Capítulo 6- Introdução à programação OO em Scheme

Objectos, mensagens, métodos e classes
Objectos e heranças
Objectos com história; classe-caixa
Objectos com história; classe-coleccao
Abstracções com objectos - classe-pilha

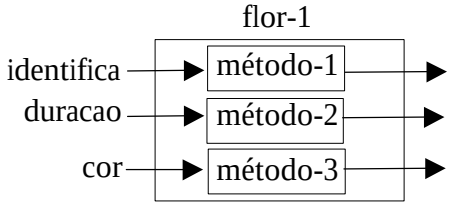
Exercícios e exemplos
Classe-fila
Classe-lista-circular
Projecto - Jogo de Dados
Projecto - Vidas

Na *programação orientada por objectos*, os objectos são entidades modulares que respondem a *mensagens*. Para cada *mensagem*, o *objecto* tem previsto um *método*, através do qual lhe responde. A partir de um objecto é possível derivar um novo objecto. Diz-se que este novo objecto *envia mensagens* para o objecto que esteve na sua origem. Também é habitual dizer-se que o novo objecto *herda*, total ou parcialmente, os métodos daquele objecto.
O grande interesse da programação *orientada por objectos* reside, em grande parte, na modularidade que apresenta, pois cada objecto surge como se fosse um módulo, e no reaproveitamento de código existente que este paradigma de programação oferece. O Scheme não é, propriamente, uma linguagem vocacionada para a programação orientada por objectos, característica que é mais notória em linguagens como Java, C++, Smalltalk ou Simula, mas permite, com relativa facilidade, introduzir os conceitos deste importante paradigma de programação.

1- Objectos, mensagens, métodos e classes

Um *objecto* reage a um certo conjunto de *mensagens*, tendo para cada uma delas um *método*. Por exemplo, um objecto *flor* poderá reagir às mensagens *identifica*, *duracao* e *cor*, dispondo para tal dos métodos respectivos. À mensagem *identifica*, reage o objecto com um método que devolve o nome que lhe está associado. O método, correspondente à mensagem *duracao*, devolve o tempo médio de vida da flor, enquanto que o método *cor* devolverá a cor dessa flor.
Em Scheme, uma objecto *flor*, com a identificação *cravo*, de cor *amarelo*, e com a duração média de *20-dias*, é facilmente implementado através de um procedimento que se designa por *flor-1*[1].

```
(define flor-1
  (lambda msg
    (let ((m (car msg)))
      (cond
        ((equal? m 'identifica) 'cravo)
        ((equal? m 'duracao) '20-dias)
        ((equal? m 'cor) 'amarelo)))))
```



Nos comandos que se seguem, exemplifica-se o envio de mensagens ao objecto *flor-1* e as respostas dadas pelos métodos respectivos.

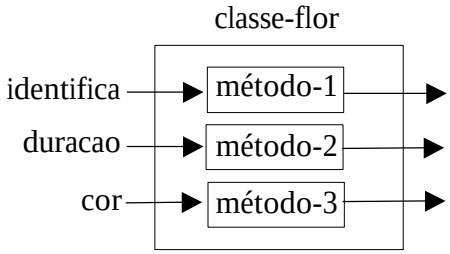
```
↦(flor-1 'identifica)
```

[1] O procedimento *flor-1* foi definido de forma a poder receber um número não fixo de argumentos, funcionalidade que não se justificava neste caso, mas que será extremamente útil na definição de outros objectos e que, por isso, já foi aqui adoptada

```
cravo
↦(flor-1 'duracao)
20-dias
↦(flor-1 'cor)
amarelo
```

Para evitar a reescrita de um objecto *flor* sempre que for necessário um novo objecto com estas características, define-se uma classe de objectos através do procedimento com a designação *classe-flor*. Uma chamada a este procedimento cria um objecto *flor*, dizendo-se que este objecto é uma *instância* de *classe-flor*.

```
(define classe-flor
  (lambda (nome tempo-vida cor)

    (lambda msg
      (let ((m (car msg)))
        (cond
          ((equal? m 'identifica)
           nome)
          ((equal? m 'duracao)
           tempo-vida)
          ((equal? m 'cor)
           cor))))))
```



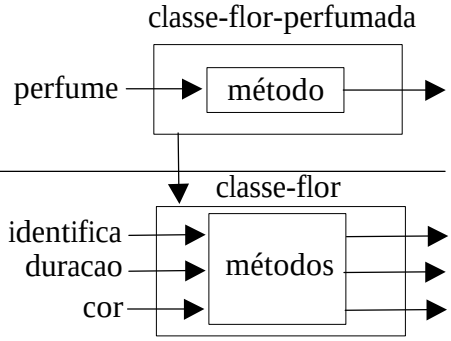
Naturalmente, o procedimento *classe-flor* tem como parâmetros as três características com que se definem estes objectos e devolve, por cada chamada, um procedimento semelhante ao designado por *flor-1*.

```
↦(define igual-a-flor-1 (classe-flor 'cravo '20-dias 'amarelo))
flor-1
↦(igual-a-flor-1 'identifica)
cravo
↦(define flor-rosa (classe-flor 'rosa '8-dias 'vermelha))
flor-rosa
↦(flor-rosa 'identifica)
rosa
```

2- Objectos e heranças

Também se podem criar objectos a partir de objectos de outras classes. Como exemplo, vai-se definir a *classe-flor-perfumada* que cria objectos com os métodos já considerados em *classe-flor*, e ainda o método *perfume*, que devolve a caracterização da fragrância associada ao objecto respectivo *(suave, doce e intenso)*. Como se pode ver, não vai ser necessário reescrever os métodos contidos em *classe-flor*.

```
(define classe-flor-perfumada
  (lambda (nome tempo-vida cor perfume)

    (let ((flor (classe-flor nome tempo-vida cor)))

      (lambda msg
        (let ((m (car msg)))
          (cond
            ((equal? m 'perfume)
             perfume)
            (else
              (apply flor msg)))))))))
```



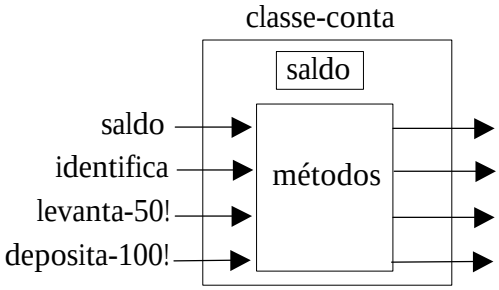
Da definição *classe-flor-perfumada* destaca-se a criação do objecto local *flor*, uma instância de *classe-flor*. Todas as mensagem que cheguem a um objecto da *classe-flor-perfurmada*, diferentes de *perfume*, serão redireccionadas para o objecto *flor*, através de *(apply flor msg)*[2]. Podemos assim dizer que *classe-flor-perfumada herda métodos* ou *passa mensagens* para *classe-flor*. Esta funciona como *classe ascendente* e aquela como *classe descendente*.

```
↦(define f-cravo (classe-flor-perfumada 'cravo '15 'branco 'doce))
f-cravo
↦(f-cravo 'perfume)
doce
↦(f-cravo 'duracao)
15
↦(f-cravo 'identifica)
cravo
```

3- Objectos com história; classe-caixa

Frequentemente, os objectos apresentam-se com uma história que é necessário preservar, como acontece no exemplo típico da conta bancária. Neste caso, é fundamental manter actualizado, pelo menos, o saldo da conta.

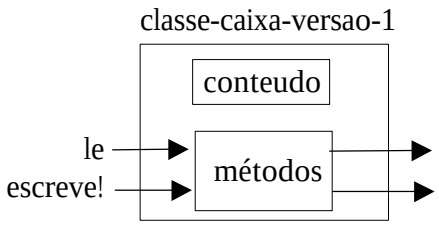
Os objectos *classe-conta* vão reagir às mensagens *saldo*, *identifica*, *levanta-50!* e *deposita-100!*, dispondo para tal dos métodos respectivos. À mensagem *saldo*, reage o objecto com um método que devolve o valor actual do saldo da conta bancária. À mensagem *identifica*, responde com um método que devolve o código associado à conta. O método correspondente à mensagem *levanta-50!*, retira 50 unidades ao saldo da conta e devolve o valor do saldo, após o levantamento. Finalmente, à mensagem *deposita-100!* corresponde um método que acrescenta 100 unidades ao saldo da conta e também devolve o valor de saldo, após o depósito.



Antes de atacar o problema da conta bancária, comecemos por tratar o caso genérico da classe que cria objectos do tipo *caixa*. Os objectos *caixa* contêm uma variável local que, no momento de criação de um objecto, é inicializada com um valor dado. Estes objectos reagem às mensagens *le* e *escreve!*, em que *le* devolve o conteúdo actual da variável local e *escreve!* actualiza esse conteúdo com um valor dado. É a partir dos objectos *caixa*, instâncias de *classe-caixa-versao-1*, que se derivam os objectos cuja história pode ser representada por um dado simples, um inteiro ou um real, como acontece com o saldo de uma conta bancária.

