Java

A linguagem Java, sendo relativamente pequena, herda muito do seu sabor e poder duma grande biblioteca de classes predefinidas, chamada **plataforma Java** (*aka* **core Java APIs**, *aka* **Java runtime environment**). Esta biblioteca é robusta, intuitiva e bem desenhada. Tem vindo a crescer e a amadurecer com cada nova versão do sistema.

Actualmente, a plataforma Java é tão vasta que os programadores já não precisam de aceder directamente aos serviços do sistema operativo. Assim, a plataforma Java apresenta-se como uma **plataforma de desenvolvimento universal**. Sobre ela correm todos os programas Java, independentemente do sistema operativo subjacente. [É por isso que a Microsoft vê o Java como uma ameaça.]

A documentação da plataforma Java está disponível na página da cadeira, na coluna da esquerda, no item "Packages & Classes".

Versões de Java (Standard Edition)

```
Java SDK 1.0b2 Dez1995
Java SDK 1.0 23Jan1996
Java SDK 1.1 18Feb1997
Java SDK 1.2 4Dez1998    (Implementa a chamada "especificação 2.0")
Java SDK 1.3 8Maio2000
Java SDK 1.4 2001
Java SDK 1.5 30Set2004  (Adiciona diversas novidades úteis, principalmente classes genéricas e
autoboxing)
```

Ao longo da evolução do Java, a linguagem tem mudado pouco. Nas versões 1.1, 1.4 e 1.5, foram introduzidas algumas extensões, mas em número relativamente reduzido.

Pelo contrário, a plataforma Java tem vindo a crescer exponencialmente.

Exemplo de programa em Java

Definição dum tipo-objecto, "IntStack", e duma sua implementação, "IntStackClass":

```
// Este é um tipo-objecto:

    interface IntStack {
        int Capacity() ;
        int Size() ;
        boolean Empty() ;
        boolean Full() ;
        int Top() ;
        void Push(int i) ;
        int Pop() ;
    }

// Esta é uma implementação do tipo-objecto anterior:

    class IntStackClass implements IntStack {
        private int[] elems ;
        private int top ;

        IntStackClass(int maxSize) {
            elems = new int[maxSize] ;
            top = 0 ;
        }
        IntStackClass() {
            this(100) ;
        }

        private void ErrorMesg(String s) {
            throw new Error(s) ;
        }
        public int Capacity() {
            return elems.length ;
        }
        public int Size() {
            return top ;
        }
        public boolean Empty() {
            return Size() == 0 ;
        }
        public boolean Full() {
            return Size() == Capacity() ;
        }
        public int Top() {
            if( Empty() ) ErrorMesg("Pop: Stack is empty") ;
            return elems[top-1] ;
        }
        public void Push(int v) {
            if( Full() ) ErrorMesg("Push: Stack is full") ;
            elems[top++] = v ;
        }
        public int Pop() {
            if( Empty() ) ErrorMesg("Pop: Stack is empty") ;
            return elems[--top] ;
        }
```

```
    }

// Esta é uma classe de teste:

    class TestStack {
        public static void main(String[] args) {
            System.out.println("Welcome to the TestStack class!!") ;

            IntStack s = new IntStackClass() ;
            System.out.println(s.Capacity()) ;
            s.Push(123) ;
            System.out.println(s.Pop()) ;
        }
    }
```

O ambiente de desenvolvimento JDK

O ambiente de desenvolvimento JDK (Java Developer's Kit) consiste num conjunto de ferramentas que ajudam a **desenvolver**, **testar**, **documentar** e **executar** programas em Java. O JDK é gratuitamente disponibilizado pela empresa Sun e existem versões para os mais variados sistema: Linux, Windows, MacOS, etc. Usaremos nas nossas aulas práticas a versão para Linux. Na página da cadeira, dentro do item "Bibliografia & Links", encontram-se disponíveis as versões para Linux e Windows (para instalar em casa).

Ferramentas do JDK

javac	Compilador
java	Executor de *programas independentes*
javadoc	Gerador de documentação
appletviewer	Executor de *applets*
jdb	Debugger
javap	Disassembler
javah	Gerador de *header files* para métodos nativos em C

Como se compilam programas em Java?

- Os programas em Java compilam-se por meio do comando **javac**. Por exemplo: `javac Sets.java`. O comando **javac** permite compilar tanto **programas independentes** como **applets**.
- Os ficheiros compilados pelo comando **javac** têm obrigatoriamente a extensão ".java".
- Quando activado, o compilador cria na directoria corrente diversos ficheiros com a extensão ".class", um por cada classe ou interface definidas nos ficheiros fonte. Os ficheiros ".class" contêm código executável, num formato padrão chamado **bytecode**.

Como se executam programas em Java?

- Se se tratarem de **programas independentes**, executam-se usando o comando **java**. Por exemplo: `java TestSets`. Neste exemplo, "TestSets" refere-se ao ficheiro "TestSets.class" na directoria corrente: repare que o comando `java` não admite a escrita da extensão ".class" [*Manias!*].
- Se se tratarem de **applets**, executam-se usando o comando **appletviewer** ou então por intermédio dum browser WWW.
- A componente de software que permite executar ficheiros de de *bytecode* chama-se **Java Virtual Machine**. Está incorporada no comando **java**, no comando **appletviewer** e na maioria dos browsers WWW modernos.

Como se documentam programas em Java?

- O gerador de documentação, **javadoc** converte ficheiros ".java" em ficheiros HTML, visualizáveis usando qualquer browser WWW. Estes ficheiros HTML incluem documentação sobre as classes, interfaces, métodos, variáveis e excepções definidos nos ficheiros fonte e ainda o conteúdo de comentários especiais, da forma /** ... */, escritos pelo programador. [Tudo o que se encontra no item "Packages & Classes" da coluna da esquerda da página da cadeira, foi gerado pelo comando **javadoc**.]

Teórica 15a

Tipos e linguagens tipificadas

Um **tipo** representa uma colecção de elementos de dados e tem um conjunto de *literais* associados e de *operações* associadas. Por exemplo, em Pascal o tipo `integer` representa o conjunto do inteiros, tem associados os literais 0, 1, -1, 2, -2, etc., e ainda as operações `+`, `-`, `*`, `div`, `mod`, `succ`, etc.

Actualmente a maioria das linguagens são **tipificadas** (e.g. Pascal, C, C++, Eiffel, Java, ML). Isto significa que, nessas linguagens, todas as variáveis, parâmetros e expressões têm um tipo que é conhecido em tempo de compilação.

Para que servem os tipos? Para o seguinte:

- Permitem a detecção antecipada de erros, em tempo de compilação.
- Ajudam a documentar os programas.
- São úteis na optimização de código gerado.

Tipos primitivos e valores primitivos

Em Java, os **tipos predefinidos** são: `boolean`, `char`, `byte`, `short`, `int`, `long`, `float` e `double`. Estes tipos têm literais predefinidos associados (`3.14`, `'a'`, `false`, `true`, `21056`, `0x234`, etc.) e operações predefinidas associadas (`+`, `-`, `*`, `/`, `&&`, `||`, `>>`, etc.).

Em Java chamam-se **valores primitivos** aos elementos que pertencem aos tipos primitivos. Atenção, os valores primitivos não são objectos: são valores, passivamente manipulados.

Alguns detalhes:

- O tipo `boolean` não é considerado numérico. Pelo contrário, o tipo `char` é considerado numérico.
- As operações aritméticas só podem envolver operandos numéricos. As operações lógicas só podem envolver operandos booleanos.
- Os valores de tipo `boolean` são representados usando um simples bit. Os valores de tipo `char` são inteiros de 16 bits sem sinal (standard Unicode). Os valores de tipo `int` são inteiros de 32 bits com sinal. Os valores de tipo `double` são reais de 64 bits (standard IEEE754).

Cast

Existe uma família de operadores unários, da forma `(T)`, que permite efectuar conversões entre tipos primitivos, nos casos em que essa conversão seja legal. Exemplo:

```
int i = 1 + (int)21.056   // sem o cast daria erro de compilação
```

Classes *wrapper* (Classes *embrulho*): representando valores primitivos como objectos

A plataforma Java disponibiliza diversas classes que permitem representar valores primitivos como objectos. Tratam-se das classes: `Boolean`, `Char`, `Byte`, `Short`, `Integer`, `Long`, `Float` e `Double`.

Estas classes são designadas por **wrappers** porque os seus objectos limitam-se a servir de *embrulho* a valores primitivos. Por exemplo, um objecto da classe `Integer` contém lá dentro uma única variável de instância, de tipo `int`, onde se guarda um inteiro.

Estas classes *embrulho* contêm também diversos métodos úteis, todos relacionados com o tipo em causa.

Exemplos

Exemplo de criação de objecto da classe `Integer`:

```
Integer iobj = new Integer(21056) ;
```

Exemplo de acesso ao valor guardado dentro dum objecto da classe `Integer`:

```
int i = iobj.intValue() ;
```

Autoboxing e Autounboxing

O Java 5.0 introduziu conversão automática entre os tipos primitivos e as respectivas classes *wrapper*. Portanto as duas atribuições anteriores podem agora escrever-se, mais simplesmente, assim:

```
Integer iobj = 21056 ;
int i = iobj ;
```

O Java encarrega-se, ele mesmo, de inserir as operações de conversão explícita que apareceram nos exemplos anteriores.

Tipos referência

Em Java há duas espécies de **tipos referência**:

- tipos-array
- tipos-objecto

Todos os tipos não primitivos são tipos referência.

Os elementos de tipos referência têm a particularidade de serem acedidos através de *referências*. Por exemplo, depois de executado o código a seguir, as variáveis "a" e "b" passam a *referir* o mesmo array, ou seja, "a" e de "b" são dois nomes para o *mesmo* array:

```
int[] a = new int[100] ;
int[] b = a ;
```

O valor predefinido **`null`** representa uma referência especial que, convencionalmente, não refere nada. Este valor é compatível com qualquer tipo referência. Por outras palavras, ele pode ser atribuído a qualquer variável dum tipo-referência.

Tipos-array

Em Java, os arrays criam-se dinamicamente usando o operador **`new`**. Os arrays podem ser multi-dimensionais.

O seguinte código declara uma variável e inicializa-a com um array bidimensional de inteiros, criado dinamicamente. Todas as posições do array são automaticamente inicializadas a 0:

```
int[][] arr = new int[100][50] ;
```

A construção **`arr.length`** permite saber qual o tamanho da *dimensão exterior* dum array. Por exemplo, considerando o código anterior, temos:

```
arr.length == 100
arr[0].length == 50
arr[12].length == 50
```

Os índices dos arrays começam em 0, tal como em C++. O acesso às componentes também se processa usando a sintaxe do C++. Eis um exemplo duma atribuição a uma componente dum array bidimensional:

```
arr[10][20] = 5 ;
```

Os arrays podem ser inicializados no ponto da declaração, tal como em C++. Exemplo:

```
String[] daysOfTheWeek =
            {"Sunday", "Monday", "Tuesday", "Wednesday",
             "Thursday", "Friday", "Saturday" } ;
```

Os arrays podem ser duplicados usando o método **`clone`**. Faz-se assim:

```
int[][] arr2 = arr.clone() ;
```

- Os arrays são objectos? Sim, em Java, os arrays são objectos.
- Então, porque razão se estudam os tipos-array separadamente dos tipos-objecto? Em Java, existem infinitos tipos-array distintos, um tipo-array diferente por cada possível tipo de componentes. Este facto obriga a tratar os tipos-array de forma especial e a colocá-los numa categoria à parte.

Assinaturas & Protocolos

Assinaturas

A **assinatura** dum método é constituída pelo nome do método e pelo número, tipo e ordem dos seus parâmetros. O tipo do resultado do método não faz parte da assinatura.

Em Java, os métodos são identificados pelas suas assinaturas. Numa classe podem coexistir vários métodos com o mesmo nome, desde que tenham assinaturas distintas (portanto o Java suporta *sobrecarga*, ou *overloading*, do nome dos métodos).

Protocolos

Chama-se **protocolo** dum objecto ao conjunto das assinaturas dos seus métodos públicos. O protocolo dum objecto determina o reportório das mensagens que esse objecto entende.

Diz-se que um objecto **suporta um dado protocolo** se definir todos os métodos públicos previstos no protocolo (pode ter mais).

Tipos-objecto

Já sabemos que um **tipo** representa um conjunto de elementos de dados. Em particular, um **tipo-objecto** representa um conjunto de objectos.

Tradicionalmente, nas linguagens orientadas pelos objectos tipificadas consideram-se duas *filosofias* relativamente aos tipos-objecto:

- Filosofia dos **tipos-objecto sintácticos** - Cada tipo-objecto é caracterizado por um protocolo. Portanto, neste caso, um tipo-objecto representa o conjunto de todos os objectos que suportam esse protocolo (independentemente da implementação interna dos objectos e do significado das suas operações). Exemplo que linguagem que assume esta filosofia: PolyToil.
- Filosofia dos **tipos-objectos semânticos** - Cada tipo-objecto caracteriza-se por uma implementação completa, ou seja por uma classe: ele representa exactamente o conjunto de todos os objectos gerados por essa classe. Exemplo que linguagem que assume esta filosofia: C++.

