Capítulo 3

Linguagens Orientadas a Objeto

Orientação a objetos utiliza classes como mecanismo básico de estruturação de programas. Uma classe é um tipo composto de variáveis (como *records* e *structs*) e procedimentos. Assim, uma classe é uma extensão de *records*/*structs* com a inclusão de comportamento representado por procedimentos. Um exemplo de declaração de classe está neste exemplo:

```
class Store {
    public int get() {
        return n;
    }
    public void put(int pn) {
        n = pn;
    }
    private int n;
}
```

É declarada uma classe `Store` com procedimentos `get` e `put` e uma variável `n`. Na terminologia de orientação a objetos, `get` e `put` são métodos e `n` é uma variável de instância. Esta é a terminologia de Smalltalk, embora neste capítulo utilizemos a sintaxe de Java.

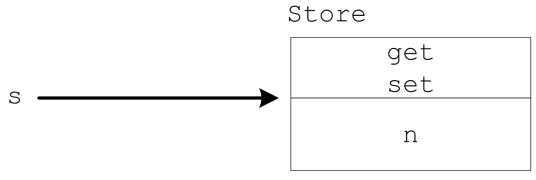
Uma variável da classe `Store`, declarada como

```
Store s;
```

é tratata como se fosse um ponteiro. Assim, deve ser alocada memória para `s` com a instrução

```
s = new Store();
```

Esta memória é um *objeto* da classe `Store`. Um objeto é o valor correspondente a uma classe assim como `3` é um valor do tipo `int` e "`Alo !`" é um valor do tipo `String`. Objetos só existem em execução e classes só existem em tempo de compilação,[1] pois são tipos. Classes são esqueletos dos quais são criados objetos e variáveis referem-se a objetos. Então o objeto referenciado por `s` possui uma variável `n` e dois métodos:



[1] Pelo menos por enquanto.

Os campos de um `record` de Pascal ou `struct` de C são manipulados usando-se "`.`" como em "`pessoa.nome`" ou "`produto.preco`". Objetos são manipulados da mesma forma:

```
s.put(5);
i = s.get();
```

Contudo, fora da classe `Store` apenas os métodos públicos são visíveis. É então ilegal fazer

```
s.n = 5;
```

já que `n` é privado à classe. Métodos e variáveis públicos são prefixados pela palava-chave `public`. Idem para métodos e variáveis privadas.[2]

Alocando dois objetos, como em

```
void main() {
    Store s, t;
    s = new Store();
    t = new Store();
    s.put(5);
    t.put(12);
    System.out.println(s.get() + " " + t.get() );
}
```

são alocados espaços para duas variáveis de instância `n`, uma para cada objeto. Em "`s.put(5)`", o método `put` é chamado e o uso de `n` na instrução

```
n = pn
```

de `put` refere-se a "`s.n`". Um método só é invocado por meio de um objeto. Assim, as referências a variáveis de instância em um método referem-se às variáveis de instância deste objeto. Afinal, os métodos são feitos para manipular os dados do objeto, adicionando comportamento ao que seria uma estrutura composta apenas por dados. Na nomenclatura de orientação a objetos, uma instrução "`s.put(5)`" é o envio da mensagem "`put(5)`" ao objeto referenciado por `s` (ou objeto `s` para simplificar).

3.1 Proteção de Informação

Em algumas linguagens, as variáveis de instância só são manipuladas por meio dos métodos da classe — todas são privadas. Dizemos que estas linguagens suportam *proteção de informação*.

Para exemplificar este conceito, usaremos a classe Pilha:

```
class Pilha {
    private static final Max = 100;
    private int topo;
    private int []vet;

    public void crie() {
        topo = -1;
        vet = new int[Max];
    }
    public boolean empilhe(int elem) {
        if ( topo >= Max - 1) return false;
        else {
```

[2] Java admite variáveis de instância privadas. Contudo, este fato não será utilizado neste livro.

```
            ++topo;
            vet[topo] = elem;
            return true;
        }
    }
    public int desempilhe() {
        if ( topo < 0 ) return -1;
        else {
            elem = vet[topo];
            topo = topo - 1;
            return elem;
        }
    }
    public boolean vazia() {
        return topo < 0;
    }
}
```

Uma pilha é uma estrutura de dados onde o último elemento inserido, com `empilhe`, é sempre o primeiro a ser removido, com `desempilhe`. Esta estrutura espelha o que geralmente acontece com uma pilha de objetos quaisquer.