[2] Neste exemplo, *msg* tem apenas um elemento e poder-se-ia escrever simplesmmente *(flor (car msg))* em vez de *(apply flor msg)*. No entanto, mais à frente, encontraremos exemplos em que *msg* poderá conter vários elementos, onde justifica plenamente a solução utilizada, pelo que se decidiu desde já adoptá-la. Não esquecer que *(apply flor msg)* é equivalente a chamar o procedimento *flor* em que os argumentos são todos os elementos da lista *msg* e não se pode confundir com *(flor msg)*...

```
(define classe-caixa-versao-1
  (lambda (valor-inic)

    (let ((conteudo valor-inic))

      (lambda msg
        (let ((m (car msg)))
          (cond
            ((equal? m 'le) conteudo)
            ((equal? m 'escreve!)
             (set! conteudo (cadr msg))
             conteudo)))))))
```

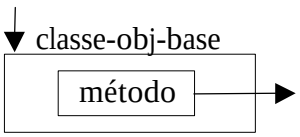


Convém verificar que o conteúdo de um objecto *classe-caixa-versao-1*, implementado como uma variável local de um procedimento, só é acessível do exterior através das mensagens *le* e *escreve!*. Em casos como este, diz-se que os dados estão *encapsulados*.

```
↦(define caixa-1 (classe-caixa-versao-1 50))
caixa-1
↦(caixa-1 'le)
50
↦(caixa-1 'escreve! 56)
56
↦(caixa-1 'le)
56
↦(caixa-1 'escrev! 78)
#f
```

Na última parte da interacção, *(caixa-1 'escrev! 78)*, observa-se a forma inadequada como o objecto *caixa-1* reage a um método que não foi previsto[3]. Para evitar situações desta natureza, define-se a *classe-obj-base* à qual poderão recorrer as outras classes quando algum dos seus objectos recebe um método que desconheça.

```
(define classe-obj-base
  (lambda ()
    (lambda msg
      (display (car msg))
      (display ": mensagem desconhecida!..."))))
```

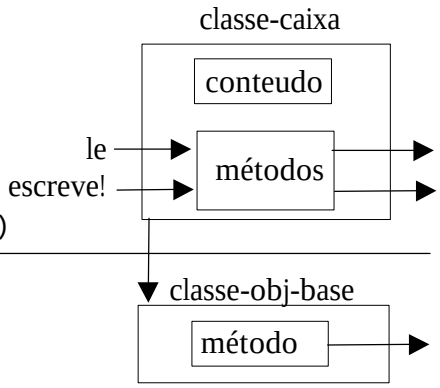



```
↦(define objecto (classe-obj-base))
objecto
↦(objecto 'le)
le: mensagem desconhecida!...
↦(objecto 'escreve!)
escreve!: mensagem desconhecida!...
```

Segue-se a redefinição de *classe-caixa* que passa a herdar os métodos de *classe-objecto-base*.

[3] Esta mensagem depende da implementação do Scheme utilizado. Esta foi obtida com o EdScheme. Mas, por exemplo, o DrScheme devolveria, na mesma situação, `no matching cond clause`

```
(define classe-caixa
  (lambda (valor-inic)

    (let ((conteudo valor-inic)
          (obj-base (classe-obj-base)))
      (lambda msg
        (let ((m (car msg)))
          (cond
            ((equal? m 'le) conteudo)
            ((equal? m 'escreve!)
             (set! conteudo (cadr msg))
             conteudo)
            (else (apply obj-base msg))))))))
```

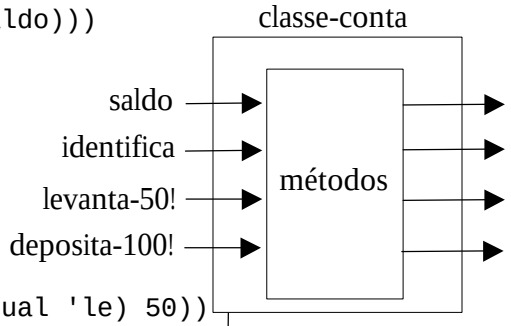



```
↦(define caixa-completa (classe-caixa 60))
caixa-completa
↦(caixa-completa 'le)
60
↦(caixa-completa 'escreve! 50)
50
↦(caixa-completa 'escrev 30)
escrev: mensagem desconhecida!...
```

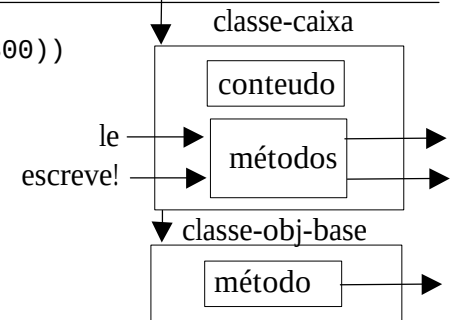
Com *classe-caixa* definida, vamos passar ao exemplo da conta bancária. A *classe-conta* tem como parâmetros um código e um saldo e define localmente um objecto *caixa* designado por *saldo-actual*. Com esta definição local, *classe-conta* passa a herdar os métodos de *classe-caixa* e, através desta, também os métodos de *classe-obj-base*. Os métodos a ter em conta são:

saldo - devolve o saldo actual da conta;
identifica - devolve o código associado à conta;
levanta-50! - retira 50 ao saldo e devolve-o actualizado;
deposita-100! - adiciona 100 em saldo e devolve-o actualizado.

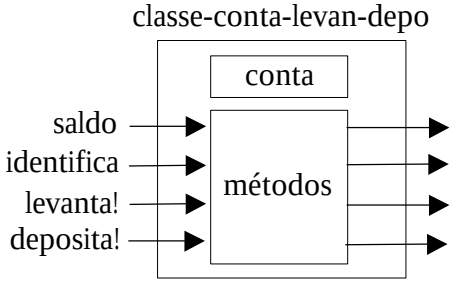
```
(define classe-conta
  (lambda (codigo saldo)

    (let ((saldo-actual (classe-caixa saldo)))
      (lambda msg
        (let ((m (car msg)))
          (cond
            ((equal? m 'identifica)
             codigo)
            ((equal? m 'saldo)
             (saldo-actual 'le))
            ((equal? m 'levanta-50!)
             (saldo-actual 'escreve!
                           (- (saldo-actual 'le) 50))
             (saldo-actual 'le))
            ((equal? m 'deposita-100!)
             (saldo-actual 'escreve!
                           (+ (saldo-actual 'le) 100))
             (saldo-actual 'le))
            (else (apply saldo-actual msg)))))))))
```




```
↦(define minha-conta (classe-conta 12345 1300))
minha-conta
↦(minha-conta 'identifica)
12345
```



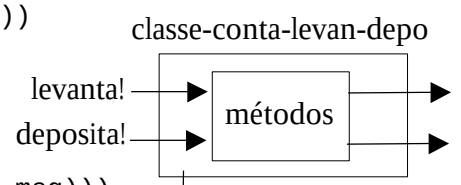
```
↦(minha-conta 'saldo)
1300
↦(minha-conta 'levanta-50!)
1250
↦(minha-conta 'deposita-100!)
1350
↦(minha-conta 'le)[4]
1350
↦(minha-conta 'sssssaldo)
sssssaldo: mensagem desconhecida!...
```



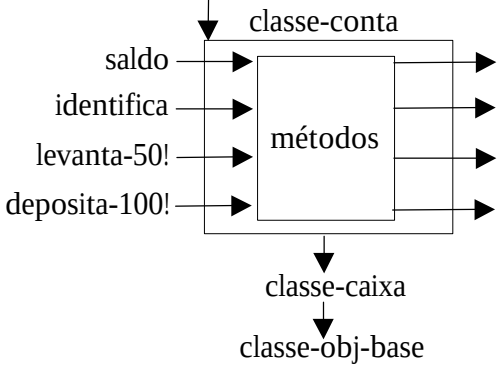
A conta bancária que se segue, designada por *classe-conta-levan-depo*, apresenta os mesmos métodos de *classe-conta*, mas dois deles são redefinidos: *levanta!* e *deposita!* não se limitam a manipular uma quantia fixa, como acontecia com *levanta-50!* e *deposita-100!*.