Como é em Java?

A linguagem Java suporta uma variante da primeira filosofia:

- Em Java um tipo-objecto sintáctico chama-se **interface**. Uma interface representa todos os objectos que suportam um dado protocolo e que sejam instâncias de classes que declarem explicitamente implementar essa interface (usa-se para isso a palavra `implements`).

Mas a linguagem Java também suporta a segunda filosofia, na sua forma original:

- Em Java, os tipos-objecto semânticos são as **classes**. Isso significa que, em Java, as classes, além de serem mecanismos geradores de objectos, também são tipos.

Exemplo

No exemplo da aula anterior, "IntStack" é um tipo-objecto sintáctico. Assim, uma variável declarada com tipo "IntStack" poderá referir qualquer objecto que suporte o protocolo declarado na interface "IntStack" e cuja classe-mãe declare implementar essa interface - em particular, pode referir um objecto da classe "IntStackClass".

No mesmo exemplo, "IntStackClass" é um tipo-objecto semântico. Uma variável declarada com tipo "IntStackClass" poderá referir apenas objectos gerados por essa classe. [E ainda pelas suas subclasses, mas ainda é cedo para se falar no mecanismo de herança em Java.]

A nossa escolha metodológica

Em Java trabalha-se muito bem usando *apenas* tipos-objecto sintácticos. Mas já não se consegue trabalhar convenientemente usando *apenas* tipos-objecto semânticos. Realmente, a primeira filosofia tende a ser bastante mais flexível do que a segunda.

Nesta cadeira vamos adoptar a filosofia dos tipos-objecto sintácticos, e só essa. Ou seja, no código que escrevermos, trataremos as nossas interfaces como tipos e as nossas classes como implementações. Vamos seguir a seguinte regra:

REGRA: Sempre que quisermos introduzir uma nova variedade de objectos, procedemos da seguinte forma:

1. Primeiro escrevemos a interface que captura o protocolo pretendido para os objectos.
2. Depois escrevemos uma implementação para esses objectos, ou seja escrevemos uma classe; essa classe usa `implements` para declarar que implementa a interface do ponto anterior.
3. Posteriormente, usaremos o nome da interface como tipo (para declarar variáveis, parâmetros e resultados), e o nome da classe como implementação (para criar objectos através do operador `new`).

Vantagem de se usar exclusivamente interfaces como tipos-objecto

A grande vantagem de se usarmos exclusivamente interfaces como tipos-objecto é o facto do nosso código não assumir compromissos escusados.

Ao declararmos as nossas variáveis, parâmetros e resultados usando exclusivamente nomes de interfaces, o nosso código torna-se compatível com todas as classes que implementem essas interfaces (classes existentes ou classes a criar futuramente).

Tipos-objecto genéricos

Em Java é possível definir tipos-objecto genéricos, como no seguinte exemplo esquemático:

```
interface Stack<T> { ... }

class StackClass<T> implements Stack<T> { ... }

class TestStack {
    public static void main(String[] args) {
        Stack<Integer> s = new StackClass<Integer>() ;
        ...
    }
}
```

Existe uma restrição importante em Java:

- O argumento duma interface genérica ou duma classe genérica é obrigatoriamente um tipo-objecto. Por exemplo, o tipo `Stack<int>` é inválido, mas o tipo `Stack<Integer>` já é válido.

Teórica 16

Subtipo

Sejam B e A tipos. Diz-se que B é **subtipo** de A, e escreve-se B«A, se toda a expressão de tipo A puder ser substituída por uma expressão de tipo B, sem que se registe qualquer incompatibilidade (este é o chamado *princípio da substitutividade*).

Por outras palavras, B é **subtipo** de A se toda a expressão de tipo B puder ser usada onde se espera uma expressão de tipo A.

Subtipo é uma relação binária entre tipos que é *reflexiva*, *anti-simétrica* e *transitiva*. Ou seja, se A, B e C forem tipos quaisquer, então:

```
A « A
B « A  &  A « B  ==>  A = B
C « B  &  B « A  ==>  C « A
```

Para que servem os subtipos?

Os subtipos permitem organizar em níveis de abstracção os conceitos usados nos programas, na medida em que permitem tratar objectos dum tipo específico como se fossem objectos dum tipo mais geral.

- Uma linguagem com subtipos é uma linguagem que se adapta bem à maneira de pensar dos programadores, pois os seres humanos pensam e falam das coisas usando abstracções.

Para dar exemplos de conceitos com diferentes níveis de abstracção, considere a seguinte cadeia de conceitos, de abstracção crescente:

```
Gato « Felino « Mamífero « Animal « SerVivo
```

Uma característica da maioria das linguagens orientadas pelos objectos é suportarem uma noção de subtipo.

Abstracção

O que significa *abstrair*? *Abstrair*, significa *ignorar deliberadamente aspectos particulares duma entidade ou dum conceito*, com o objectivo de enfatizar os restantes aspectos, considerados de maior relevância. Por exemplo, o conceito de *Animal* é mais abstracto do que o conceito de *Gato*, porque o primeiro ignora mais informação do que o segundo.

Repare que quanto **mais abstracto** é um conceito, *menos* informação específica tem associada, mas *mais* elementos representa. Pelo contrário, quanto **mais concreto** é um conceito, *mais* informação específica tem associada, mas *menos* elementos representa.

Considerando que `ops(S)` representa o conjunto de operações associadas ao tipo `S`, a ideia anterior formaliza-se assim:

```
B « A  sse
(B contido em A) && (ops(B) contém ops(A))
```

*"**Possibilidades da abstracção**" em "Histórias de Cronópios e de Famas" de Júlio Cortázar*

> *"Há anos que trabalho na Unesco e outros organismos internacionais e ainda conservo algum sentido de humor, especialmente uma notável capacidade de abstracção, que é como quem diz se um tipo me está a chatear risco-o do mapa num abrir e fechar de olhos, e enquanto ele fala que se farta eu penso em Melville e o desgraçado julga que o estou a ouvir. É a mesma coisa se uma rapariga me cai no goto, abstraio da roupa dela assim que entra no meu campo visual, e enquanto fala do frio que está lá fora eu passo um belo bocado a admirar-lhe o umbigo. Chega a ser doentia esta faculdade que tenho.*
>
> *Na passada segunda-feira foram as orelhas. À hora da entrada era extraordinário o número de orelhas que havia na sala de entrada. No meu gabinete encontrei seis orelhas ao meio-dia, no refeitório, eram mais de quinhentas, simetricamente dispostas em duas filas. Era divertido ver de vez em quando duas orelhas que se erguiam, saíam da fila e se afastavam. Pareciam asas. ..."*

Subtipos em Java

Em Java, tal como em C++, os subtipos são explicitamente declarados. Em Java, essa declaração efectua-se usando as palavras reservadas **`implements`** e **`extends`**:

- **implements** serve para dizer que uma classe C implementa uma dada interface I, mas também declara C como subtipo de I;
- **extends** serve para dizer que uma classe C2 extende uma outra classe C1, ou uma interface I2 extende uma outra interface I1, mas também declara uma entidade como subtipo de outra entidade.

Consideremos o seguinte exemplo em Java, que introduz três interfaces e uma classe:

```
interface A1 {
    int p1() ;
}

interface A2 {
    int p2() ;
}

interface B extends A1, A2 {
    int q() ;
}

class C1 implements B {
    public int p1() { return 11 ; }
    public int p2() { return 12 ; }
    public int q() { return 2 ; }
}
```

Quais são as declarações explícitas de subtipo, aqui presentes?

São as seguintes:

```
B « A1
B « A2
C1 « B
```

E quais são as declarações ímplicitas de subtipos?

Por transitividade, são:

```
C1 « A1
C1 « A2
```

Polimorfismo em Java

A linguagem Java suporta uma forma muito especial de polimorfismo, baseada na relação de subtipo, que se chama **polimorfismo de inclusão**.

Uma **variável polimórfica** é uma variável que pode guardar objectos de diversos tipos, mais exactamente dum tipo A e de todos os seus subtipos.

Um **método polimórfico** é um método que aceita argumentos de diversos tipos, mais exactamente dum tipo A e de todos os seus subtipos.

Problema da perda de informação

Os subtipos são uma ajuda preciosa. No entanto, por vezes, os subtipos criam um novo problema que se chama: *o problema da perda de informação*.

Consideremos uma variável de tipo A contendo um objecto de tipo B, com B«A. O problema da perda de informação ocorre se precisarmos de aplicar a essa variável uma operação definida no tipo B, mas não no tipo A. Tal aplicação não é possível; no entanto, o objecto guardado na variável estaria apto a aceitá-la.

O que se passa é que o sistema de tipos, o garante da segurança da linguagem, é obrigado a rejeitar a operação anterior, pois toma as suas decisões com base no tipo declarado para a variável, e nada mais.

A linguagem Java dispõe de mecanismos que permitem contornar este problema:

- O operador de cast **(T)** permite mudar o *tipo* (por vezes chamado *tipo estático*) duma expressão. Este operador tem influência em tempo de compilação (muda o tipo da expressão) e em tempo de execução (o compilador insere um teste para verificar se o valor da expressão tem mesmo o tipo indicado no cast; se não tiver é gerada uma excepção).
- A instrução **instanceof** permite determinar qual a *classe* (por vezes chamada *tipo dinâmico*) dum objecto. Esta instrução tem influência apenas em tempo de execução.

Para exemplificar, considere o seguinte pedaço de código, onde se assume que Gato«Animal:

```
Animal a = new Gato() ;    // Atribuição polimórfica
...
a.Miau() ;                 // ERRO (devido a perda de informação)
...
if( a instanceof Gato )
    ((Gato)a).Miau() ;     // OK (informação recuperada)
else
    // não é gato
```

Note que o envio da mensagem `Miau()` à variável `a` produz um erro de compilação.

Já o envio da mesma mensagem à expressão `((Gato)a)` não dá erro de compilação, embora não evite um possível erro de execução. Neste exemplo concreto, garante-se que não há erro de execução devido ao teste "`a instanceof Gato`".

Subtipos e tipos genéricos

A relação de subtipo também se pode estabelecer entre interfaces e classes genéricas.

Consideremos novamente o exemplo que introduz três interfaces e uma classe, mas agora em versão genérica:

```
interface A1<T> {
    int p1() ;
}

interface A2<T> {
    int p2() ;
}

interface B<T> extends A1<T>, A2<T> {
    int q() ;
}

class C1<T> implements B<T> {
    public int p1() { return 11 ; }
    public int p2() { return 12 ; }
    public int q() { return 2 ; }
}
```

Neste caso, qualquer que seja o tipo-objecto T, verifica-se o seguinte:

```
B<T> « A1<T>
B<T> « A2<T>
C1<T> « B<T>
```

Por exemplo, considerando T=Animal verifica-se:

```
B<Animal> « A1<Animal>
```

Até aqui tudo está está conforme à nossa intuição.

Não-variância dos argumentos genéricos

Uma questão que interessa agora colocar é a seguinte. Será que se verifica

```
B<Gato> « B<Animal>      // inválido
```

Na verdade não. Esta regra só seria válida se os valores dos tipos envolvidos não pudessem mudar; mas eles podem mudar, em geral.

Para nos convencermos da não validade da regra, consideremos a classe `Vector`, uma classe genérica de biblioteca cujos elementos podem mudar. Raciocinemos sobre a proposição

```
Vector<Gato> « Vector<Animal>      // inválido
```

A pergunta é: Será que um vector de gatos pode ser considerado, para todos os efeitos, um vector de animais? Não, pela seguinte razão:

- Num vector de animais podemos inserir qualquer animal. Assim, se um vector de gatos fosse um vector

de animais então poderiamos inserir nele qualquer animal, digamos, um crocodilo, Mas isso não pode ser.

```
    Vector<Gato> gs = new Vector<Gato>(10) ;
    Vector<Animal> as = gs ;      // inválido (gera erro de compilação)
    as.add(0, new Crocodilo()) ;
    gs.get(0).Miau() ;            // se a atribuição inválida tivesse sido aceite, daria
ERRO DE EXECUÇÃO
```

Teórica 17

Listas funcionais

Listas funcionais são listas não-mutáveis onde as alterações se exprimem através da produção de cópias modificadas (como nas linguagens funcionais). Por outras palavras:

- As listas funcionais só devem suportar operações não-destrutivas.

Tipos soma em Java

Em Java, um tipo-soma é introduzido através duma interface (é assim que qualquer tipo-objecto deve ser introduzido), mas implementa-se usando múltiplas classes: uma classe por cada forma que os valores do tipo-soma possam assumir.