Esta pilha poderia ser utilizada como no programa abaixo.

```
void main() {
    Pilha p, q;

    p = new Pilha();
    p.crie();   // despreza o valor de retorno
    p.empilhe(1);
    p.empilhe(2);
    p.empilhe(3);
    while ( ! p.vazia() )
        System.out.println( p.desempilhe() );
    q = new Pilha();
    q.crie();
    q.empilhe(10);
    if ( ! p.empilhe(20) )
        erro();
}
```

O programador que usa `Pilha` só pode manipulá-la por meio de seus métodos, sendo um erro de compilação o acesso direto às suas variáveis de instância:

```
p.topo = p.topo + 1; // erro de compilacao
p.vet[p.topo] = 1;   // erro de compilacao
```

A proteção de informação possui três características principais:

1. torna mais fácil a modificação de representação da classe, isto é, a estrutura de dados usada para a sua implementação. No caso de `Pilha`, a implementação é um vetor (`vet`) e um número inteiro (`topo`).

Suponha que o projetista de `Pilha` mude a estrutura de dados para uma lista encadeada, retirando o vetor `vet` e a variável `topo` e resultando na seguinte classe:

```
class Pilha {
    private Elem topo;

    public void crie() {
        topo = null;
    }
    ...
    public boolean vazia() {
        return topo == null;
    }
}
```

O que foi modificado foi o código dos métodos (veja acima), não a interface/assinatura deles.[3] Assim, todo o código do procedimento `main` visto anteriormente não será afetado. Por outro lado, se o usuário tivesse declarado `vet` e `topo` como públicos e usado

```
p.topo = p.topo + 1;
p.vet[p.topo] = 1
```

para empilhar `1`, haveria um erro de compilação com a nova representação de `Pilha`, pois esta não possui vetor `vet`. E o campo `topo` é do tipo `Elem`, não mais um inteiro;

2. o acesso aos dados de `Pilha` (`vet` e `topo`) por métodos tornam a programação de mais alto nível, mais abstrata. Lembrando, abstração é o processo de desprezar detalhes irrelevantes para o nosso objetivo, concentrando-se apenas no que nos interessa.

   Nesse caso, a instrução

   ```
   p.empilhe(1)
   ```

   é mais abstrata do que

   ```
   p.topo = p.topo + 1;
   p.vet[p.topo] = 1;
   ```

   porque ela despreza detalhes irrelevantes para quem quer empilhar um número (`1`), como que a `Pilha` é representada como um vetor, que esse vetor é `vet`, que `p.topo` é o topo da pilha, que `p.topo` é inicialmente `-1`, etc;

3. os métodos usados para manipular os dados (`crie`, `empilhe`, `desempilhe`, `vazia`) conferem a utilização adequada dos dados. Por exemplo, "`p.empilhe(1)`" confere se ainda há espaço na pilha, enquanto que em nas duas instruções alternativas mostradas acima o usuário se esqueceu disto. Resumindo, é mais seguro usar Proteção de Informação porque os dados são protegidos pelos métodos.

3.2 Herança

Herança é um mecanismo que permite a uma classe `B` herdar os métodos e variáveis de instância de uma classe `A`. Tudo se passa como se em `B` tivessem sido definidos os métodos e variáveis de instância de `A`. A herança de `A` por `B` é feita com a palavra chave `extends` como mostrado no exemplo abaixo.

[3] Considere que a interface ou assinatura de um método é composto pelo seu nome, tipo dos parâmetros e tipo de retorno.

```
class A {
    public void put(int pn) {
        n = pn;
    }
    public int get() {
        return n;
    }
    private int n;
}

class B extends A {
    public void imp() {
        System.out.println( get() );
    }
}
```

A classe `B` possui todos os métodos definidos em `A` mais aqueles definidos em seu corpo:

```
void put(int pn)
int get()
void imp()
```

Assim, podemos utilizar todos estes métodos com objetos de `B`:

```
void main() {
    B b;
    b = new B();
    b.put(12);         // invoca A::put
    b.imp();           // invoca B::imp
    System.out.println( b.get() );  // invoca A::get

}
```

`A::put` é o método `put` da classe `A`. A classe `B` é chamada de "subclasse de `A`" e `A` é a "superclasse de `B`".[4]

O método `B::imp` possui uma chamada para um método `get`. O método invocado será `A::get`. Esta chamada poderia ser escrita como "`this.get()`" pois `this`, dentro de um método, refere-se ao objeto que recebeu a mensagem que causou a execução do método. Assim, o envio de mensagem "`b.put(5)`" causa a execução do método `A::put` e conseqüentemente da atribuição "`n = pn`". O "`n`" refere-se a "`b.n`" e poderíamos ter escrito esta atribuição como "`this.n = pn`". `this` é o objeto que recebeu a mensagem, `b`. Em outras linguagens como Smalltalk, Self e Cyan, usa-se `self` ao invés de `this`.