Encontramos finalmente uma classe cujas mensagens não apresentam um comprimento fixo, uma vez que *levanta!* e *deposita!* passam a especificar, para além do nome do método, a quantia a manipular. São classes como esta que justificam que os seus objectos sejam definidos com procedimentos com um número não fixo de parâmetros. A *classe-conta-levan-depo* vai herdar os métodos *saldo* e *identifica* de *classe-conta* e definir internamente os métodos *levanta!* e *deposita!*.

```
(define classe-conta-levan-depo
  (lambda (ident saldo)

    (let ((conta (classe-conta ident saldo)))
      (lambda msg
        (let ((m (car msg)))
          (cond
            ((equal? m 'levanta!)
             (conta 'escreve!
                    (- (conta 'saldo) (cadr msg)))
             (conta 'saldo))
            ((equal? m 'deposita!)
             (conta 'escreve!
                    (+ (conta 'saldo) (cadr msg)))
             (conta 'saldo))
            (else (apply conta msg))))))))
```




```
↦(define outra-conta
    (classe-conta-levan-depo 12345 2000))
outra-conta
↦(outra-conta 'levanta! 100)
1900
↦(outra-conta 'deposita! 200)
2100
↦(outra-conta 'identifica)
12345
↦(outra-conta 'saldo)
2100
```



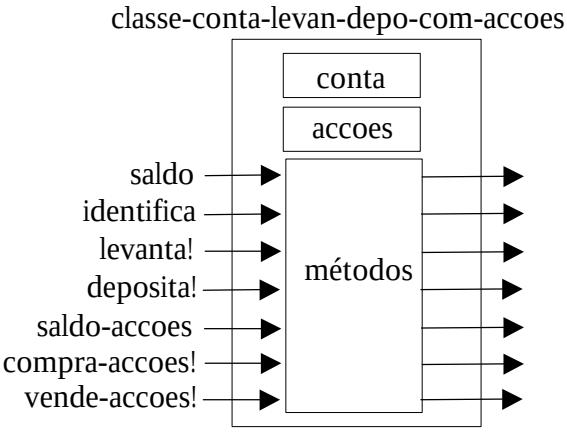
A partir de *classe-conta-levan-depo* vai ser definida a *classe-conta-levan-depo-com-accoes*, a qual vai também gerir uma carteira de acções. Herda os métodos *saldo*, *levanta!* e *deposita!*, altera o método *identifica* para indicar não só o código da conta, mas também o nome do titular. Os novos métodos são:

compra-accoes! -retira ao saldo da conta a quantia necessária e coloca-a no saldo das acções;

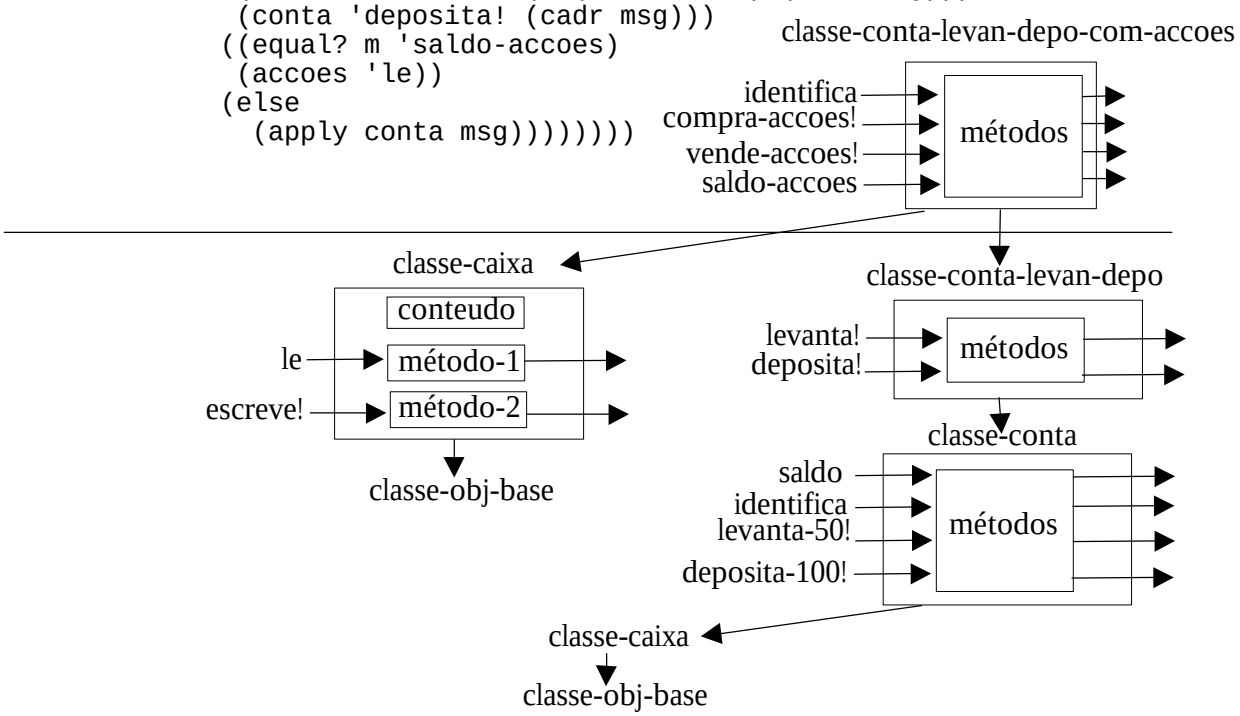
[4] Apesar de acessível, não se recomenda a utilização deste método, por não fazer parte dos métodos de *classe-conta*

vende-accoes! - retira ao saldo das acções a quantia necessária e coloca-a no saldo da conta;
saldo-accoes - devolve o valor das acções.

```
↦(define conta-accoes
    (classe-conta-levan-depo-com-accoes 1234567 'ana 1200 400))
conta-accoes

↦(conta-accoes 'identifica)
1234567: ana

↦(conta-accoes 'compra-accoes! 200)
1000          ; no final da compra ou venda de acções, visualiza o saldo da conta

↦(conta-accoes 'saldo-accoes)
600

↦(conta-accoes 'vende-accoes! 500)
1500

↦(conta-accoes 'saldo-accoes)
100
```



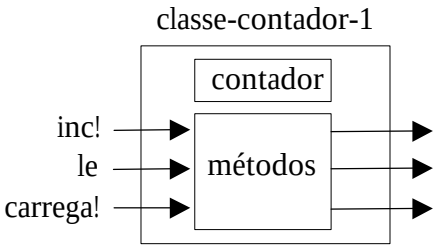

```
(define classe-conta-levan-depo-com-accoes
  (lambda (codigo nome saldo val-accoes)

    (let ((accoes (classe-caixa val-accoes))
          (conta (classe-conta-levan-depo codigo saldo)))
      (lambda msg
        (let ((m (car msg)))
          (cond
            ((equal? m 'identifica)
             (display (conta 'identifica))
             (display ": ")
             (display nome))
            ((equal? m 'compra-accoes!)
             (accoes 'escreve! (+ (accoes 'le) (cadr msg)))
             (conta 'levanta! (cadr msg)))
            ((equal? m 'vende-accoes!)
             (accoes 'escreve! (- (accoes 'le) (cadr msg)))
             (conta 'deposita! (cadr msg)))
            ((equal? m 'saldo-accoes)
             (accoes 'le))
            (else
              (apply conta msg))))))))
```



A definição de *classe-conta-levan-depo-com-accoes* requere algumas explicações adicionais. Em primeiro lugar, salienta-se o recurso directo a *classe-caixa*, para definir o objecto local *accoes*, pois, para além do registo do saldo da conta, é ainda necessário manter o estado do valor da carteira de acções. O método correspondente a *identifica* é redifinido, apresentando a curiosidade de recorrer ao anterior método com o mesmo nome, para ir procurar o código da conta. É assim possível concluir que, ao criar um objecto do tipo *classe-conta-levan-depo-com-accoes*, com os parâmetros *codigo*, *nome*, *saldo* e *valor-accoes*, tudo acontece como se o referido objecto tivesse duas variáveis locais referentes ao saldo da conta e ao valor das acções e respondesse às mensagens *saldo*, *levanta!*, *deposita!*, *identifica*, *compra-accoes!*, *vende-accoes!*, e *saldo-accoes*.
Este exemplo demonstra bem o reaproveitamento de código que a *programação orientada por objectos* permite e a consequente popularidade que este paradigma de programação goza.

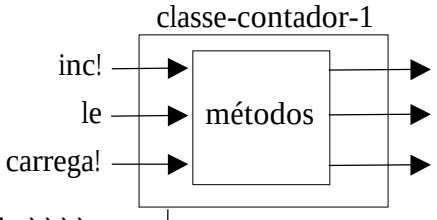
Exemplo 6.1

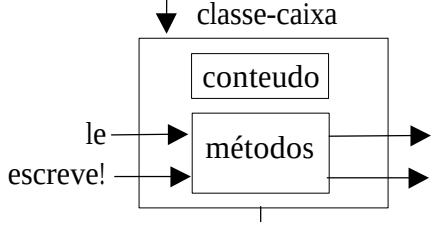
A classe de objectos designada por *classe-contador-1* cria objectos que reagem às mensagens *inc!* (incrementa o conteúdo do contador de uma unidade, o qual é, seguidamente, devolvido), *le* (devolve o conteúdo do contador) e *carrega!* (carrega o contador com um valor fornecido e, seguidamente, devolve o respectivo conteúdo). Quando são criados, os objectos deste tipo apresentam conteúdo zero.




```
↦(define conta1 (classe-contador-1))
conta1
↦(conta1 'inc!)
1
↦(conta1 'inc!)
2
↦(conta1 'carrega! 56)
56
↦(conta1 'inc!)
57
↦(conta1 'l3)
l3: mensagem desconhecida!...
↦(conta1 'le)
57
```

```
(define classe-contador-1
  (lambda ()

    (let ((contador (classe-caixa 0)))
      (lambda mensagem
        (let ((m (car mensagem)))
          (cond ((equal? m 'inc!)
                 (contador 'escreve!
                           (add1 (contador 'le))))
                ((equal? m 'le) (contador 'le))
                ((equal? m 'carrega!)
                 (contador 'escreve! (cadr mensagem)))
                (else (apply contador mensagem)))))))))
```