Implementação de listas funcionais em Java

Como sabemos do ML, o tipo "lista-funcional" é um tipo-soma cujos valores assumem duas formas distintas:

- forma "lista vazia";
- forma "lista não-vazia"

Vamos programar o tipo "lista-funcional" usando uma interface chamada "FList", e duas classes chamadas "Nil" e "Cons":

- a classe "Nil" implementará as listas vazias;
- a classe "Cons" implementará as listas não-vazias.

Usaremos o método indutivo na escrita da maioria dos métodos. Como se sabe, quando se usa o método indutivo em ML, as operações sobre listas têm de ser programadas fazendo análise de casos: caso "lista vazia" e caso "lista não-vazia". Quando se usa o método indutivo em Java, as operações sobre listas programam-se através de dois métodos: um trata o caso da "lista vazia" e define-se na classe "Nil"; o outro trata o caso da "lista não-vazia" e define-se na classe "Cons". Leia com atenção o código das duas classes para perceber a ideia.

De qualquer forma, repare que não é obrigatório user sempre método indutivo aqui (programamos a operação "Print" de forma imperativa). O que é essencial é que as operações sejam não-destrutivas, como planeado.

```
interface FList<T> {
    T Head() ;
    FList<T> Tail() ;
    boolean IsEmpty() ;
    void Print() ;
    FList<T> Append(FList<T> l) ;
    FList<T> Rev() ;
}

class Cons<T> implements FList<T> {
    private T head ;
    private FList<T> tail ;

    Cons(T h, FList<T> t) { head = h ; tail = t ; }
    Cons(T a) { this(a, new Nil<T>()) ; }
    Cons(T a, T b) { this(a, new Cons<T>(b)) ; }
    Cons(T a, T b, T c) { this(a, new Cons<T>(b,c)) ; }
    Cons(T a, T b, T c, T d) { this(a, new Cons<T>(b,c,d)) ; }

    public T Head() { return head ; }
    public FList<T> Tail() { return tail ; }
    public boolean IsEmpty() { return false ; }

    public void Print() {   /* código imperativo */
        System.out.print("[" + head) ;
        for( FList<T> l = tail ; !l.IsEmpty() ; l = l.Tail() )
            System.out.print("; " + l.Head()) ;
        System.out.println("]") ;
    }
    public FList<T> Append(FList<T> l) {   /* código indutivo */
        FList<T> base = tail.Append(l) ;
        return new Cons<T>(head, base) ;
    }
    public FList<T> Rev() {   /* código indutivo */
```

```
            FList<T> base = tail.Rev() ;
            return base.Append(new Cons<T>(head)) ;
        }
    }

    class Nil<T> implements FList<T> {
        Nil() {}

        public T Head() { throw new Error("Head: Empty list!") ; }
        public FList<T> Tail() { throw new Error("Tail: Empty list!") ; }
        public boolean IsEmpty() { return true ; }

        public void Print() { System.out.println("[]") ; }
        public FList<T> Append(FList<T> l) { return l ; }
        public FList<T> Rev() { return this ; }
    }

    class TestFLists {
        public static void main(String[] args) {
            FList<Integer> l = new Cons<Integer>(1,2,3,4) ;
            FList<Integer> r = l.Rev() ;
            l.Print() ;
            r.Print() ;
        }
    }
```

Listas imperativas

Listas imperativas são listas mutáveis que, portanto, podem ser alvo de modificação directa. Por exemplo, uma lista mutável pode passar de vazia a não-vazia e vice-versa.

Implementação de listas imperativas em Java

Tradicionalmente as listas imperativas implementam-se usando uma lista ligada de nós, sendo o final dessa lista de nós marcado pela constante `null`. Nesta secção, é nosso objectivo mostrar como se escreve essa implementação tradicional.

Introduzimos o tipo "lista-imperativa" através duma interface, chamada "IList, e implementamo-lo numa única classe, chamada "IListClass". Precisamos ainda duma classe interna auxiliar, chamada "Node", para representar internamente os nós da lista. Na classe "IListClass", a variável de instância privada "items" indica o início da lista de nós.

Analizando a interface "IList", é fácil concluir que esta caracteriza realmente listas imperativas. Exceptuando a operação "Clone", nenhuma das outras operações retorna uma lista, o que significa que será sempre a lista "this" o alvo das alterações.

Nesta implementação os elementos da classe "Node" deixam-se passivamente manipular pelos métodos, comportando-se mais como valores do que como objectos.

```
    interface IList<T> {
        int Size() ;
        T Get(int i) ;
        void PutAtHead(T i) ;
        void PutAtEnd(T i) ;
        IList<T> Clone() ;
        void Append(IList<T> dl) ;
        void Rev() ;
        void Print() ;
    }

    class IListClass<T> implements IList<T> {
        private static class Node<E> {  // private inner class
            Node(E v, Node<E> n) { value = v ; next = n ; }
            public E value ;
            public Node<E> next ;
        }
        private Node<T> items ;  // null means empty list

        private IListClass(Node<T> it) { items = it ; }
        IListClass() { items = null ; }
        IListClass(T a) { items = new Node<T>(a, null) ; }
        IListClass(T a, T b) { items = new Node<T>(a, new Node<T>(b, null)) ; }
        IListClass(T a, T b, T c)
```

```
        {items = new Node<T>(a, new Node<T>(b, new Node<T>(c, null))) ; }
    IListClass(T a, T b, T c, T d)
        {items = new Node<T>(a, new Node<T>(b, new Node<T>(c, new Node<T>(d, null)))) ;
}

    public int Size() {
        int r = 0 ;
        for( Node<T> l = items ;  l != null ; l = l.next, r++ ) ;
        return r ;
    }
    public T Get(int i) {
        Node<T> l = items ;
        for( int j = 0 ; j < i && l != null ; j++, l = l.next) ;
        if( l == null )
            throw new Error("Get: invalid index") ;
        return l.value ;
    }
    public void PutAtHead(T i) {
        items = new Node<T>(i, items) ;
    }
    public void PutAtEnd(T i) {
        if( items == null )
            items = new Node<T>(i, null) ;
        else {
            Node<T> l ;
            for( l = items ; l.next != null ; l = l.next ) ;
            l.next = new Node<T>(i, null) ;
        }
    }
    public IList<T> Clone() {
        if( items == null ) return new IListClass<T>() ;
        Node<T> r = new Node<T>(items.value, null) ;
        Node<T> p = r ;
        for( Node<T> l = items.next ; l != null ; l = l.next ) {
            p.next = new Node<T>(l.value, null) ;
            p = p.next ;
        }
        return new IListClass<T>(r) ;
    }
    public void Append(IList<T> dl) {
        int size = dl.Size() ;
        for( int i = 0 ; i < size ; i++ )
            PutAtEnd(dl.Get(i)) ;
    }
    public void Rev() {
        Node<T> r = null ;
        Node<T> l = items ;
        while( l != null ) {
            Node<T> temp = l.next ;
            l.next = r ;
            r = l ;
            l = temp ;
        }
        items = r ;
    }
    public void Print() {
        if( items == null )
            System.out.print("[]") ;
        else {
            Node<T> l = items ;
            System.out.print("[" + l.value) ;
            for( l = l.next ; l != null ; l = l.next )
                System.out.print("; " + l.value) ;
            System.out.println("]") ;
        }
    }
}

class TestIList {
    public static void main(String[] args) {
        IList<Integer> l = new IListClass<Integer>(1,2,3,4) ;
        IList<Integer> m = new IListClass<Integer>(5,6,7,8) ;
        l.Append(m) ;
```

```
            l.Print() ;
            l.Rev() ;
            l.Print() ;
            l.Clone().Print() ;
            System.out.println("size = " + l.Size())  ;
            System.out.println("l0 = " + l.Get(0))  ;
            System.out.println("l1 = " + l.Get(1))  ;
            System.out.println("l2 = " + l.Get(2))  ;
            System.out.println("l3 = " + l.Get(3))  ;
            System.out.println("l4 = " + l.Get(4))  ;
            System.out.println("l5 = " + l.Get(5))  ;
        }
    }
```

Comparação entre os dois tipos de listas

Listas funcionais

- As listas funcionais são as mais seguras, pois não suportam operações de modificação directa (as quais podem ser uma perigosa fonte de confusão).
- As listas funcionais são especialmente úteis no caso de programas que manipulam numerosas listas e as combinam de formas sofisticadas.
- Em muitas situações as listas funcionais permitem um bom aproveitamento da memória: por exemplo, o método "Append" reutiliza directamente a sua lista-argumento como parte da lista-resultado. Em geral, duas listas funcionais podem partilhar as suas porções finais.
- Mas existem operações com as quais as listas funcionais se dão mal. É o caso da operação de remoção dum elemento duma lista. Não podemos eliminar simplesmente o elemento; temos de criar uma cópia modificada da lista. Caso a operação de remoção seja frequentemente aplicada a listas muito grandes, a eficiência do programa ressentir-se-á.

Listas imperativas

- Num programa que trabalhe com poucas listas e sem grande sofisticação, não há problema em usar listas imperativas.
- Em geral permitem um bom aproveitamento da memória: por exemplo, a remoção dum elemento é efectuada por alteração directa.

Que variedades de listas são suportadas pela plataforma Java?

A plataforma Java suporta apenas **listas imperativas** (através duma interface chamada "List"). Esta escolha parece correcta: na prática, só uma minoria dos programas necessita realmente de usar listas funcionais (e nesse caso temos de ser nós a programá-las).

Existem tipos funcionais predefinidos em Java?

Alguns dos tipos predefinidos em Java são funcionais, no sentido de só suportarem operações não destrutivas. É o que se passa com todos os tipos primitivos e ainda com a classe "String". [Sugestão: vá à página da cadeira ver a documentação sobre a classe "String", e verifique que, o seu protocolo é realmente *funcional*.]

De qualquer forma a esmagadora maioria dos tipos predefinidos na plataforma Java tem natureza imperativa, ou seja, suportam uma ou mais operações destructivas.

Gestão automática de memória em Java

Em Java, o programador não tem de se preocupar com a libertação da memória ocupada pelos objectos de que já não necessita. O sistema de suporte inclui uma rotina (chamada *garbage collector*) que periodicamente analisa os objectos existentes, determina quais são os que se encontram inacessíveis, e elimina-os.

Já em C++, é ao programador que cabe a responsabilidade de eliminar esses objectos (usando primitiva `delete`). Ora, a gestão de memória nos programas tende a ser algo complicada. Não é pois de estranhar que a utilização da primitiva `delete` seja uma das principais fontes de erros de execução nos programas escritos em C++.

Do ponto de vista da gestão de memória, o problema das listas funcionais, atrás analisado, é particularmente interessante. O facto duma lista poder ter partes suas partilhadas com outras listas, inviabiliza as estratégias de libertação de memória tradicionais. Assim, se desejássemos trabalhar com listas funcionais em C++, teríamos apenas duas hipóteses:

- Ou escrevíamos, nós mesmos, em C++, um sistema recolha de lixo para as nossas listas (usaríamos um dos vários algoritmos descritos na literatura);

- Ou abdicávamos de eliminar os objectos *mortos*, havendo, neste caso, o perigo do programa terminar abruptamente por falta de memória livre para criar novos objectos.

Teórica 18

Herança entre classes

Uma característica essencial das classes é o facto de elas serem incrementalmente modificáveis. Usando o *mecanismo de herança*, o programador consegue criar de forma expedita uma nova classe (**subclasse**) a partir de outra (**superclasse**), definindo apenas as componentes da nova classe que são *adicionadas* ou *modificadas*, relativamente à classe original. As componentes da superclasse que não forem redefinidas na subclasse são automaticamente **herdadas**, ou seja, são reaproveitadas na subclasse.

Nas subclasses, não é permitida a eliminação de componentes herdadas. Quando se sente a necessidade de proceder dessa forma, isso significa que as classes estão deficientemente organizadas: a subclasse deveria antes ser *superclasse*!

Em Java cada classe só pode ter uma superclasse imediata, pelo que a herança se diz **simples**. Já no C++, cada classe só pode diversas superclasses imediatas, dizendo-se por isso que a herança é **múltipla**.

`extends` é a palavra reservada que se usa em Java para definir subclasses. Exemplo:

```
interface I {
    void p1() ;
}
class A implements I {   // superclasse
    public void p1() { }
}
class B extends A {    // subclasse
    public void p2() { }
}
```

Em Java uma subclasse define um subtipo. Por isso convém que, do ponto de vista lógico, uma subclasse represente um refinamento, ou especialização, da sua superclasse. Por exemplo, faz sentido definir uma classe `MammalClass` como subclasse duma classe `AnimalClass`.

Classe que usa outra classe não implica herança

O facto duma classe B utilizar uma classe A, **não** significa que B deva ser subclasse de A. (Este é um erro relativamente comum, por vezes é cometido por alguns alunos nos exames).

Por exemplo, o facto dum automóvel ter um volante não significa que a classe que define automóveis `CarClass` deva herdar da classe que define volantes `DrivingWheelClass`. Isso seria um erro grave pois assim teriamos `CarClass « DrivingWheelClass`. Desde quando é que um carro é um volante?!