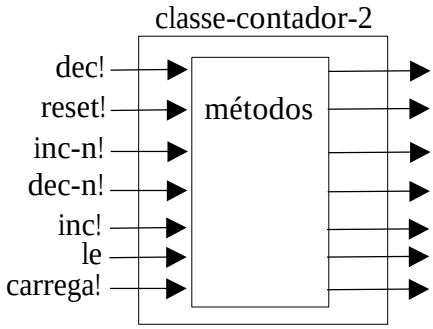




Exercício 6.1

Escrever em *Scheme* a *classe-contador-2* que herda os métodos de *classe-contador-1* e ainda reage às mensagens *dec!* (decrementa o conteúdo do contador de uma unidade, o qual é, seguidamente, devolvido), *reset!* (carrega o contador com zero e, seguidamente, devolve o respectivo conteúdo), *inc-n!* (incrementa o conteúdo do contador de *n* unidades, o qual é, seguidamente, devolvido) e *dec-n!* (decrementa o conteúdo do contador de *n* unidades, o qual é, seguidamente, devolvido).

```
↦(define c2 (classe-contador-2))
c2
↦(c2 'inc!)
1
↦(c2 'inc!)
2
↦(c2 'carrega! 56)
56
↦(c2 'inc!)
57
↦(c2 'dec!)
56
↦(c2 'dec-n! 5)
51
↦(c2 'dec-n! 30)
21
↦(c2 'dec-10)
dec-10: mensagem desconhecida!...
↦(c2 'reset!)
0
↦(c2 'dec!)
-1
```



4- Objectos com história; classe-coleccao

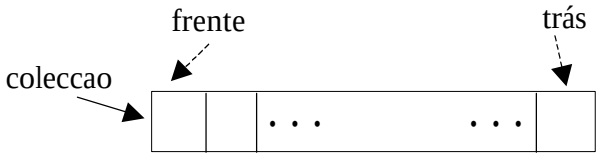
A história dos objectos nem sempre se representa por dados simples, como acontecia nos exemplos das contas bancárias. Para as contas bancárias a *classe-caixa* chegava perfeitamente para guardar o saldo.

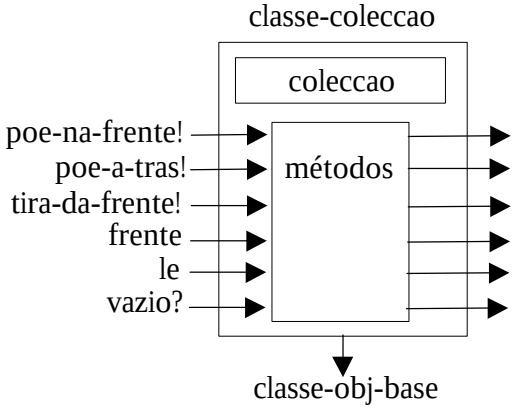
Introduz-se agora a *classe-coleccao*, com a qual se criam objectos associados a dados compostos. Um objecto da *classe-coleccao* pode crescer juntando-lhe novos elementos, na frente ou na parte de trás, ou diminuir retirando-lhe o elemento da frente. Os métodos, a considerar, são os seguintes:

poe-na-frente! - coloca um novo elemento na frente;

poe-a-tras! - coloca um novo elemento na parte de trás;

tira-da-frente! - retira o elemento da frente;





frente - devolve o valor do elemento da frente;
le - devolve uma lista com o conteúdo da colecção;
vazio? - devolve *#t*, se colecção vazia, caso contrário, devolve *#f*

Podemos assim imaginar uma sessão com objectos criados por instanciação de *classe-coleccao*.

```
↦(define col (classe-coleccao))
col
↦(col 'poe-na-frente! 6)
ok
↦(col 'poe-a-tras! 8)
ok
↦(col 'poe-na-frente! 1)
ok
↦(col 'le)
(1 6 8)
↦(col 'poe-a-tras! 12)
ok
↦(col 'frente)
1
↦(col 'le)
(1 6 8 12)
↦(col 'tira-frente)
tira-frente: mensagem desconhecida!...
↦(col 'tira-da-frente!)
ok
↦(col 'le)
(6 8 12)
↦(col 'tira-da-frente!)
ok
↦(col 'tira-da-frente!)
ok
↦(col 'le)
(12)
↦(col 'tira-da-frente!)
ok
↦(col 'frente)
sem elementos!...
↦(col 'tira-da-frente!)
sem elementos!...
```

A implementação que se segue, recorre a uma lista mutável, com a cabeça *'coleccao:*. Fica assim garantido que, mesmo na ausência de qualquer elemento, ou seja, quando a colecção está vazia, que a sua representação é ainda uma lista não vazia, pois terá sempre o elemento cabeça, acima indicado.

```
(define classe-coleccao
  (lambda ()

    (let ((coleccao (list 'coleccao))
          (obj-base (classe-obj-base)))
      (lambda mensagem
```

```
(let ((m (car mensagem)))
  (cond
    ((equal? m 'le) (cdr coleccao))
    ((equal? m 'vazio?) (null? (cdr coleccao)))
    ((equal? m 'poe-na-frente!)
     (set-cdr! coleccao
               (cons (cadr mensagem)
                     (cdr coleccao)))
     'ok)
    ((equal? m 'poe-a-tras!)
     (append! coleccao
              (cons (cadr mensagem) '()))
     'ok)
    ((equal? m 'frente)
     (if (null? (cdr coleccao))
         (display "sem elementos!...")
         (cadr coleccao)))
    ((equal? m 'tira-da-frente!)
     (if (null? (cdr coleccao))
         (display "sem elementos!...")
         (begin
           (set-cdr! coleccao (cddr coleccao))
           'ok)))
    (else
      (apply obj-base mensagem)))))))
```

Exemplo 6.2

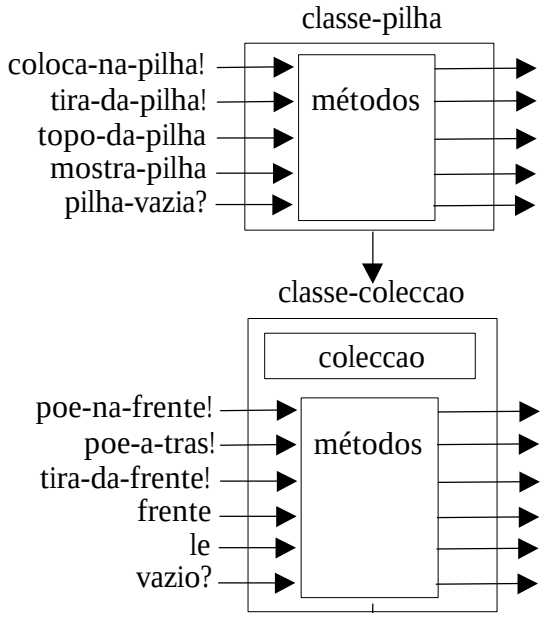
Tomando como base *classe-coleccao*, definir *classe-pilha*. Um objecto criado por instanciação desta classe, designado por *pilha*, é composto por uma colecção ordenada de elementos, em que um novo elemento é colocado antes do elemento mais recente da pilha, o *topo da pilha*, passando ele a constituir o *topo* dessa pilha. O primeiro elemento a sair é o último elemento colocado na pilha, ou seja, o *topo* da pilha, razão que justifica a designação *LIFO*, *last-in-first-out*, para esta estrutura de dados. Frequentemente, faz-se a analogia da *pilha informática* com a *pilha de pratos* que vai crescendo em cima de uma mesa. Também nesta, o elemento mais facilmente acessível é o que se encontra no topo da pilha dos pratos, ou seja, o último que lá foi colocado.

Os métodos normalmente associados à *pilha* são os seguintes:

poe-na-pilha! [5] - coloca um novo elemento no topo da pilha;
tira-da-pilha! [6] - tira o elemento do topo da pilha;
pilha-vazia? - devolve #t, se pilha vazia, se não devolve #f.
topo-da-pilha - devolve o elemento do topo da pilha, sem a alterar;
mostra-pilha - visualiza o conteúdo da pilha.