Um carro não é um volante. Um carro **tem** sim um volante! Na prática isto significa que se define uma variável de tipo `DrivingWheel` dentro da classe `CarClass`.

Herança entre interfaces

Além da herança de código, a linguagem Java também prevê a herança de assinaturas entre interfaces. Portanto, considera-se em Java a existência **subinterfaces** e **superinterfaces**.

A herança de interfaces é **múltipla**, pois uma interface pode herdar de diversas interfaces. A palavra reservada que se usa em Java para declarar uma subinterface é a palavra **`extends`**. Exemplo:

```
interface I1 {
    int p1() ;
}

interface I2 {
    int p2() ;
}

interface J extends I1, I2 {
    int q() ;
}
```

Toda a classe que implemente uma dada interface também é obrigada a implementar as suas superinterface.

Exemplo de reutilização de código

O mecanismo de herança ajuda a escrever código **reutilizável**.

No seguinte exemplo definimos uma interface, Bag, e uma sua implementação, BagClass. Depois reutilizamos ambas na definição duma nova interface, Set, e duma nova implementação, SetClass.

Aqui trabalhamos com colecções de inteiros, mas não haveria problema em definir colecções genéricas, parametrizadas num tipo T.

Interface Bag e classe BagClass

```
interface Bag {
    int Size() ;
    int Get(int i) ;
    void Store(int v) ;
    int Many(int v) ;
    Bag UnionF(Bag s) ;   // união funcional
    void UnionI(Bag s) ;  // união imperativa
}

class BagClass implements Bag {
    private int[] elems ;
    private int nElems ;

    BagClass() {
        elems = new int[100] ;
        nElems = 0 ;
    }
    public int Size() {
        return nElems ;
    }
    public int Get(int i) {
        if( 0 <= i  && i < nElems )
            return elems[i] ;
        else throw new Error("Get: Outside valid range") ;
    }
    public void Store(int v) {
        if( nElems < elems.length )
            elems[nElems++] = v ;
        else throw new Error("Store: Too many elements") ;
    }
    public int Many(int v) {
        int n = 0 ;
        for( int i = 0 ; i < nElems ; i++ )
            if( elems[i] == v ) n++ ;
        return n ;
    }
    public Bag UnionF(Bag s) {
        Bag n = new BagClass() ;
        for( int i = 0 ; i < Size() ; i++ )
            n.Store(Get(i)) ;
        for( int i = 0 ; i < s.Size() ; i++ )
            n.Store(s.Get(i)) ;
        return n ;
    }
    public void UnionI(Bag s) {
        for( int i = 0 ; i < s.Size() ; i++ )
            Store(s.Get(i)) ;
    }
}
```

Subinterface Set e subclasse SetClass

```
interface Set extends Bag {
    boolean Belongs(int v) ;
    Set UnionF(Set s) ;
}

class SetClass extends BagClass implements Set {
   SetClass() {
        super() ;    // Invoca o constructor da superclasse
    }
    public boolean Belongs(int v) {
        return Many(v) > 0 ;
    }
```

```
    public void Store(int v) {
        if( !Belongs(v) )
            super.Store(v) ; // Tira partido do método da superclasse
    }
    public Set UnionF(Set s) {
        Set n = new SetClass() ;
        for( int i = 0 ; i < Size() ; i++ )
            n.Store(Get(i)) ;
        for( int i = 0 ; i < s.Size() ; i++ )
            n.Store(s.Get(i)) ;
        return n ;
    }
}

class TestBags {
    public static void main(String [] args) {
        Bag b1 = new BagClass() ;
        Bag b2 = new SetClass() ;
        Set s = new SetClass() ;
        // ...
    }
}
```

Teórica 19

Factorização

O mecanismo de herança ajuda a escrever código **factorizado**, ou seja código **sem redundância**.

Se diversas classes apresentarem bastante código duplicado, provavelmente é porque se tratam de implementações de casos particulares do mesmo conceito. Esse conceito deve ser identificado, e depois materializado numa classe que concentre, num único ponto, portanto sem redundância, todo o código comum.

As classes originais passam a incorporar o código factorizado através do mecanismo de herança.

Classes abstractas e métodos abstractos

Em resultado duma factorização, muitas vezes surgem classes que correspondem a conceitos tão gerais (e.g. Animal, Veículo ou Entidade) que não faz sentido criar instâncias directas dessas classes, nem concretizar código concreto para alguns dos métodos. Essas classes são chamadas classes abstractas.

A linguagem Java suporta **classes abstractas**, classes *incompletas* de que não geram instâncias e onde podem ocorrer métodos sem corpo, ditos **métodos abstractos**.

Todas as classes e métodos abstractos requerem uma declaração explícita, usando a palavra reservada **`abstract`**.

Chamamos **classes concreta** a qualquer classe não abstracta. Uma classe concreta tem a responsabilidade de definir todos os métodos abstractos que herda; uma classe abstracta já não tem essa responsabilidade.

Exemplo de factorização

Apresentamos agora um exemplo, em três passos, que pretende ilustrar as vantagem da factorização e das classes abstractas.

Pretendemos representar **expressões algébricas** nas quais possam ocorrer as seguintes entidades: o operador binário "+", o operador binário "*", o operador unário "-", uma variável "x" e ainda literais reais. Exemplos:

```
2*x2+5
2*x7+5*(x-5)
```

Uma representação em árvore é a ideal para capturar a estrutura destas expressões e facilitar a sua manipulação.

Como está em causa um tipo soma (pois necessitamos de diversas variedades de nós), o tipo das expressões introduz-se através duma interface e implementa-se usando múltiplas classes: uma classe por variante.

Expressões algébricas sem factorização

Começamos por implementar o tipo soma *Expressão algébrica* sem usar factorização. Repare na grande quantidade de código repetido que aparece nas várias classes.

```
interface Exp {
    boolean Big() ;
    int Size() ;
    int Height() ;
    double Eval(double v) ;
    Exp Deriv() ;
}

class AddClass implements Exp {
    private Exp l, r ;
    AddClass(Exp l, Exp r) { this.l = l ; this.r = r ; }
    public boolean Big() { return Size() > 1000 ; }
    public int Size() { return 1 + l.Size() + r.Size() ; }
    public int Height() { return 1 + Math.max(l.Height(),r.Height()) ; }
    public double Eval(double v) { return l.Eval(v) + r.Eval(v) ; }
    public Exp Deriv() { return new AddClass(l.Deriv(), r.Deriv()) ; }
}

class MultClass implements Exp {
    private Exp l, r ;
    MultClass(Exp l, Exp r) { this.l = l ; this.r = r ; }
    public boolean Big() { return Size() > 1000 ; }
```

```
    public int Size() { return 1 + l.Size() + r.Size() ; }
    public int Height() { return 1 + Math.max(l.Height(),r.Height()) ; }
    public double Eval(double v) { return l.Eval(v) * r.Eval(v) ; }
    public Exp Deriv() {
        return new AddClass(new MultClass(l, r.Deriv()),
                            new MultClass(l.Deriv(), r)) ;
    }
}

class SimClass implements Exp {
    private Exp e ;
    SimClass(Exp e) { this.e = e ; }
    public boolean Big() { return Size() > 1000 ; }
    public int Size() { return 1 + e.Size() ; }
    public int Height() { return 1 + e.Height() ; }
    public double Eval(double v) { return -e.Eval(v) ; }
    public Exp Deriv() { return new SimClass(e.Deriv()) ; }
}

class ConstClass implements Exp {
    private double c ;
    ConstClass(double c) { this.c = c ; }
    public boolean Big() { return Size() > 1000 ; }
    public int Size() { return 1 ; }
    public int Height() { return 1 ; }
    public double Eval(double v) { return c ; }
    public Exp Deriv() { return new ConstClass(0) ; }
}

class VarClass implements Exp {
    VarClass() { }
    public boolean Big() { return Size() > 1000 ; }
    public int Size() { return 1 ; }
    public int Height() { return 1 ; }
    public double Eval(double v) { return v ; }
    public Exp Deriv() { return new ConstClass(1) ; }
}

class TestExps {
    public static void main(String [] args) {
        Exp e = new MultClass(
                    new AddClass(new ConstClass(4), new VarClass()),
                    new ConstClass(6)) ;
        System.out.println(e.Deriv().Eval(3)) ;
    }
}
```

Expressões algébricas com um nível de factorização

Aqui factorizamos o código do exemplo anterior. Para isso, introduzimos as classes abstractas BinClass, UnClass e ZeroClass que implementam, respectivamente, os conceitos abstractos de *nó binário*, *nó unário* e *nó zeroário*.

Apesar das classes BinClass, UnClass, e ZeroClass não conterem qualquer método abstracto, elas foram declaradas como abstractas. Por duas razões:

- para impedir a criação de intâncias dessas classes;
- para alertar o leitor do código que estão realmente em causa três conceitos abstractos.

```
interface Exp {
    boolean Big() ;
    int Size() ;
    int Height() ;
    double Eval(double v) ;
    Exp Deriv() ;
}

abstract class BinClass {
    protected Exp l, r ;
    BinClass(Exp l, Exp r) { this.l = l ; this.r = r ; }
    public boolean Big() { return Size() > 1000 ; }
    public int Size() { return 1 + l.Size() + r.Size() ; }
    public int Height() { return 1 + Math.max(l.Height(),r.Height()) ; }
```

```
}

abstract class UnClass {
    protected Exp e ;
    UnClass(Exp e) { this.e = e ; }
    public boolean Big() { return Size() > 1000 ; }
    public int Size() { return 1 + e.Size() ; }
    public int Height() { return 1 + e.Height() ; }
}

abstract class ZeroClass {
    ZeroClass() { }
    public boolean Big() { return Size() > 1000 ; }
    public int Size() { return 1 ; }
    public int Height() { return 1 ; }
}

class AddClass extends BinClass implements Exp {
    AddClass(Exp l, Exp r) { super(l,r) ; }
    public double Eval(double v) { return l.Eval(v) + r.Eval(v) ; }
    public Exp Deriv() { return new AddClass(l.Deriv(), r.Deriv()) ; }
}

class MultClass extends BinClass implements Exp {
    MultClass(Exp l, Exp r) { super(l,r) ; }
    public double Eval(double v) { return l.Eval(v) * r.Eval(v) ; }
    public Exp Deriv() {
        return new AddClass(new MultClass(l, r.Deriv()),
                            new MultClass(l.Deriv(), r)) ;
    }
}

class SimClass extends UnClass implements Exp {
    SimClass(Exp e) { super(e) ; }
    public double Eval(double v) { return -e.Eval(v) ; }
    public Exp Deriv() { return new SimClass(e.Deriv()) ; }
}

class ConstClass extends ZeroClass implements Exp {
    private double c ;
    ConstClass(double c) { super() ; this.c = c ; }
    public double Eval(double v) { return c ; }
    public Exp Deriv() { return new ConstClass(0) ; }
}

class VarClass extends ZeroClass implements Exp {
    VarClass() { super() ; }
    public double Eval(double v) { return v ; }
    public Exp Deriv() { return new ConstClass(1) ; }
}
```

Expressões algébricas com dois níveis de factorização

A factorização anterior deixa ainda um pouco a desejar porque o método `Big` aparece repetido em três classes. Por isso vamos factorizar ainda um pouco mais. Introduzimos uma nova classe, `ExpClass`, para implementar o conceito geral de *nó*.

Repare que o método `Big` invoca o método `Size` na classe ExpClass, logo `Size` terá de ser conhecido nessa mesma classe. Mas ao nível desta classe não é possível concretizar o seu código. Por isso, o método `Size` teve de ser declarado como abstracto.