Para clarificar o funcionamento de *pilha*, segue-se um exemplo de utilização deste tipo de objecto.

```
↦(define pilha (classe-pilha))
pilha
↦(pilha 'mostra-pilha)
topo:
ok
↦(pilha 'poe-na-pilha! 54)
ok
↦(pilha 'poe-na-pilha! 35)
ok
↦(pilha 'mostra-pilha)
```



[5] *push!*
[6] *pop!*

```
topo:  35  54
ok
↦(pilha 'topo-da-pilha)
35
↦(pilha 'mostra-pilha)
topo:  35  54
ok
↦(pilha 'tira-da-pilha!)
ok
↦(pilha 'mostra-pilha)
topo:  54
ok
↦(pilha 'pilha-vazia?)
#f
↦(pilha 'tira-da-pilha!)
ok
↦(pilha 'tira-da-pilha!)
sem elementos!...
↦(pilha 'mostra-pilha)
topo:
ok
```

A definição de *classe-pilha*, recorrendo a *classe-coleccao*, torna-se relativamente simples. O método *poe-na-pilha!* recorre a *poe-na-frente!*, *tira-da-pilha!* a *tira-da-frente!*, *topo-da-pilha* a *frente*, *mostra-pilha*, recorre a *le*, para obter o conteúdo da pilha, e depois passa à visualização, e, finalmente, *pilha-vazia?* que recorre a *vazio?*.

```
(define classe-pilha
  (lambda ()

    (let ((coleccao (classe-coleccao)))
      (lambda mensagem
        (let ((m (car mensagem)))
          (cond
            ((equal? m 'poe-na-pilha!)
             (coleccao 'poe-na-frente! (cadr mensagem)))
            ((equal? m 'tira-da-pilha!)
             (coleccao 'tira-da-frente!))
            ((equal? m 'topo-da-pilha)
             (coleccao 'frente))
            ((equal? m 'pilha-vazia?)
             (coleccao 'vazio?))
            ((equal? m 'mostra-pilha)
             (display "topo:  ")
             (for-each (lambda (x)
                         (display x)
                         (display "  "))
                       (coleccao 'le))
             (newline)
             'ok)
            (else
              (apply coleccao mensagem))))))))
```

Exercício 6.2

Recorrendo à *classe-pilha* apresentada no exemplo anterior, escrever a *classe-pilha-com-comprimento* que, para além dos métodos daquela classe, tem um método adicional, designado por *comprimento*, que devolve o número de elementos contidos na pilha.

Exercícios e exemplos de final de capítulo

Nesta Secção, entre os exercícios que se apresentam para consolidação da matéria dada, alguns são aproveitados para introduzir novas classes. Entre estas salientam-se *classe-fila*, *classe-coleccao-melhorada*, e *classe-lista-circular*.

Exercício 6.3

A *classe-acumulador-com-funcao-fixa* cria objectos que são inicializados com zero e aos quais é associada uma função de dois operandos. Estes objectos reagem a três mensagens: *funcao!*, *carrega!*, e *le*. A mensagem *funçao!* é acompanhada de um valor *v* e, perante ela, o objecto calcula a função associada para *v* e para o conteúdo do acumulador, e o respectivo resultado passa a constituir o novo conteúdo do acumulador. A mensagem *carrega!*, que também é acompanhada de um valor *v*, actualiza o conteúdo do acumulador com *v*. A mensagem *le* devolve o conteúdo do acumulador.

```
↦(define acumula-* (classe-acumulador-com-funcao-fixa *))
acumula-*
↦(acumula-* 'le)
0
↦(acumula-* 'carrega! 5)
ok
↦(acumula-* 'funcao! 6)
ok
↦(acumula-* 'le)
30
```

Escrever em *Scheme classe-acumulador-com-funcao-fixa*.

Exercício 6.4

A *classe-acumulador-com-funcao-geral* cria objectos que são inicializados com zero. Estes objectos reagem a três mensagens: *funcao!*, *carrega!*, e *le*. A mensagem *funçao!* é acompanhada de uma função *func* de dois operandos e de um valor *v* e, perante ela, o objecto calcula a *func* para *v* e para o conteúdo do acumulador, e o respectivo resultado passa a constituir o novo conteúdo do acumulador. A mensagem *carrega!*, que também é acompanhada de um valor *v*, actualiza o conteúdo do acumulador com *v*. A mensagem *le* devolve o conteúdo do acumulador.

```
↦(define acumula-geral (classe-acumulador-com-funcao-geral))
acumula-geral
↦(acumula-geral 'le)
0
↦(acumula-geral 'carrega! 5)
ok
↦(acumula-geral 'funcao! * 6)
ok
↦(acumula-geral 'le)
30
```

```
↦(acumula-geral 'funcao! max 36)
ok
↦(acumula-geral 'le)
36
```

Escrever em *Scheme classe-acumulador-com-funcao-geral*.

Exercício 6.5

A *classe-contador-de-n-a-m* cria objectos contadores cuja gama de contagem vai de *n* a *m* (em que *n* < *m*). Os valores que definem esta gama constituem os argumentos a fornecer nas chamadas desta classe. Os objectos, inicializados com *n*, reagem a quatro mensagens: *inc!* (incrementa o contador de uma unidade e devolve o conteúdo; nesta operação se atinge *m+1*, passa automaticamente para *n*), *dec!* (decrementa o contador de uma unidade e devolve o conteúdo; nesta operação se atinge *n-1*, passa automaticamente para *m*), *carrega!* (carrega o contador com um valor fornecido e devolve o conteúdo; se aquele valor se situar fora da gama de contagem, o contador é carregado com *n*), e *le* (devolve o conteúdo do contador).

```
↦(define conta-entre-3-e-15 (classe-contador-de-n-a-m 3 15))
conta-entre-3-e-15
↦( conta-entre-3-e-15 'inc)
4
↦( conta-entre-3-e-15 'dec)
3
↦( conta-entre-3-e-15 'dec)
15
↦( conta-entre-3-e-15 'dec)
14
↦( conta-entre-3-e-15 'inc)
15
↦( conta-entre-3-e-15 'inc)
3
↦( conta-entre-3-e-15 'carrega 23)
3
```

Escrever em *Scheme classe-contador-de-n-a-m*.

Exercício 6.6

Recorrenco a *classe-coleccao*, apresentada no decorrer do presente capítulo, escrever *classe-fila*.
Um objecto criado por instanciação desta classe é composto por uma colecção ordenada de elementos, em que um novo elemento é colocado a seguir ao último elemento, passando o novo elemento a ser o último. O primeiro elemento a sair é o primeiro elemento da fila, razão que justifica a designação *FIFO, first-in-first-out*, normalmente associada a esta estrutura de dados. Os métodos desta classe *fila* são:

poe-na-fila![7] - coloca um novo elemento no fim da fila;
tira-da-fila![8] - tira o primeiro elemento da fila;
fila-vazia? - devolve *#t*, se fila vazia, se não devolve *#f*.
frente-da-fila - devolve o valor do primeiro elemento da fila;
mostra-fila - visualiza o conteúdo da fila.

Segue-se um exemplo de utilização deste tipo de objecto.

---

[7] *enqueue!*
[8] *dequeue!*

```
↦(define cantina (classe-fila))
cantina
↦(cantina 'mostra-fila)
frente:
ok
↦(cantina 'poe-na-fila! 'jose)
ok
↦(cantina 'poe-na-fila! 'maria)
ok
↦(cantina 'poe-na-fila! 'manuel)
ok
↦(cantina 'mostra-fila)
frente:  jose  maria  manuel
ok
↦(cantina 'frente-da-fila)
jose
↦(cantina 'mostra-fila)
frente:  jose  maria  manuel
↦(cantina 'tira-da-fila!)
ok
↦(cantina 'mostra-fila)
frente:  maria  manuel
ok
```

Exercício 6.7

No presente capítulo, a implementação de *classe-coleccao* desenvolvida recorreu a uma lista mutável, motivando ordens de crescimento *O(n)* na operação de inserção de elementos no fim da lista[9]. Esta operação não foi utilizada na definição de *classe-pilha*, mas, provavelmente, o mesmo não se dirá da *classe-fila* (ver exercício anterior). Recordemos que na *pilha* as operações de entrada e saída de elementos ocorrem sempre na parte da frente da lista mutável.

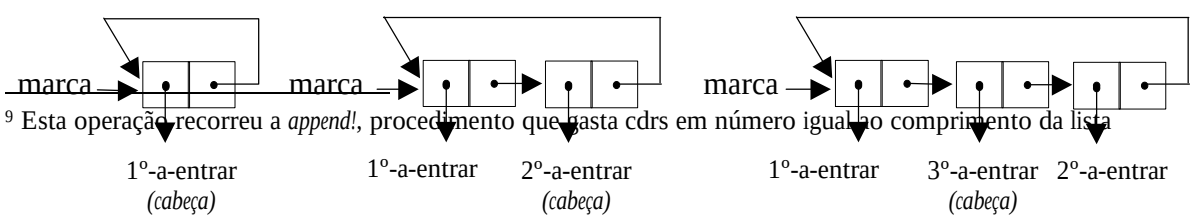
Para permitir a inserção de elementos no fim da lista como uma operação *O(1)*, escrever em *Schem* a *classe-coleccao-melhorada*, que baseia a sua implementação num par. Neste, o elemento da esquerda aponta para a frente da lista mutável e a parte direita aponta para o fim da mesma lista.

Posteriormente, identificar as alterações a introduzir nas definições de *classe-pilha* e *classe-fila* (exercícios anteriores), motivadas pela utilização de *classe-coleccao-melhorada*.

Exercício 6.8

Uma outra abstracção interessante é a chamada *lista-circular*. Trata-se de uma estrutura muito flexível e que se adapta bem à implementação de FIFOs e LIFOs, como se tentará mostrar.