Nota: Na realidade, declarar o método `Size` como abstracto é facultativo, neste caso: a classe `ExpClass` já o declara indirectamente como abstracto ao dizer que implementa a interface `Exp`. No entanto é boa idaia explicitar a declaração para tornar o código mais fácil de perceber.

```
interface Exp {
    boolean Big() ;
    int Size() ;
    int Height() ;
    double Eval(double v) ;
    Exp Deriv() ;
}
```

```
abstract class ExpClass implements Exp {
    ExpClass() { }
    public boolean Big() { return Size() > 1000 ; }
    public abstract int Size() ;
}

abstract class BinClass extends ExpClass {
    protected Exp l, r ;
    BinClass(Exp l, Exp r) { super() ; this.l = l ; this.r = r ; }
    public int Size() { return 1 + l.Size() + r.Size() ; }
    public int Height() { return 1 + Math.max(l.Height(),r.Height()) ; }
}

abstract class UnClass extends ExpClass {
    protected Exp e ;
    UnClass(Exp e) { super() ; this.e = e ; }
    public int Size() { return 1 + e.Size() ; }
    public int Height() { return 1 + e.Height() ; }
}

abstract class ZeroClass extends ExpClass {
    ZeroClass() { super() ; }
    public int Size() { return 1 ; }
    public int Height() { return 1 ; }
}

class AddClass extends BinClass {
    AddClass(Exp l, Exp r) { super(l,r) ; }
    public double Eval(double v) { return l.Eval(v) + r.Eval(v) ; }
    public Exp Deriv() { return new AddClass(l.Deriv(), r.Deriv()) ; }
}

class MultClass extends BinClass {
    MultClass(Exp l, Exp r) { super(l,r) ; }
    public double Eval(double v) { return l.Eval(v) * r.Eval(v) ; }
    public Exp Deriv() {
        return new AddClass(new MultClass(l, r.Deriv()),
                            new MultClass(l.Deriv(), r)) ;
    }
}

class SimClass extends UnClass {
    SimClass(Exp e) { super(e) ; }
    public double Eval(double v) { return -e.Eval(v) ; }
    public Exp Deriv() { return new SimClass(e.Deriv()) ; }
}

class ConstClass extends ZeroClass {
    private double c ;
    ConstClass(double c) { super() ; this.c = c ; }
    public double Eval(double v) { return c ; }
    public Exp Deriv() { return new ConstClass(0) ; }
}

class VarClass extends ZeroClass {
    VarClass() { super() ; }
    public double Eval(double v) { return v ; }
    public Exp Deriv() { return new ConstClass(1) ; }
}
```

Teórica 20

Excepções em Java

A introdução da existência dum mecanismo de excepções em Java é idêntica à motivação da existência desse mesmo mecanismo em ML, C++ e noutras linguagens. Trata-se da possibilidade de escrever programas robustos que:

- Por um lado, sabem lidar com situações *excepcionais*, em particular com erros gerados em tempo de execução pelo hardware ou pelo sistema operativo;
- Por outro lado, a parte que lida com as situações *normais* não fica obscurecida pelo tratamento dessas circunstâncias excepcionais.

Em Java uma excepção é um **objecto** que, entre outra informação, regista os métodos que estavam activos quando essa excepção foi criada.

Todas as excepções são instâncias da classe predefinida `Throwable` ou de suas subclasses. Cada tipo de excepção é representado por meio duma subclasse específica de `Throwable`. Existem muitas destas classes predefinidas na plataforma Java. O utilizador pode definir ainda mais.

Excepções verificadas e não-verificadas

A classe `Throwable` tem duas subclasses imediatas chamadas `Error` e `Exception`. Por sua vez a classe `Exception` tem, entre outras, uma subclasse imediata chamada `RuntimeException`. Estas três classes determinam uma importante divisão no conjunto das excepções:

- **Excepções não-verificadas:** São as excepções definidas pelas subclasses de `Error` e de `RuntimeException`. Representam erros que se espera que **não** sejam tratados. Exemplos: `OutOfMemoryError`, `StackOverflowError`.
- **Excepções verificadas:** As restantes subclasses de `Exception` definem excepções que são obrigatoriamente tratadas. Exemplo: `IOException`.

Instruções e declarações relacionadas com as excepções

Em Java, a captura de excepções efectua-se usando a construção `try-catch-finally`. Esta construção efectua uma análise de casos. Exemplo:

```
try {
    int i = arr[5] ;
}
catch ( ArrayIndexOutOfBoundsException e ) {
    System.out.println("Out of bounds.");
}
catch ( NullPointerException e ) {
    System.out.print("arr ainda não foi inicializado");
}
finally {
    System.out.println("Terminou!");
}
```

A parte `finally` é opcional: quando está presente garante-se a sua execução, quer tenha sido gerada excepção ou não. As excepções que não forem tratadas dentro duma função provocam o retorno imediato dessa função.

Por vezes é necessário activar deliberadamente uma excepção: isso faz-se usando a comando `throw`. Exemplo:

```
throw new FontFormatException("Bad font!") ;
```

Um método que active deliberadamente uma excepção, pode anunciar esse facto usando a declaração `throws`. No caso de se tratar duma excepção verificada, essa declaração é mesmo obrigatória. Exemplo: Nas classe `Reader` e `Writer`, os métodos são obrigados a declarar que activam a excepção `IOException`. Eis os cabeçalhos de dois deles:

```
public abstract int read() throws IOException ;
public abstract int read(char[] cbuf) throws IOException ;
```

A classe Object

Em Java, a relação de herança determina uma organização hierárquica das classe. Posicionada no topo desta hierarquia encontra-se uma classe especial chamada **Object** e pertencente ao package java.lang. A classe

Object é especial por duas razões:

- É a única classe que não tem superclasse.
- É superclasse de todas as classes, existentes ou a existir: portanto todas as outras classes herdam dela.

Quando não se declara explicitamente a superclasse imediata duma classe, implicitamente essa classe será subclasse imediata de Object.

Métodos

Alguns dos métodos mais importantes da classe Object, herdados por todas as outras classes, são os seguintes:

```
public boolean equals(Object) ;
public String toString() ;
public Class getClass() ;
```

Eis uma descrição sumária deles:

equals(Object)

Compara dois objectos. Geralmente interessa redefinir este método nas subclasses, porque a definição original limita-se a comparar referências: ou seja, a definição original implementa a operação de *teste de identidade*.

Exemplo de redefinição:

```
class StackClass implements Stack {
    ....
    public equals(Object obj) {
        if( obj == null || !(obj instanceof Stack) )
            return false ;
        else {
            Stack sobj = (Stack)obj ;
            // aqui, comparam-se os stacks this e obj
        }
    }
}
```

toString()

Produz uma representação textual dum objecto. Geralmente interessa redefinir este método nas subclasses, pois a definição original não é muito útil. Por exemplo, assumindo que este método não é redefinido na classe StackClass, a seguinte expressão

```
(new StackClass()).toString()
```

produz uma string semelhante a

```
"StackClass@3A56EB44"
```

Importa saber que na stream `System.out`, os métodos `print(Object)` e `println(Object)` estão definidos à custa de `toString()`. Assim, por exemplo, o seguinte comando

```
System.out.println(new StackClass())
```

produz um output semelhante a

```
StackClass@3A56EB44
```

O operador primitivo "+" de soma de Strings também tira partido do método `toString()`, o que permite, por exemplo, escrever `("" + o)` para obter a representação textual do objecto `o`.

getClass()

Nunca interessa redefinir este método nas subclasses. Retorna a classe dum objecto.

Teórica 21

Extensibilidade

Um **sistema extensível** é um sistema que se pode fazer crescer, sem que se tenha de se alterar o que já foi escrito.

A extensibilidade obtém-se pela via da **abstracção**:

- Código que seja escrito com base em conceitos abstractos consegue lidar com as entidades iniciais descritas no enunciado do problema, mas também (e aqui é que surge a extensibilidade) com entidades a criar no futuro. Claro que isto só funciona se as entidades futuras se enquadrarem nas abstracções inicialmente consideradas.

Há todo o interesse em software extensível. As principais razões são as seguintes:

- Os programas ficam mais fáceis de actualizar.
- Os programas ficam mais fiáveis.
- O tempo de vida útil dos programas aumenta.
- Poupa-se tempo e dinheiro.
- Os programas ficam mais elegantes e tornam-se fonte de prazer estético (pelo menos para quem os escreve).

Extensibilidade em Java

Os mecanismos da linguagem Java que ajudam a escrever programas extensíveis são os seguintes:

- **Herança**: Este é o mecanismo de extensão mais óbvio, na medida em que torna as classes entidades extensíveis. O código da superclasse é reutilizado (e reinterpretado) no contexto da subclasse, sem que tenha de ser reescrito.
- **Subtipo**: Este mecanismo, por vezes também designado por *polimorfismo (de inclusão)*, permite a escrita de código abstracto, que funciona tão bem com as classes existentes como com novas classes a criar no futuro.
- **Envio de mensagens**: Este mecanismo incorpora uma análise de casos implícita, que funciona tão bem com as classes existentes como com novas classes a criar no futuro.

Estes três mecanismos ajudam a escrever código extensível, mas não dão quaisquer garantias, só por si. O programador que deseje escrever código extensível tem de organizar cuidadosamente o seu código para obter extensibilidade. Em particular tem de evitar a armadilha dos testes explícitos de classe.

Testes explícitos de classe

Existe uma questão que, de forma dramática, dá origem a **código não-extensível**. Trata-se do uso de **testes explícitos para determinar a classe concreta dum objecto** (feita usando `instanceof` ou de outras formas).

Tal código não pode ser extensível, pois está comprometido com as classes existentes: ou seja, não conseguirá lidar com objectos de classes a criar no futuro. Para lidar com esses novos objectos, seria necessário reescrever o código...

Como evitar os testes explícitos de classe

Por vezes surge a tentação de testar directamente a classe concreta a que pertence um objecto. Isso tem de ser evitado a todo o custo, se tivermos como objectivo a escrita de código extensível.

Vejamos algumas técnicas que permitem evitar os testes explícitos de classe.

Técnica do envio de mensagem

Em vez de testar directamente a classe concreta dum objecto, podemos enviar-lhe uma mensagem perguntando algo. Tal código já é extensível pois funciona com quaisquer objectos que suportem um dado método, mesmo com objectos de classes a criar futuramente.

Por exemplo, num jogo baseado numa matriz bidimensional, em que vários monstros perseguem um herói humano, o que é que um monstro deverá fazer quando se cruza com outra personagem?

- **Versão errada** (visão dum mundo fechado) -- O monstro determina, usando a expressão `vizinho instanceof ManClass`, se o vizinho é o herói. Se for o herói, então o monstro almoça o herói. Se não for o herói, então não faz nada. Este código não é nada extensível pois a sua lógica está dependente

das classes iniciais.

- **Versão correcta** (visão dum mundo aberto) -- O monstro envia ao vizinho a mensagem `vizinho.Comestível()` e depois actua em conformidade. Este código já é extensível, pois funciona com qualquer personagem, mesmo com uma personagem a criar futuramente, desde que esta saiba responder à mensagem `Comestível()`.

Em conclusão, conseguimos escrever código extensível introduzindo o conceito abstracto *comestível*.

Técnica das interfaces auxiliares

O conceito *comestível*, referido no ponto anterior, é realmente abstracto. Suponha que introduzimos esse conceito usando uma interface `Comestível`. Suponha ainda que todas as classes que definem personagens comestíveis seguem a regra de implementar a interface `Comestível`.

Nestas condições é fácil testar de uma dada personagem vizinha é, ou não, comestível. Basta escrever:

```
vizinho instanceof Comestível
```

Este código é extensível porque é abstracto.

A interface `Comestível` pode ficar vazia; a sua simples existência é quanto basta.

Técnica dos níveis

Suponha que no jogo existem muitas classes diferentes de personagens, e que entre essas classes de personagens se estabelecem complexas regras de alimentação, do tipo *cadeia alimentar*.

Nesse caso, convém associar um *nível alimentar* a cada tipo de personagem e estabelecer a seguinte regra: uma personagem pode comer outra personagem caso o nível alimentar da primeira seja superior ao da segunda. Concretamente, um objecto pode comer o seu vizinho se:

```
NivelAlimentar() > vizinho.NivelAlimentar()
```

Técnica da *mesma classe*

Para efeitos de reprodução, por exemplo, um objecto pode precisar de saber se um outro objecto, seu vizinho, é da mesma classe. Como fazer isso sem escrever um teste explícito de classe?

Faz-se simplesmente assim:

```
getClass() == vizinho.getClass()
```

Para criar o objecto-filho, que vai nascer, fazer:

```
getClass().newInstance()
```

Conclusão

Todas estas técnicas apontam no mesmo sentido:

- Para escrever um programa extensível devemos:
  - **fugir ao concreto** (evitando testar explicitamente a classe concreta dos objectos a relacionar);
  - **procurar o abstracto** (inventando e usando conceitos abstractos).

Factorização

Um **sistema factorizado** é um sistema sem código repetido, ou seja, sem redundância. Num sistema factorizado, código que em princípio apareceria repetido em várias classes, fica concentrado (ou seja, *factorizado*) numa superclasse, ficando disponível por herança.

A factorização tem muitas vantagens:

- Antes de mais nada, contribui para a extensibilidade dos programas.
- Aumenta o nível de generalidade das ideias usadas.
- Torna os programas mais compactos e mais bem organizados.
- Facilita a correcção de erros.
- Durante o desenvolvimento dos programas, reduz a probabilidade de introdução de inconsistências.

Paradigma orientado pelos objectos

O paradigma orientado pelos objectos responde à necessidade prática de se escrever código extensível e

factorizado. Repare: convém que os programas sobrevivam à passagem do tempo, que sejam fáceis de adaptar quando as regras do mundo envolvente mudam.

Na base do paradigma orientado pelos objectos estão algumas descobertas geniais, que se podem resumir no seguinte:

- Uma linguagem que suporte os mecanismos de **herança**, **subtipo** e **envio de mensagem** apoia eficazmente a escrita de código extensível e factorizado.

Teórica 22

Detalhes do mecanismo de herança em Java

O **mecanismo de herança** permite reutilizar nas subclasses, as variáveis de instância e os métodos de instância definidos nas superclasses.

Note que nunca são herdados os construtores.

Também nunca são herdadas as variáveis de classe e os métodos de classe (ou seja, as variáveis e métodos **`static`**).

Detalhes da herança de métodos de instância

A generalidade dos métodos de instância das superclasses são herdados pelas subclasses, ficando a fazer parte destas. Exceptuam-se:

- os métodos privados
- os métodos não-privados que são redefinidos

Um subclasse não tem qualquer forma de acesso aos métodos não-herdados privados (pois claro, eles são *privados*!). Também não consegue aceder directamente aos métodos não-herdados redefinidos. Mas existe uma forma especial de acesso indirecto aos métodos não-herdados redefinidos da *superclasse imediata* (e só desta), usando o identificador **`super`**.

No exemplo que se segue, o método `m()` de A é herdado pelas as classes B e C. O método `z()` de B também é herdado pela classe C.

O método `a()` de A não é herdado pelo facto de ser privado. O método `p()` de A não é herdado pois é redefinido em B. O método `p()` de B não é herdado pois é redefinido em C.

O método `a()` de C não é considerado uma redefinição do método `a()` de A, pois este último é privado (só se redefinem métodos não-privados).

Repare como o método `z()` de B acede ao método (redefinido) `p()` de A, usando a palavra **`super`**.

```
class A {
    protected int x, y ;
    private int z ;
    public void m() { z = 0 ; this.p() ; }
    private void a() { }
    public void p() { System.out.println(1) ; }
}

class B extends A {
    public float y ;
    public void z() { super.p() ; }
    public void p() { System.out.println(2) ; }
}

class C extends B {
    private double y ;
    public void p() { System.out.println(3) ; }
    public void a() { y = super.y + ((A)this).y ; }
```

Detalhes da herança de variáveis de instância

Todas as variáveis de instância das superclasses (mesmo as variáveis `private`) são herdadas pelas subclasses, ficando a fazer parte dos seus objectos. No entanto a subclasse fica sem acesso

- às variáveis herdadas privadas

e apenas com acesso indirecto

- às variáveis herdadas não-privadas que são redefinidas

As variáveis herdadas, consideradas neste último caso, dizem-se **ocultas** (em Inglês: **hidden** ou **shadowed**). O acesso a uma variável oculta originária da superclasse imediata efectua-se usando da palavra **`super`**. O acesso a uma variável oculta originária duma superclasse mais acima na hierarquia, efectua-se fazendo um cast de **`this`** para o tipo que essa superclasse constitui.

No exemplo anterior, todas as variáveis são herdadas (esta é a regra geral, lembra-se?). Em termos de acesso, a variável `x` de A fica directamente acessível nas subclasses B e C. A variável `y` de A fica indirectamente

acessível em B e C por ser redefinida em B. A variável `z` de A fica inacessível em todas as subclasses por ser privada. A variável `y` de B fica indirectamente acessível em C por ser redefinida.

Repare que os objectos da classe A têm três variáveis `[x ; y ; z]`; os objectos da classe B têm quatro variáveis `[x ; super.y ; z ; y]` estando o `z` inacessível; os objectos da classe C têm cinco variáveis `[x ; ((A)this).y ; z ; super.y ; y]` estando o `z` inacessível.

O método `a()` de C exemplifica dois acessos indirectos a variáveis ocultas.

`this`, ou a reinterpretação dos métodos herdados nas subclasses

Quando um método é herdado, o código do seu corpo é reinterpretado no contexto da subclasse que o herda. Talvez não pareça, mas as consequências dessa reinterpretação concentram-se todas no identificador **`this`**.

No exemplo, analisemos a ocorrência de **`this`** no corpo do método `m()`. Dentro da classe A, considera-se que essa ocorrência de **`this`** representa um objecto da classe A. Mas o método `m()` é herdado pela classe B, e dentro de B a mesma ocorrência de **`this`** já representa um objecto da classe B. O método `m()` é também herdado pela classe C, e dentro de C a mesma ocorrência de **`this`** representa agora um objecto da classe C.

Assim se explica que as três mensagens seguintes produzam os resultados indicados, apesar do código do método `m()` nunca mudar (só muda o **contexto** da sua utilização):

```
(new A()).m()   -->  1
(new B()).m()   -->  2
(new C()).m()   -->  3
```

`super`, ou o acesso a componentes escondidas

A palavra **`super`** foi inventada para permitir o acesso aos métodos e variáveis não-privados da superclasse imediata que ficam *escondidos* em virtude de redefinição.

No exemplo, o método `z()` de B usa a mensagem **`super`**`.p()` com o objectivo de aplicar ao receptor da mensagem o método `p()` de A. Assim se explica que a mensagem seguinte produza o resultado indicado:

```
(new B()).z()   --->  1
```

Diferença entre `this` e `super`

Dentro de qualquer método, ambos os nomes, **`this`** e **`super`**, representam o *objecto do método* (ou seja, o *receptor da mensagem*). Temos portanto uma situação, um pouco confusa, em que uma mesma entidade é conhecida por dois nomes diferentes. **A diferença** entre **`this`** e **`super`** tem a ver com a forma como as mensagens que se enviam para o objecto são tratadas.

- As mensagens enviadas para **`this`** invocam os métodos que se encontram na **classe do receptor da mensagem**;
- As mensagens enviadas para **`super`** invocam os métodos que se encontram na **superclasse imediata da classe onde a palavra `super` aparece escrita**.

Assim, repare que **`this`** tem um carácter *variável* (i.e. *dinâmico*) e que `super` tem um carácter *fixo* (i.e. *estático*):

- Considerando a ocorrência de **`this`** no método herdado `m()`, verifica-se o seguinte: na classe A, essa ocorrência de **`this`** representa um objecto da classe A; na classe B, essa ocorrência de **`this`** representa um objecto da classe B; na classe C, essa ocorrência de **`this`** representa um objecto da classe C. Assim, a classe a que `this` pertence é *variável*, num certo sentido.
- Pelo contrário, a classe a que se considera `super` pertencer é *fixa*: concretamente, trata-se da superclasse imediata da classe **onde a palavra `super` aparece escrita**. Repare que a palavra `super` foi inventada para permitir o acesso a definições escondidas, um problema de natureza estática, não de natureza dinâmica.

Esta semântica de `super` até dá jeito: sempre que escrevemos a palavra `super` numa classe, estamos claramente a referir-nos à superclasse imediata da classe que estamos a escrever, e mais nada.

No exemplo, apesar do método `z()` ser herdado pela classe C, no contexto da classe C o nome `super` continua a ser tratado como se representasse um elemento da classe A (por ser na classe B que `super` aparece escrito). Assim se explica que a mensagem seguinte produza o resultado:

```
(new C()).z()   --->  1
```

Precauções

Precaução 1

Nem todos os livros se entendem relativamente à definição do termo *herança*.

Nós adoptamos a visão de que *se consideram herdadas todas as componentes da superclasse que são reutilizadas na implementação dos objectos da subclasse*. Esta visão permite discutir a implementação dos objectos com maior clareza.

A outra visão considera que *só são herdadas as componentes que ficam directamente visíveis na subclasse*. Esta visão é confusa pois obriga a esclarecer que, certas componentes da superclasse, apesar de se dizerem *não-herdadas*, são reutilizadas na implementação dos objectos das subclasses, estando disponíveis para acesso indirecto dentro da subclasse.

Precaução 2

Muitos livros livros sobre Java dão informação errada sobre a semântica da palavra `super`. Dizem geralmente que quando se envia uma mensagem para `super`, o método a activar é procurado na superclasse imediata do receptor, o que é um erro grave. O correcto seria dizer que **o método é procurado na superclasse imediata da classe onde a palavra `super` aparece escrita**.

Infelizmente, o livro escrito em Português indicado na bibliografia adicional da cadeira contém este erro. Correcta está a especificação *oficial* da linguagem Java, disponível na página da cadeira. Também está correcto o livro recomendado para a cadeira, disponível na página da bibliografia da cadeira.

Teórica 23

Packages

Um **package** é um conjunto de interfaces e classes, geralmente relacionadas.

Na plataforma Java 5.0 existem mais de 150 packages predefinidos. Os três mais usados são os seguintes:

- `java.lang` -- Colecção de classes e interfaces de que a própria linguagem Java depende: classes String, Integer, Boolean, Object, NullPointerException, etc.; interfaces Cloneable, Comparable, etc.
- `java.io` -- Input/output.
- `java.util` -- Estruturas de dados de utilidade geral, e.g. interface List e classe Vector.

Referenciação de packages

Para referenciar um elemento dum package, pode usar-se directamente uma referência qualificada, como em

```
java.io.System.out.println("ola") ;
```

Mas é bastante mais prático introduzir primeiro uma instrução de importação, como em

```
import java.io.* ;
```

para, depois, não ser necessário usar prefixos qualificadores.

O package `java.lang` é sempre automaticamente importado por qualquer programa.

Nomes dos packages

Ao nome dum package, digamos `java.io`, corresponde sempre um *pathname* relativo no sistema de ficheiros: no caso do package `java.io`, o *pathname* é `java/io`.

No Unix, a variável de ambiente CLASSPATH contém uma lista de directorias nas quais se assume que os packages estão guardadas e onde serão procuradas quando forem usadas.

Declaração do package duma classe ou interface

Se um ficheiro, contendo classes e interfaces, for iniciado por uma declaração do género

```
package games.monsters ;
```

então todas as classes e interfaces incluídas nesse ficheiro são declaradas como fazendo parte do package referido. No entanto a instalação das classes e interfaces não é automática: eles têm de ser arrumadas manualmente dentro das directorias correctas.

Criação de novos packages

Normalmente, o programador de Java cria novos packages para definir uma biblioteca pessoal de interfaces e classes, ou então para organizar um projecto de programação.

Interfaces e classes públicas

Em Java, um ficheiro pode conter, no máximo, uma única classe ou uma interface públicas. Isso significa que um package que contenha diversas classes e interfaces públicas, terá de ser definido em vários ficheiros.

Um ficheiro que defina uma classe ou interface públicas tem obrigatoriamente o mesmo nome dessa entidade pública (e ainda mais a extensão ".java"). Isso permite que o compilador e o sistema de execução de Java saibam que, por exemplo, a classe pública "Ola" se encontra definida dentro dum ficheiro chamado "Ola.java", e que o correspondente código objecto está armazenado num ficheiro chamado "Ola.class".

Outra vez as classes

Vamos agora fazer uma descrição completa do que é uma classe em Java. Em Java, uma **classe** é:

- **É um gerador de objectos extensível** (extensível por causa ao mecanismo de herança).
- **É um tipo** (representa o conjunto de todos os objectos gerados por essa classe e ainda pelas suas subclasses).
- **É um objecto especial**. Sim, em Java uma classe é um objecto, mais concretamente uma instância da classe Class. Uma classe possui as suas próprias *variáveis de classe* e os seus próprios *métodos de classe*.

Modificadores de classe

Os principais são três:

- `public` -- Indica que a classe pode ser acedida fora do package à qual pertence. Este modificador também se aplica a interfaces.
- `abstract` -- Indica que a classe é abstracta, ou seja, que pode conter métodos abstractos. Uma classe abstracta também não admite instâncias.
- `final` -- Indica que não é possível herdar a partir da classe.

Existem ainda os modificadores `private`, `protected` e `static`, que se podem aplicar apenas a classes internas a outras classes (*nested classes*, uma novidade introduzida no Java 1.1). Uma classe interna `static` pode aceder apenas às componentes de classe da classe envolvente; uma classe interna não `static` pode aceder a todas as componentes da classe envolvente.

Componentes de classe

Há cinco espécies de componentes de classes:

- **Variáveis de instância** -- Caracterizam a estrutura das instâncias da classe.
- **Métodos de instância** -- Caracterizam o comportamento das instâncias da classe.
- **Variáveis de classe** -- Pertencem à classe e são acessíveis a todas as suas instâncias. Também são acessíveis do exterior, se forem declaradas públicas. Exemplo: A classe System contém uma variável de classe pública chamada `out`, usada por exemplo em `System.out.println("ola") ;`.
- **Métodos de classe** -- Pertencem à classe e são acessíveis a todas as suas instâncias. Também são acessíveis do exterior, se forem declarados públicos. Exemplo: ver a descrição do centro de serviços Math, dois parágrafos abaixo.
- **Construtores** -- São métodos de classe especiais que servem para inicializar as instâncias da classe, no momento da criação.

Centro de serviços: dá-se por vezes este nome a uma classe sem instâncias que só contém variáveis e métodos de classe, alguns dos quais públicos.

A classe Math da plataforma Java é um centro de serviços que disponibiliza uma grande lista de operações matemáticas e duas constantes úteis: PI e E. Como todas as componentes são `static`, as respectivas mensagens são enviadas para Math, como em `Math.max(2,3)`.