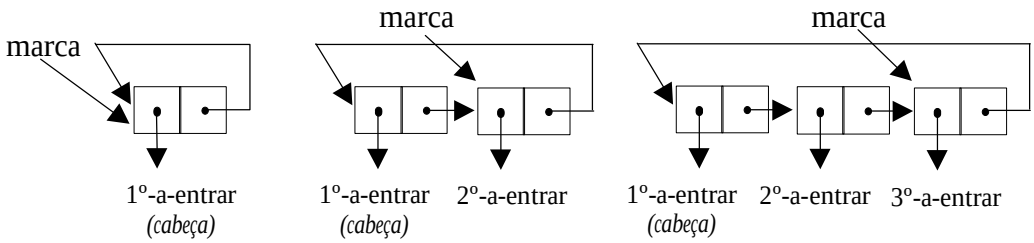
Na *lista-circular*, o *cdr* do último par, em vez de apontar para uma lista vazia, como acontece nas listas normais, aponta para o primeiro par da lista. Nesta estrutura, um novo elemento é sempre colocado a seguir ao elemento referido por um apontador que, nas figuras, é designado por *marca*. A entrada de um novo elemento é realizada com `(set-cdr! marca (cons novo (cdr marca)))`. A *cabeça* da *lista-circular* é o elemento acessível através de `(cadr marca)`, ou seja, o elemento a seguir à *marca*. A saída da *cabeça* acontece fazendo `(set-cdr! marca (cddr marca))`



[9] Esta operação recorreu a *append!*, procedimento que gasta cdrs em número igual ao comprimento da lista

Na primeira série de três figuras, a *marca* é mantida fixa, apontando sempre o elemento que entrou em primeiro lugar. Como o último elemento que entrou é sempre colocado a seguir ao apontado por *marca*, ele será a *cabeça* da *lista-circular*. Todas as operações necessárias à implementação de estruturas LIFO, *pilhas*, encontram-se em *lista-circular*, uma vez que o último elemento a entrar, o *topo* da *pilha*, é facilmente lido ou retirado.

Na segunda série de três figuras, depois de se fazer entrar um novo elemento, a *marca* é movida para a frente, passando a apontar o elemento que acabou de entrar, fazendo `(set! marca (cdr marca))`. Verifica-se assim, analisando a série de figuras, que a *cabeça* da *lista-circular* será sempre o elemento que nela entrou há mais tempo. Notar também que, ao retirar a *cabeça* da *lista-circular*, esta designação passa automaticamente para o elemento que nela entrou há mais tempo. Todas as operações necessárias à implementação de estruturas FIFO, *filas-de-espera*, encontram-se em *lista-circular*, uma vez que o seu elemento mais antigo, a *frente* da *fila-de-espera*, é facilmente lido ou retirado.



1- Definir a *classe-lista-circular* que cria objectos que respondem às seguintes mensagens:
lista-circular-vazia? - devolve *#t* se a lista é vazia, ou *#f* no caso contrário;
entra-na-lista-circular! - insere na lista um novo elemento que recebe como argumento;
retira-cabeca-da-lista-circular! - retira a *cabeça* da lista;
move-marca! - faz avançar a *marca* da lista;
cabeça-da-lista-circular - devolve a *cabeça* da lista, que se mantém intacta;
mostra-lista-circular - visualiza os elementos da lista, com dois espaços entre eles.

2- Tomando por base a *classe-lista-circular*, definir a *classe-pilha* que cria objectos do tipo *pilha* e que respondem às seguintes mensagens:
poe-na-pilha! - coloca um novo elemento no topo da pilha;
tira-da-pilha! - tira o elemento do topo da pilha;
pilha-vazia? - devolve *#t*, se pilha vazia, se não devolve *#f*;
topo-da-pilha - devolve o elemento do topo da pilha, sem a alterar;
mostra-pilha - visualiza o conteúdo da pilha.

3- Tomando por base a *classe-lista-circular*, definir a *classe-fila-de-espera* que cria objectos do tipo *fila-de-espera* e que respondem às seguintes mensagens:
poe-na-fila-de-espera! - coloca um novo elemento no fim da fila;
tira-da-fila-de-espera! - tira o primeiro elemento da fila;
fila-vazia? - devolve *#t*, se fila vazia, se não devolve *#;*
frente-da-fila - devolve o primeiro elemento da fila;
mostra-fila - visualiza o conteúdo da fila.

6.1- Projecto- Jogo de dados
Definir a *classe-dados* que cria objectos que sabem jogar dados e que respondem às seguintes mensagens:

aposta! - simula lançamento de 3 dados e, conforme as faces voltadas para cima *(ver regras mais à frente)*, verifica se o jogador ganhou ou perdeu e o respectivo montante. Posteriormente actualiza o saldo do jogo;
saldo - visualiza o saldo actual *(positivo ou negativo)*;
termina - termina o jogo, com a visualização do saldo.

Regras para o cálculo do montante que se ganha ou perde em cada jogada:
3 faces iguais: ganha 72 x valor da aposta;
2 faces iguais: ganha 36 x valor da aposta;
3 faces diferentes
e soma-das-3-faces menor ou igual a 11: ganha soma-das-3-faces x valor-da-aposta;
3 faces diferentes
e soma-das-3-faces maior que 11: perde soma-das-3-faces x valor-da-aposta.

```
↦(define jogo-1 (classe-dados))      ; cria um jogo designado por jogo-1
jogo-1
↦(jogo-1 'saldo)                     ; verifica o saldo de jogo-1
saldo: 0
↦(jogo-1 'aposta! 20)                ; faz uma aposta de 20 com jogo-1
dado 1: 4
dado 2: 3
dado 3: 4
ganhaste: 36 x 20 = 720              ; sorte...

↦(jogo-1 'aposta! 100)                ; faz uma aposta de 100 com jogo-1
dado 1: 6
dado 2: 3
dado 3: 4
perdeste: 13 x 100 = 1300             ; grande azar...

↦(jogo-1 'saldo)
saldo: -580

↦(define jogo-2 (classe-dados))  ; cria um novo jogo designado por jogo-2
jogo-2
↦(jogo-2 'aposta! 50)            ; jogando com jogo-2 ...
...

                                       ; mas o jogo-1 continua preparado
↦(jogo-1 'aposta! 200)                 ; para actuar
dado 1: 1
dado 2: 3
dado 3: 4
ganhaste: 8 x 200 = 1600

↦(jogo-1 'saldo)
saldo: 1020

↦(jogo-1 'termina)
O jogo vai terminar
e o saldo e': 1020
```

6.2- **Projecto**- Vidas

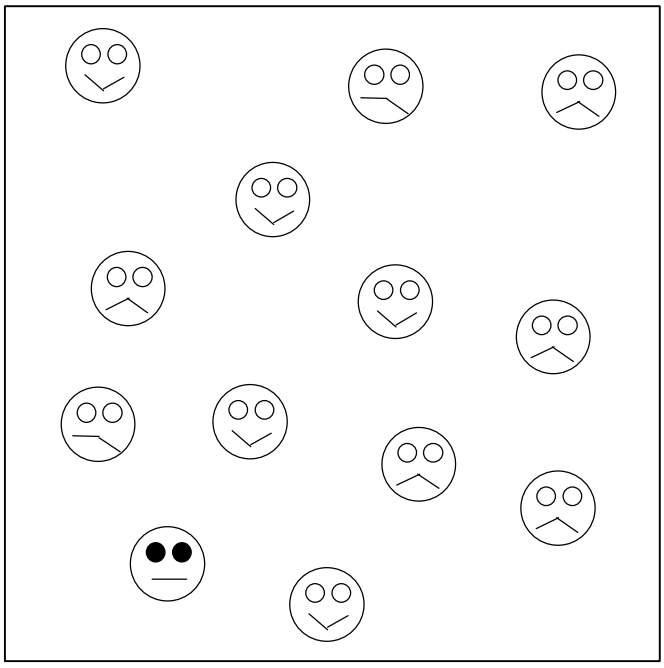
Uma criatura simples, que dá pelo nome de *vidas*, tem alguns problemas de visão. Por isso, desde que nasce e até que morre, o *vidas* não faz outra coisa que não seja esbarrar-se com outras criaturas, algumas simpáticas que até lhe oferecem prendas (20 pontos), mas também outras, pelo contrário, que são antipáticas e que acabam por lhe retirar alguma coisa (10 pontos), e outras que são neutras, que lhe retiram 50% do que tenha amealhado até essa altura (ou seja, se o *vidas* tiver amealhado, por exemplo, 30 passa a ter 15, mas se tiver -30 passa para -15).


criatura simpática


criatura antipática

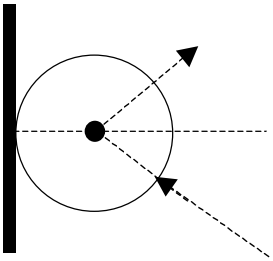

criatura neutra

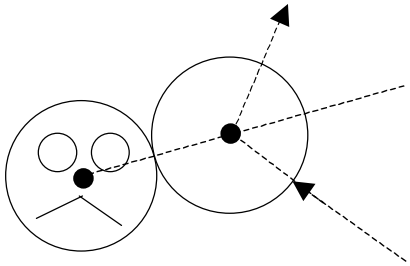

o vidas



Todas as criaturas referidas, incluindo o *vidas*, são colocadas aleatoriamente num tabuleiro. Mas, contrariamente ao *vidas* que não pára, apesar da sua deficiente visão, as outras criaturas depois de colocadas no tabuleiro permanecem fixas. Limitam-se a que o *vidas*, nos seus passeios, se encontre com elas, para o presentear ou para lhe retirar alguma coisa. Na figura, o tabuleiro mostra 5 criaturas simpáticas, 5 antipáticas, 2 neutras e o *vidas*.