Modificadores de componentes de classe

Estes são os mais importantes:

- `public` -- Indica que a componente pode ser acedida por quem tiver acesso à sua classe.
- `protected` -- Indica que a componente pode ser acedida dentro de todo o package da sua classe, e também dentro de todas as subclasses da sua classe.
- `private` -- Indica que a componente só pode ser acedida dentro da sua classe.
- *nenhum dos três anteriores* -- Indica que a componente pode ser acedida dentro do package envolvente.
- `static` -- Indica que se trata duma *componente de classe*. A ausência deste modificador indica que se trata duma *componente de instância*.
- `final` -- No caso duma variável, indica que se trata duma *constante*, a qual tem, portanto, de ter uma expressão de inicialização associada. No caso dum método, indica que o método não pode ser redefinido nas subclasses, apesar de ser herdado se não for privado. [Um exemplo interessante: todos os arrays contêm uma variável de instância `final` chamada *length*.]
- `abstract` -- Este modificador aplica-se só a métodos. Indica que o método é abstracto, i.e. que não tem implementação.
- `native` -- Este modificador aplica-se só a métodos. Indica que o método foi implementado noutra linguagem, e.g. C, C++, Assembly. O programa fica dependente da máquina para a qual os métodos foram compilados.
- `syncronized` -- Este modificador aplica-se só a métodos. Num ambiente de programação concorrente, garante-se a exclusão mútua de todos os métodos `syncronized` pertencentes ao mesmo objecto.

Construtores

Os **construtores** são métodos de classe especiais que servem para inicializar as instâncias da classe, logo que são criadas usando `new`. Os construtores têm sempre o nome da classe a que pertencem.

O modificadores `public`, `protected` e `private` podem ser aplicados a construtores. Se todos os construtores duma classe forem privados, então não é possível criar instâncias dessa classe no seu exterior.

Construtor por omissão

Se uma classe não tiver qualquer construtor definido, então um *construtor por omissão*, sem parâmetros, é automaticamente inserido pelo compilador. No seu corpo, o construtor por omissão chama o construtor da superclasse `super()`, e depois inicializa as variáveis de instância.

Corpo dum construtor

A primeira instrução dum construtor pode ser uma instrução especial. Nesse caso, corresponde a uma chamada explícita de outro construtor da mesma classe, escrito como `this(`*`args`*`)`, ou a uma chamada explícita dum construtor da superclasse imediata, escrito como `super(`*`args`*`)`. Não são permitidas chamadas recursivas de construtores.

Se uma chamada explícita dum construtor não ocorrer no princípio do corpo dum construtor, então o compilador insere automaticamente a instrução `super()`.

Levando em conta todas regras, percebe-se que quando um objecto é criado, construtores de todas as superclasses da sua classe serão activados.

Finalizadores

Em Java não há destructores à maneira do C++, pois a linguagem Java dispõe dum sistema de gestão de memória automático. No entanto uma classe pode requerer a *finalização* das suas instâncias, bastando para isso que implemente um método com o seguinte cabeçalho:

```
void finalize() ;
```

Quando um objecto inútil está prestes a ser reciclado, a mensagem `finalize()` é-lhe automaticamente enviada para que ele possa executar uma acção final antes de desaparecer (*RIP - rest in peace*).

Teórica 24

Input/Output em Java

Streams

Na plataforma Java, as abstracções de I/O chamam-se **streams**. As streams são independentes dos dispositivos físicos concretos e dos formatos de armazenamento de dados usados: é nesse sentido que as streams são *abstracções*. Recordamos que em ML as abstracções de I/O se chamam *canais*.

Em Java, são suportados dois tipos de streams: **streams de bytes** (ou **binárias**) para lidar com dados em formato-máquina, e **streams de caracteres** (ou **de texto**) para lidar com texto legível por humanos. Neste último caso é usada a convenção Unicode, que permite escrever, por exemplo, em Russo, Hebraico ou Birmanês.

Escrita e leitura nas streams padrão

As habituais **três streams predefinidas** estão disponíveis através de três constantes públicas chamadas `in`, `out` e `err`, definidas na classe `System` do package `java.lang`. Esta classe é um *centro de serviços* (cf. def. aula 23) que oferece diversas variáveis e métodos de utilidade geral.

Escrita

Para escrever texto nos canais de saída padrão usam-se os métodos overloaded `print` e `println`. Eis alguns exemplos:

```
System.out.println(123) ;
System.out.println(123.34) ;
System.out.println("ola") ;
System.err.println("Store: Argumento " + i + " fora dos limites") ;
```

Leitura

Para ler de forma confortável texto do canal de entrada padrão, é preciso criar primeiro um `Scanner` sobre essa stream. Isso faz-se assim:

```
Scanner sc = new Scanner(System.in) ;    // Nunca criar vários scanners sobre a mesma
stream
```

Um scanner permite ler **`tokens`**, que são sequências de caracteres separados por delimitadores. Eis algumas leituras de tokens:

```
String aToken = sc.next() ;          // Lê um token
int aInt = sc.nextInt() ;            // Lê um token inteiro
double aDouble = sc.nextDouble() ;   // Lê um token real
```

Por defeito, os **delimitadores** são os espaços em branco, tabs e mudanças de linha.

Para validar o input, é possível **testar o tipo do próximo token** a ser lido. Por exemplo:

```
if( sc.hasNextDouble() )    // Verifica se próximo token é um real
    aDouble = sc.nextDouble() ;
```

O método `nextLine()` permite **ler uma linha completa**. O seguinte código copia um ficheiro completo, linha a linha:

```
try {
    for(;;)
        System.out.println(sc.nextLine()) ;
} catch( NoSuchElementException e ) {}
```

Repare que a ocorrência da excepção `NoSuchElementException` indica que não há mais linhas para ler.

A classe `Scanner` pertence ao package `java.util` e foi introduzida no Java 5.0.

O package `java.io`

Em Java é possível abrir streams sobre ficheiros, strings, filtros, pipes, sockets, URL, etc. O package `java.io` inclui cerca de cinquenta classes que implementam diversas formas de streams. Para ter acesso a estas classes

é necessário incluir a instrução `import java.io.*;` no início do programa.

Qualquer classe que implemente uma forma particular de stream deve ser subclasses de alguma das seguintes quatro *classes abstractas* do package `java.io`:

```
InputStream
OutputStream
Reader
Writer
```

No caso das streams binárias, qualquer classe que as implemente deverá ser subclasse de `InputStream` ou `OutputStream`.

No caso das streams de texto, qualquer classe que as implemente deverá ser subclasse de `Reader` ou `Writer`.

As classes `FileWriter` e `FileReader`

Para ler e escrever ficheiros de texto usam-se instâncias das classes `FileReader` e `FileWriter`, respectivamente.

Para ler e escrever ficheiros binários usam-se instâncias das classes `FileInputStream` ou `FileOutputStream`, respectivamente.

Todas estas classes são de bastante baixo nível: as operações disponíveis são muito básicas e em pequeno número.

Métodos mais importantes da classe `FileWriter`

```
public void write(char[] s) ;
public void write(char[], int off, int len) ;
public void write(int c) ;        // escreve char
public void write(String i) ;
public void close() ;
```

Exemplo de utilização da classe `FileWriter`

```
import java.io.*;

class Test {
    static public void main(String[] args) {
        int i = 123 ;
        try {
            FileWriter w = new FileWriter("ola.txt") ;
            w.write((new Integer(i)).toString()+"\n") ;
            w.close() ;
        } catch(IOException e) {
        // Tratar a excepção. Por exemplo:
            System.out.println(e.getMessage()) ;
            e.printStackTrace() ;
            System.exit(1) ;
        }
    }
}
```

Métodos mais importantes da classe `FileReader`

```
public int read() ;            // lê char
public int read(char[] s) ;
public int read(char[] s, int off, int len) ;
public void close() ;
```

Exemplo de utilização da classe `FileReader`

```
import java.io.*;

class Test {
    static public void main(String[] args) {
        char[] a = new char[100] ;
        try {
            FileReader r = new FileReader("ola.txt") ;
            r.read(a) ;
            r.close() ;
```

```
                System.out.print(a) ;
            } catch(IOException e) {
                // Tratar a excepção.
            }
        }
    }
```

Exemplo de cópia de ficheiro de texto

```
import java.io.*;

class Test {
    static public void main(String[] args) {
        char[] buffer = new char[1024] ;
        int len ;
        try {
            FileReader r = new FileReader("ola.txt") ;
            FileWriter w = new FileWriter("ole.txt") ;
            while( (len = r.read(buffer)) != -1 )
                w.write(buffer, 0, len) ;
            r.close() ;
            w.close() ;
        } catch(IOException e) {
            // Tratar a excepção.
        }
    }
}
```

As *classes embrulho* do package `java.io`

O package `java.io` está inteligentemente desenhada. Algumas das classes funcionam como *classes embrulho* (*wrapper classes*) e destinam-se a acrescentar funcionalidade a streams existentes.

A classe embrulho `PrintWriter`

A classe `PrintWriter` é uma classe embrulho. Permite adicionar a qualquer stream de texto a capacidade de escrita de representações legíveis por humanos de diversos tipos de dados (inteiros, reais, booleanos, etc.).

Métodos mais importantes da classe embrulho `PrintWriter`

```
public void print(String s) ;
public void print(char c) ;
public void print(int i) ;
public void print(long l) ;
public void print(float f) ;
public void print(double d) ;
public void print(boolean b) ;

public void println() ;

public void println(String s) ;
public void println(char c) ;
public void println(int i) ;
public void println(long l) ;
public void println(float f) ;
public void println(double d) ;
public void println(boolean b) ;

public void close() ;
```

Exemplo de utilização da classe embrulho `PrintWriter`

(Versão melhorada do 1ºexemplo.)

```
import java.io.*;

class Test {
    static public void main(String[] args) {
        int i = 123 ;
        try {
            PrintWriter w = new PrintWriter(new FileWriter("ola.txt")) ; // Um
PrintWriter embrulhando um FileWriter
```

```
                w.println(i) ;
                w.close() ;
            } catch(IOException e) {
                // Tratar a excepção.
            }
        }
    }
```

A classe embrulho `BufferedWriter`

A classe `BufferedWriter` permite interpor um buffer à saída de qualquer stream de texto. Por exemplo, faria todo o sentido usar no exemplo anterior:

```
PrintWriter w =
    new PrintWriter(new BufferedWriter(new FileWriter("ola.txt")));
```

Na classe `FileReader` não há qualquer método para ler linhas completas, mas na classe `BufferedReader` já há: o método chama-me `readLine`.

Outras classes embrulho

A classe `DataOutputStream` permite adicionar a qualquer stream de output binária a capacidade de escrever representações optimizadas para serem lidas pela máquina, de diversos tipos de dados (inteiros, reais, booleanos, etc.)

A classe `InputStreamReader` permite que uma stream binária seja usada como stream de texto.

Existem ainda mais classes embrulho.

Leitura de input formatado nas versões antigas do Java

No Java 5.0 foi introduzida a classe `Scanner`, a qual facilita imenso a leitura de input formatado.

Nas versões anteriores do Java, o programador tinha de ler linhas completas de texto usando o método `readLine` do `BufferedReader` e depois decompor cada linha nas suas partes usando a classe `StringTokenizer`.

Exemplo de leitura e escrita dum inteiro em Java 1.4

```
try {
    BufferedReader r = new BufferedReader(new InputStreamReader(System.in)) ;
    String s = r.readLine() ;
    int i = (new Integer(s)).intValue() ;
    System.out.println(i) ;
} catch(IOException e) { }
```

Exemplo de leitura e escrita dum inteiro em Java 5.0

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

Teórica 25

Tipo duma expressão e tipo dum valor

Tipo duma expressão

É o tipo da expressão, tal como é conhecido pelo compilador. Por vezes chama-se **tipo estático** a esse tipo, para enfatizar que é determinado em tempo de compilação.

Exemplo

Considere as seguintes declarações:

```
Animal a ;
int i, j ;
```

O tipo das seguintes expressões é o que se indica:

```
a           // tipo Animal
i + j       // tipo int
a + 1       // não tem tipo; é um erro
```

Tipo dum valor

É o tipo que, em tempo de execução, fica associado a um valor. Por vezes chama-se **tipo dinâmico** a este tipo, para enfatizar que só é conhecido em tempo de execução.

Exemplo

Considere as seguinte declaração:

```
Animal a = new Gato() ;
```

A seguir a esta declaração, antes da variável ser alterada, a variável `a` refere um valor de tipo `Gato`.

Relação entre estes dois tipos

Em Java, verifica-se sempre a seguinte propriedade fundamental:

- Uma expressão de tipo estático A, quando avaliada produz sempre um valor dum tipo dinâmico B, tal que B « A.

No caso particular de A não admitir subtipos, então A e B são obrigados a coincidir. Em Java, os únicos tipos que não admitem subtipos são os *tipos primitivos* e os tipos declarados por classes `final`.