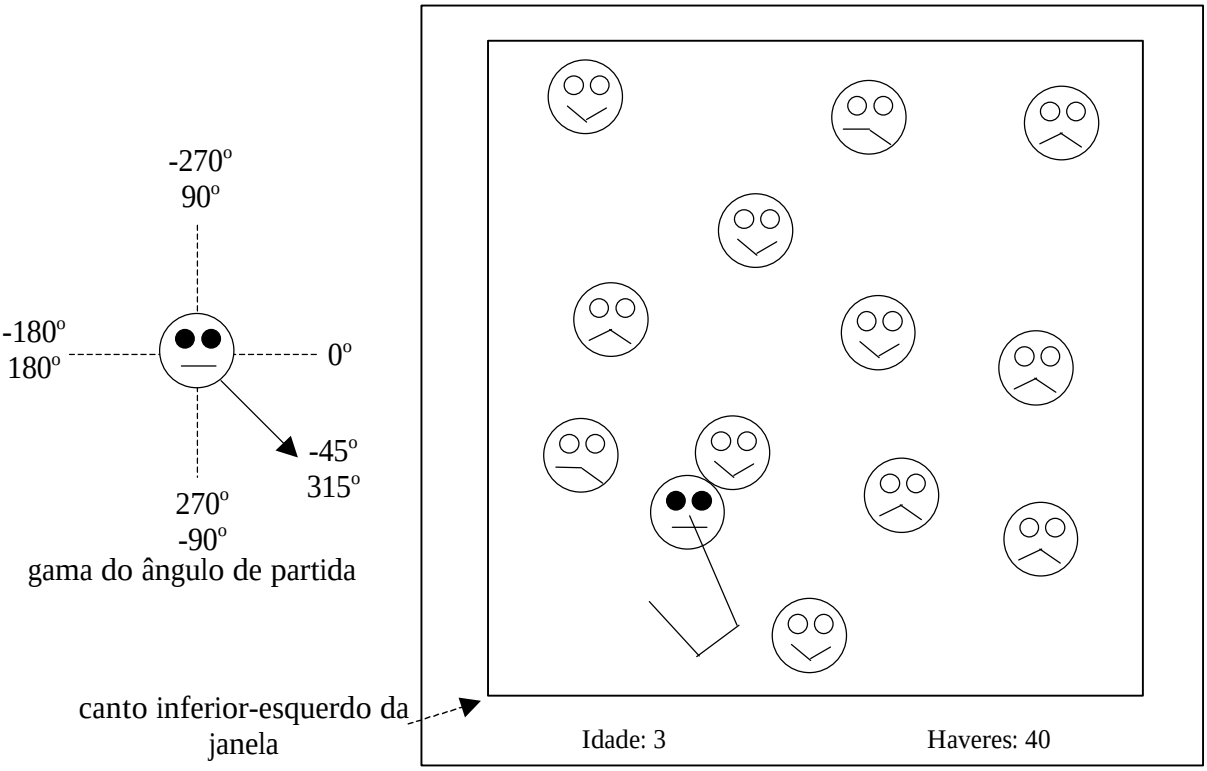
Como se desloca o *vidas*?
Quando surge o tabuleiro com todos os actores, é seleccionado um ângulo de partida para o início da trajectória do *vidas*. Depois, inicia-se o percurso sobre o tabuleiro, sempre em linha recta, até encontrar ou uma das 4 paredes do tabuleiro ou uma das outras criaturas. Nessa altura, a direcção da deslocação sofre uma reflexão.


Reflexão numa parede


Reflexão numa criatura

Quando termina o percurso do *vidas*?
O tempo de vida do *vidas* é parametrizado, bem como os pequenos passos que dá. Sempre que esbarra noutra criatura, mas não numa parede, conta como dia de aniversário, e a idade avança de 1 ano. É por isso que algumas criaturas o presenteiam, se bem que outras, como já se viu, não se importam muito com esse dia festivo, antes pelo contrário. Quando atinge o tempo

limite de vida, o percurso do *vidas* termina e é altura de avaliar os haveres que conseguiu amealhar.
Com base na *programação OO*, pretende-se desenvolver um programa para simular o percurso de criaturas do tipo *vidas*. Sempre que o nosso amigo esbarra, é dia de comemorar o aniversário, por isso são actualizados a sua idade (+ 1 ano) e os seus haveres (de acordo com a criatura em que esbarrou), e porque certamente vai beber um pouco mais de champanhe na comemoração, perde o rumo e muda de direcção. Apesar de não contar como dia de aniversário, também muda de direcção quando esbarra numa parede. Numa janela gráfica são visualizados o tabuleiro com todos os actores, incluindo o *vidas*, o percurso que vai fazendo, a sua idade e haveres. A figura dá uma ideia do que será a visualização ao fim de 2 aniversários, quando, para a situação inicial descrita na primeira figura, se forneceu um ângulo de partida de -45º (equivalente a 315º).



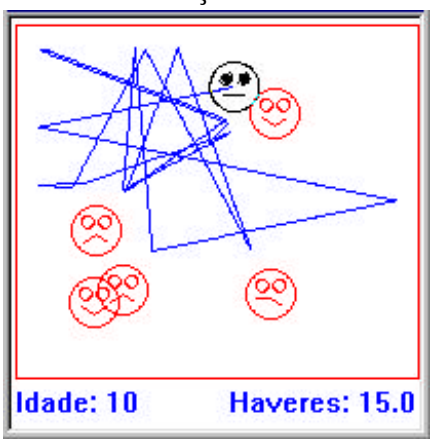
Estruturar e escrever em *Scheme* o programa *vidas* que tem como principal objectivo dar a conhecer qual o montante em haveres com que o *vidas* termina o seu percurso. Este programa aceita como argumentos tempo de vida do actor principal e a dimensão do seu pequeno passo, normalmente o valor 1. No início do programa, é pedido a origem do canto inferior-esquerdo do tabuleiro, a sua largura e altura, o raio da cara dos actores, e o número de simpáticos, antipáticos e neutros.

```
↦(janela 200 200 "")
#[graphics window 7344376]

↦(vidas 10 1)   ; lançamento do programa vidas com os argumentos 10, tempo de vida
caracteristicas tabuleiro                                    ; e 1, passo
origem-x origem-y larg alt raio n-simp n-ant n-neutros:
2                ; coordenada x da origem do tabuleiro
25               ; coordenada x da origem do tabuleiro
196              ; largura do tabuleiro
172              ; altura do tabuleiro
12               ; raio da cara dos actores
2                ; número de simpáticos
2                ; número de antipáticos
1                ; número de neutros
```

```
Angulo de partida:
45
...
Fim... do Vidas...
```

Situação inicial

Situação final



Pistas:

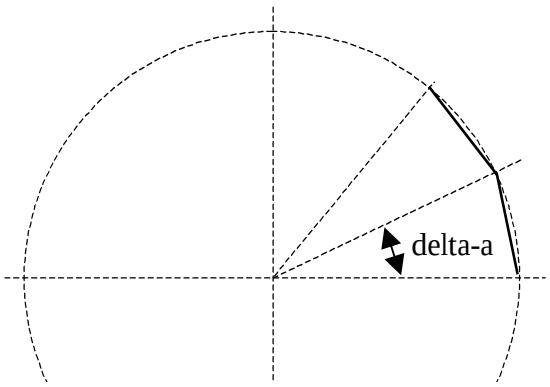
1- Analisar cuidadosamente o problema exposto e identificar as principais classes a considerar, incluindo os seus *métodos*[10]. Analisar as hipóteses da *classe-vidas* (para criar objectos do tipo *vidas)* e *classe-tabuleiro* (para criar objectos do tipo tabuleiro com os respectivos actores, mas excluindo o *vidas*). Que métodos devem ter?

2- Para não se atacar inicialmente o problema na sua globalidade, pois já se apresenta com uma razoável complexidade, considerar as seguintes fases:

1ª Fase

Tomando por base as primitivas gráficas apresentadas no *Anexo B*, escrever em *Scheme* os procedimentos que desenham as caras dos vários actores deste jogo, ou seja, *o-vidas*, *o-simpatico*, *o-antipatico* e *o-neutro*. Uma vez que o elemento gráfico mais usado é a circunferência, apresenta-se um procedimento para esta tarefa, com os parâmetros *centro*, *raio* e *pintar*. Os dois primeiros parâmetros tem um significado óbvio e o terceiro é um booleano que sendo *#f* indica desenhar a circunferência e sendo *#t* indica pintar a área delimitada pela circunferência.