Mecanismo de chamada de funções em linguagens procedimentais

No contexto duma linguagem procedimental (e.g. C, Pascal, ML), consideremos o problema:

- *Como se determina qual a função invocada numa chamada* `F(args)`*?*

Nas linguagens procedimentais, este problema é simples de resolver porque:

- As funções são identificadas por *nome*, apenas.
- Em cada contexto sintáctico, nunca duas funções têm o mesmo nome.

A descoberta da função que `F` representa em `F(args)` é imediata:

- O nome `F` é *ligado* à única função de nome `F` que está visível no contexto sintáctico da chamada.

A ligação entre o nome `F` e a função por ele designada efectua-se em tempo de compilação, pelo que se diz que esta é uma **ligação estática** [Em Inglês: *static binding* ou *early binding*].

Mecanismo de envio de mensagens em Java

No contexto da linguagem Java, consideremos agora o problema:

- *Como se determina qual o método invocado numa mensagem* `exp.F(args)`*?*

Na linguagem Java, este problema é relativamente complicado de resolver porque:

- Os métodos são identificados por *assinatura*.

- É permitido que em classes distintas possam ocorrer métodos com a mesma assinatura.
- Uma expressão de tipo estático A pode produzir valores de variados tipos dinâmicos B (com B « A).

A descoberta do método que `F` representa em `exp.F(args)` efectua-se em duas fases:

- **Fase estática** (em tempo de compilação): A mensagem `exp.F(args)` é *ligada* a uma *assinatura* `A`, escolhida em função do tipo estático de `exp`.
- **Fase dinâmica** (em tempo de execução): A assinatura `A` é *ligada* a um método com essa *assinatura*, escolhido em função do tipo dinâmico do valor de `exp`.

A ligação entre a assinatura `A` e o método por ela designado efectua-se em tempo de execução, pelo que se diz que esta é uma **ligação dinâmica** [Em Inglês: *dynamic binding* ou *late binding*].

Vamos ver agora os detalhes mais importantes da fase estática e da fase dinâmica. A explicação completa encontra-se na especificação oficial da linguagem (disponível online na página da cadeira) onde ocupa cerca de 20 páginas (trata-se do capítulo "15.11 Method Invocation Expressions").

Fase estática

Considerando os argumentos `args` e considerando o protocolo associado ao tipo estático da expressão `exp`, liga-se a mensagem `exp.F(args)` à assinatura *mais específica* para `F(args)` que conste desse protocolo. A *assinatura mais específica* é aquela com que `F(args)` melhor se consegue conformar, no sentido em que é minimizado o número de conversões implícitas dos argumentos `args`.

Se não existir qualquer assinatura com a qual `F(args)` se consiga conformar, então a mensagem `exp.F(args)` é considerada *inválida* e é gerado um erro de compilação.

Se existirem diversas assinaturas com as quais `F(args)` se consiga conformar e se nenhuma delas for *mais específica*, então a mensagem `exp.F(args)` é *ambígua* e é gerado um erro de compilação.

Fase dinâmica

Liga-se a assinatura `A` ao método com essa assinatura que se encontra na classe do objecto que resulta da avaliação de `exp`. É só!

Exemplo

Considere o seguinte programa incompleto em Java:

```
interface Bag {
    int Size() ;
    int Get(int i) ;
    void Add(int v) ;
    int Many(int v) ;
    Bag Union(Bag s) ;
}
class BagClass implements Bag { ...  }

interface Set extends Bag {
    boolean Belongs(int v) ;
    Set Union(Set s) ;
}
class SetClass extends BagClass implements Set { ...  }
class SetClass2 extends SetClass implements Set { ...  }

class TestBags {
    public static void main(String [] args) {
        Bag bb = new BagClass() ;
        Bag bs = new SetClass() ;
        Set ss = new SetClass() ;
        Set ss2 = new SetClass2() ;

        bs.Belongs(3) ; // ERRO: mensagem inválida
        ss.Belongs(3) ; // assinatura mais específica: Belongs(int)
                        // classe do objecto: SetClass
        ss2.Belongs(3) ;// assinatura mais específica: Belongs(int)
                        // classe do objecto: SetClass2

        bb.Union(bs) ;  // assinatura mais específica: Union(Bag)
                        // classe do objecto: BagClass
        bb.Union(ss) ;  // assinatura mais específica: Union(Bag)
                        // classe do objecto: BagClass
```

```
            bs.Union(bb) ;  // assinatura mais específica: Union(Bag)
                            // classe do objecto: SetClass
            bs.Union(ss) ;  // assinatura mais específica: Union(Bag)
                            // classe do objecto: SetClass

            ss.Union(bb) ;  // assinatura mais específica: Union(Bag)
                            // classe do objecto: SetClass
            ss.Union(bs) ;  // assinatura mais específica: Union(Bag)
                            // classe do objecto: SetClass
            ss.Union(ss) ;  // assinatura mais específica: Union(Set)
                            // classe do objecto: SetClass

            ss.Union(ss2) ; // assinatura mais específica: Union(Set)
                            // classe do objecto: SetClass
            ss2.Union(bs) ; // assinatura mais específica: Union(Bag)
                            // classe do objecto: SetClass2
            ss2.Union(ss) ; // assinatura mais específica: Union(Set)
                            // classe do objecto: SetClass2
        }
    }
```

Dentro da função `main` da classe `TestBags` encontram-se diversos exemplos de mensagens. Em cada um dos casos, indicamos qual a *assinatura mais específica*, obtida na fase estática, e qual a *classe do objecto*, obtida na fase dinâmica, que vai determinar a escolha do método. Note que alguns dos casos são bastante subtis.

Teórica 26

Teste sobre o projecto de Java.

Teórica 27

Applets

Uma **applet** é um programa em Java capaz de correr dentro dum rectângulo, numa página WEB.

Uma applet é sempre programada como uma subclasse da classe `Applet`. Esta classe pertence ao package `java.applet` e é subclasse de `Panel`.

Eis a documentação da classe Applet. Uma applet geralmente tem necessidade de redefinir os métodos `init`, `start`, `stop` e `paint`.

Exemplo

```
import java.awt.*;
import java.applet.*;

public class MyApplet extends Applet {
    public void init() {
        // activado para inicializar a applet.
        super.init() ;
    }
    public void start() {
        // activado no inicio da execucao da applet.
        super.start() ;
    }
    public void stop() {
        // activado no final da execucao da applet.
        super.stop() ;
    }
    public void paint(Graphics g) {
        // actualização do rectangulp
        Font font = new Font("Times", Font.BOLD, 48) ;
        g.setFont(font) ;
        g.drawString("Ola' ola' ola'!!!", 30, 60) ;
    }
}
```

Instalação e execução

Assumindo que esta applet já se encontra compilada no ficheiro `MyApplet.class`, para a instalar numa página WEB usa-se o seguinte código HTML:

```
<applet code="MyApplet" width=400 height=80></applet>
```

Mais informação sobre applets

Em alguns web browsers a applet anterior só corre se for instalado um plugin para Java. Por exemplo, no Linux, usando o Galeon (ou o Mozilla) e o Java 5.0, é necessário estabelecer um link simbólico de dentro de "/usr/lib/mozilla/plugins" para "/usr/java/jdk1.5.0/jre/plugin/i386/ns7-gcc29/libjavaplugin_oji.so".

Muito mais informação sobre applets pode ser encontrada no capítulo 6 do livro "David J. Eck, Programming Using Java". Este livro está disponível *on-line* dentro do item "Bibliografia & Links" da página da cadeira.

O processo de conversão duma aplicação Java numa applet Java não é trivial.

Algumas tecnologias Java

Java Beans

Trata-se dum sistema de componentes - uma componente é um objecto reutilizável - que podem ser ligadas entre si usando uma ferramenta visual - editor de Beans, por exemplo Beans Builder, NetBeans ou Eclipse - para obter aplicações completas, sem ser necessária qualquer programação. Também se pode escrever software preparado para funcionar acoplando componentes à maneira de "plugins".

Qualquer objecto em Java que seja gerado por uma classe que suporte um certo conjunto pequeno de convenções é considerada uma componente. As componentes comunicam entre si usando *eventos*.

Disponível na versão standard do Java - J2SE.

A tecnologia correspondente da Microsoft chama-se ActiveX. O Java Beans inspirou-se no ActiveX, que já existia antes.

Servlets, JSP e Tomcat

Servlets - É uma espécie de applet que corre do lado do servidor e implementa um serviço.

JSP - JavaServer Pages: Mecanismo que permite criar dinamicamente páginas WEB, combinando HTML com scripts escritos em Java. (Semelhante mas mais simples e elegante que PHP e ASP)

Tomcat WEB server - É um servidor de páginas WEB desenvolvido para suportar especificamente servlets e JSPs.

Enterprise Java Beans (EJB)

Trata-se duma arquitectura de componentes orientada para a programação de servidores em ambiente empresarial. Permite o desenvolvimento rápido e simplificado de aplicações distribuídas, seguras e portáveis.

Só disponível na versão do Java para empresas - J2EE.

Swing

Arquitectura de componentes para desenvolver interfaces gráficas. As componentes são: janelas, painéis, botões, barras de scrolling, etc.

Baseia-se no Java Beans: as componentes gráficas são efectivamente *beans*.

Disponível na versão standard do Java - J2SE.

Java vs. .NET

A Microsoft desenvolveu uma tecnologia alternativa à do Java e designou-a de .Net. As duas tecnologias têm muitas semelhanças:

- Ambas fornecem um ambiente de execução que suporta gestão automática de memória, threads e uma máquina abstracta. A máquina abstracta chama-se JVM - Java Virtual Machine, no caso da plataforma Java, e VES - Virtual Execution System, no caso do .NET.
- Ambas oferecem uma biblioteca de classes, extensa e integrada.
- Através das linguagens Java e C#, os programadores têm acesso a todas a todas as funcionalidades da plataforma Java e da plataforma .NET, respectivamente.

A plataforma .NET fornece ainda o seguinte:

- Uma especificação de ficheiros-objecto que o compilador de qualquer linguagem pode usar para permitir o acesso às funcionalidades do .NET. Essa especificação chama-se CLS - The Common Language Specification.

A plataforma Java goza actualmente de imensa popularidade. A biblioteca de classes facilita muito o desenvolvimento de aplicações com sofisticados interfaces gráficos e que funcionam através da Internet em qualquer hardware e em qualquer sistema operativo.

A plataforma .NET consiste essencialmente numa nova fundação para o desenvolvimento de programas sobre o Windows, que dará à Microsoft espaço para crescer nos próximos anos. A biblioteca de classes da plataforma .NET é tão prática de usar que, hoje em dia, a maioria dos programadores profissionais em Windows usa o .NET (através do Visual Studio .NET da Microsoft ou do Delphi da Borland).

Windows

As duas tecnologias estão implementadas no Windows.

Linux

A empresa Sun sempre suportou a plataforma Java no Linux. Por seu lado, a Microsoft planeia suportar o .NET exclusivamente no Windows. No entanto existe um projecto de software livre, em rápida evolução, chamado Mono, que pretende implementar partes do .Net no Linux.

Programar numa linguagem OO

Se o programa modelar bem a realidade em que se insere o problema, o programa fica mais intuitivo e mais fácil de manter.

Como programar numa linguagem orientada pelos objectos para conseguir obter um programa de elevada qualidade?

- Identificar no problema quais são as *entidades*.

- Determinar, para cada tipo de entidade, quais as acções que lhes são *naturais*.
- Escrever código bem factorizado e extensível.

Numa linguagem não-OO também se pode procurar imitar este estilo, mesmo que a linguagem não ajude muito.

Comparação entre um engenheiro civil e um engenheiro informático

O engenheiro civil deve conceber estruturas:

- resistentes
- duráveis
- fáceis de manter
- de concepção modular
- tão económicas quanto possível

O engenheiro informático deve conceber software:

- extensível (para ser durável)
- reutilizável
- modular
- que imite o mundo em que se insere o problema
- fácil de manter
- seguro
- portável
- eficiente

Aprofundar o conhecimento da linguagem Java

- **Generics in the Java Programming Language**
- **Construção for-each**
  - Construção que permite iterar sobre os elementos duma colecção (Vector, List, Queue, etc).
- **Enumerados**
  - Tipo cujos valores são identificadores. Exemplo:

```
enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }
```

- **Importação de membros estáticos duma classe**
  - Exemplo:

```
import static java.lang.Math.* ;
...
double r = cos(PI * theta);
```

- **Tipo de retorno dos métodos covariante**
  - Quando se redefine um método numa subclasse, é possível especializar o tipo de retorno. Lembre-se que a assinatura dum método é constituida por *nome/tipo dos argumentos*; o resultado não faz parte. Exemplo:

```
class A {
    Animal m(int i) { ... }
}

class B extends A {
    Gato m(int i) { ... } // considera-se uma redefinição do método m(int)
}
```

Java 6.0

Sairá na Primavera de 2006.

Já existem uma versão pré-alpha com algumas novidades ao nível da plataforma Java. Mas ainda é muito cedo para dizer qual será a funcionalidade do sistema final.

Índice de funções e classes