A ideia que se explorou na concepção deste procedimento residiu em aproximar a circunferência a uma linha poligonal fechada, com um número suficiente de polígonos, para garantir um resultado visual de boa qualidade. É por esta razão que o ângulo *delta-a* é função do raio (quanto maior for o raio menor deverá ser *delta-a*, para se obter uma poligonal com um maior número de segmentos). Outra ideia que se explorou foi determinar as extremidades dos segmentos do 1º quadrante em coordenadas relativas ao centro



[10] Procedendo desta forma, estamos mais uma vez perante uma abordagem do tipo *de-cima-para-baixo*, já que, ao identificar as *classes principais*, também se procura dividir o problema em problemas mais simples. Por outro lado, quando a definição de uma classe se baseia noutra classe, ou seja, quando uma classe herda os métodos de outra e se lhe junta novos métodos, a abordagem passa a ser do tipo *de-baixo-para-cima*...
É claro que tudo isto, se a complexidade do problema ainda o justificar, não substitui a abordagem *de-cima-para-baixo* para identificar as tarefas e sub-tarefas principais do problema. E, no contexto destas tarefas e sub-tarefas, as classes e os respectivos objectos seriam utilizados, surgindo como a *abstracção de dados* do problema...

(procedimento local *circ1/4*), e a partir destas determinar as coordenadas de todos os quadrantes.

```
(define circ                    ; desenho de circunferência ou de círculo
  (lambda (centro raio pintar)        ; pintar = #f circunferência, =#t círculo
    (let ((pi1/2 (/ pi 2))
          (delta-a (/ pi (* 4 raio)))  ; passo do ângulo, em função do raio
          (x-centro (car centro))
          (y-centro (cadr centro)))
      (letrec ((xform               ; com os pontos que definem o 1º quadrante da
                 (lambda (lis kx ky)  ; circunferência, em relação ao centro,
                                      ; encontrar os pontos, em coordenadas
                                      ; absolutas, de qualquer quadrante
                                            ; Código quadrante
                                            ; 1º kx= 1 ky= 1
                                            ; 2º kx=-1 ky= 1
                                            ; 3º kx=-1 ky=-1
                   (if (null? lis)          ; 4º kx= 1 ky=-1
                       '()
                       (cons (list (+ x-centro (* kx (caar lis)))
                                   (+ y-centro (* ky (cadar lis))))
                             (xform (cdr lis) kx ky)))))

               (cir1/4          ; devolve uma lista com os pontos do 1º quadrante
                 (lambda (a)     ; de uma circunferência, em relação ao centro
                   (if (> a pi1/2)
                       '()
                       (begin
                         (cons (list
                                 (* raio (cos a))
                                 (* raio (sin a)))
                               (cir1/4 (+ a delta-a))))))))
        (let ((cir90 (cir1/4 0)))
          (let ((cir (append (xform cir90 1 1)                ; 1º quadrante
                             (xform (reverse cir90) -1 1)     ; 2º quadrante
                             (xform cir90 -1 -1)              ; 3º quadrante
                             (xform (reverse cir90) 1 -1)))) ; 4º quadrante
            (move (list (+ raio (car centro))
                        (cadr centro)))
            (if pintar
                (pinta cir)             ; círculo
                (desenha cir)))))))))  ; circunferência
```

2ª Fase

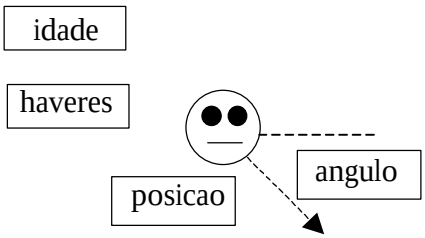
Escrever em *Scheme* a *classe-tabuleiro*, com os parâmetros *origem* (do tabuleiro na janela), *largura* (do tabuleiro), *altura* (do tabuleiro), *raio* (das circunferências que definem as caras), e o *número de simpáticos*, *antipáticos* e *neutros*. Considerar, para já, os métodos que permitem seleccionar as principais características dos objectos gerados a partir desta classe e uma variável interna a considerar nestes objectos, que se pode designar por *tabuleiro* (do tipo *coleccao*), que conterá a definição das paredes do tabuleiro e a posição dos actores (os simpáticos, os antipáticos e os neutros).

```
↦(define tabul-1 (classe-tabuleiro '(10 30) 250 250 12 2 2 1))
tabul-1
↦(tabul-1 'largura)
250
↦(tabul-1 'raio)
12
↦(tabul-1 'altura)
250
```

```
↦(tabul-1 'origem)
(10 30)
↦(tabul-1 'paredes)                          ; os primeiros 4 elementos da colecção
((p . 22) (p . 268) (p . 248) (p . 42))      ; têm a ver com as paredes do tabuleiro
↦(tabul-1 'actores)    ; os restantes elementos da colecção representam os diversos
((s 185 93) (s 131 128) (a 207 214) (a 214 124) (n 141 196))   ; actores no
                                                                ; tabuleiro
```

3ª Fase

Escrever em *Scheme* a *classe-vidas*, com os parâmetros *passo* (define o passo elementar dado pelo *vidas*) e *tabul* (tabuleiro onde o *vidas* criado vai passear), que responde aos métodos que se indicam no diálogo. Nesta altura será importante identificar quais as variáveis internas a considerar nos objectos desta classe, sugerindo-se *idade* (do tipo contador), *haveres* (do tipo caixa), *angulo* (do tipo caixa), *posicao* (do tipo caixa).




```
↦(define vidas-1 (classe-vidas 1 tabul-1))
vidas-1
↦(vidas-1 'idade)
0
↦(vidas-1 'idade+1!)
1
↦(vidas-1 'idade)
1
↦(vidas-1 'poe-angulo! 45)
45
↦(vidas-1 'angulo)
45
↦(vidas-1 'haveres)
0
↦(vidas-1 'poe-haveres 5)
poe-haveres: mensagem desconhecida!...
↦(vidas-1 'poe-haveres! 5)
5
↦(vidas-1 'haveres)
5
↦(vidas-1 'posicao)
(171 93)
↦(vidas-1 'andar!)
90.2509035245341
↦(vidas-1 'posicao)
(171.525321988818 93.8509035245341)
↦(vidas-1 'andar!)
91.1018070490682
↦(vidas-1 'andar!)
91.9527105736024
↦(vidas-1 'posicao)
(172.575965966453 95.5527105736024)
```

4ª Fase

Escrever em *Scheme* a *classe-tabuleiro-2*, com os mesmos parâmetros de *classe-tabuleiro* e que para além dos métodos que herda desta classe, ainda possui *visu* que visualiza o tabuleiro e os vários actores associados, com excepção do *vidas*, e *colisao* com um argumento *vidas* e que devolve *#t* ou *#f*, conforme o objecto *vidas* tenha colidido ou não, naquele momento, com qualquer elemento do tabuleiro. Mais pormenorizadamente, sugere-se que o valor devolvido por este último método, caso não haja colisão, seja a lista (#f), ou a lista (#t 'p ou 'c[11] novos-haveres novo-angulo), se houver colisão.

5ª Fase

Criar objectos com as classes anteriormente desenvolvidas e testar os vários métodos de cada um deles.

É natural que se venha a detectar um problema com os objectos da *classe-tabuleiro-2*. O método *colisão* permite verificar se algum dos elementos de um objecto desta classe se encontra em colisão com *vidas* e, se assim for, procede-se à actualização da idade, dos haveres e do ângulo associados a *vidas*. Ora, devido ao passo que *vidas* dá, poderá acontecer que, ao colidir com um elemento do tabuleiro, *entre por ele a dentro*. O caso não seria grave se, no passo seguinte, já depois das actualizações anteriormente citadas, o *vidas* deixasse de intersectar o elemento com que colidiu. Mas nem sempre assim acontece, e nova colisão seria identificada, ocorrendo novas actualizações da idade, haveres e ângulo. A confusão seria grande, pois tratar-se-ia da mesma colisão processada mais do que uma vez. É por este motivo que se vai juntar uma variável do tipo caixa ao objecto tabuleiro, que se pode designar por *em-interseccao*, colocada a *#t* quando *vidas* colide com um elemento do tabuleiro e colocada a *#f* quando deixa de o intersectar. Sendo assim, mesmo que o método *colisao* devolva (#t 'p ou 'c novos-haveres novo-angulo), a situação do *vidas* não se altera, se *em-interseccao* for *#t*...

Considerando o que acaba de se apresentar, escrever a *classe-tabuleiro-3*, com os mesmos parâmetros de *classe-tabuleiro-2* e que para além dos métodos que herda desta classe, ainda possui *interseccao* que devolve o valor de *em-interseccao*, e *poe-interseccao!* que coloca o argumento *#t* ou *#f* na variável *em-interseccao*.

6ª Fase

Criar objectos com as classes anteriormente desenvolvidas e testar os vários métodos de cada um deles. Introduzir as alterações que venham a ser consideradas como necessárias.

7ª Fase

Desenvolver o programa principal *vidas*, com os parâmetros *tempo-vida* e *passo* que pede as características do tabuleiro, cria um objecto *tabuleiro* e um objecto *vidas*, visualiza o tabuleiro e logo de seguida o *vidas*, visualiza os campos da idade e dos haveres, pede o ângulo de partida (em graus, internamente convertido em radianos) e, finalmente, dá ordem ao *vidas* para fazer o percurso.

8ª Fase

Experimentar o programa *vidas*... e introduzir-lhe os melhoramentos para eliminar eventuais problemas que venham a ser detectados.

---

[11] Para distinguir se a intersecção se deu com uma parede, *'p*, ou com um dos outros actores, *'c*