A classe `B` pode redefinir métodos herdados de `A`:

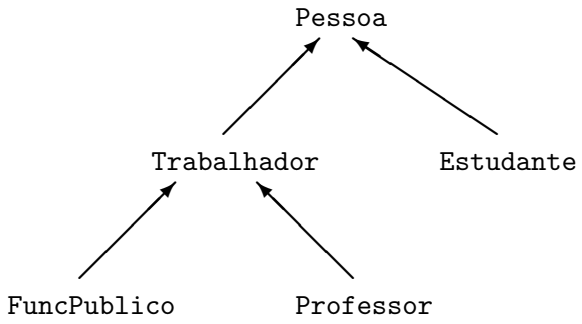
```
class B extends A {
    public int get() {
        return super.get() + 1;
    }
    public void imp() {
        System.out.println( get() );
```

[4]Na terminologia usualmente empregada em C++, `A` é a "classe base" e `B` a "classe derivada".

```
    }
}
```

"`super.get()`" invoca o método `get` da superclasse de `B`, que é `A`. Na chamada a `get` em `imp`, o método `get` a ser usado é o mais próximo possível na hierarquia de classes , que é `B::get`.

Herança é utilizada para expressar relacionamentos do tipo "é um". Por exemplo, um estudante é uma pessoa, um funcionário público é um trabalhador, um professor é um trabalhador, um trabalhador é uma pessoa. Estes relacionamentos são mostrados abaixo, na qual a herança de `A` por `B` é representada através de uma seta de `B` para `A`. A subclasse sempre aparecerá mais embaixo nas figuras.



Uma subclasse é sempre mais específica do que a sua superclasse. Assim, um trabalhador é mais específico do que uma pessoa porque todo trabalhador *é uma* pessoa, mas o contrário nem sempre é verdadeiro. Se tivéssemos feito `Pessoa` herdar `Trabalhador`, haveria um erro lógico no programa, mesmo se não houvesse nenhum erro de compilação.

Considere agora a hierarquia de classes dadas abaixo:

```
class Figura {
    public Figura(int px, int py) {
        x = px;
        y = py;
    }
    public void imp() {
        System.out.println( "Centro(" + x + ", " + y +  ")" );
    }
    public int getX() {
        return x;
    }
    public int getY() {
        return y;
    }
    private int x, y;
}

class Circulo extends Figura
    public Circulo(float p_raio, int x, int y) {
        super(x, y);
        raio = p_raio;
    }
    public void setRaio( float p_raio ) {
```

```
        raio = p_raio;
    }
    public float getRaio() {
        return raio;
    }
    public void imp() {
        System.out.println("raio = " + raio);
        super.imp();
    }
    public float getArea() {
        return PI*raio*raio;
    }
    private float raio;
}
```

Se a classe `Circulo` precisar utilizar as variáveis `x` e `y` herdadas de `Figura`, ela deverá chamar os métodos `getX()` e `getY()` desta classe. Uma subclasse *não* pode manipular diretamente a parte privada da superclasse. Se isto fosse permitido, modificações na representação de uma classe poderiam invalidar as subclasses.

Classe `Figura` possui um método com este mesmo nome. Este método é chamado de *construtor* da classe. Quando um objeto é criado, com `new`, os parâmetros para `new` são passados ao construtor. No construtor da classe `Circulo` há uma instrução "`super(x, y)`". Esta instrução invoca o construtor da superclasse `Figura`.

Algumas linguagens, como C++ e Eiffel, permitem que uma classe herde de mais de uma superclasse. Esta facilidade causa uma ambigüidade quando dois métodos de mesmo nome são herdados de duas superclasses diferentes. Por exemplo, suponha que uma classe `JanelaTexto` herde de `Texto` e `Janela` e que ambas as superclasses definam um método `getNome`. Que método o envio de mensagem "`jt.getNome()`" deverá invocar se o tipo de `jt` for `JanelaTexto`? Em C++, há duas formas de se resolver esta ambigüidade:

1. a primeira é especificando-se qual superclasse se quer utilizar:
   ```
   nome = jt.A::getNome()
   ```
2. a segunda é definir um método `getNome` em `JanelaTexto`.

Em Eiffel o nome do método `getNome` herdado de `Texto` ou `Janela` deve ser renomeado, evitando assim a colisão de nomes.

Uma linguagem em que é permitido a uma classe herdar de mais de uma superclasse suporta *herança múltipla*. Este conceito, aparentemente muito útil, não é muito utilizado em sistemas reais e torna os programas mais lentos porque a implementação de envio de mensagens é diferente do que quando só há herança simples. Herança múltipla pode ser simulada, pelo menos parcialmente, declarando-se um objeto da superclasse na subclasse e redirecionando mensagens a este objeto:

```
class B {
    public B() {
        a = new A();
    }
    public int get() { return a.get(); }
    public void put(int pn) { a.put(pn); }
```

```
    public void imp() { System.out.println(get()); }
    public A getA() { return a; }
}
```

Há ainda outro problema com herança múltipla. Considere que as classes `B` e `C` herdem da classe `A` e a classe `D` herde de `B` e `C`, formando um losango. Um objeto da classe `D` também é um objeto de `A`, `B` e `C`. Este objeto deve ter todas as variáveis de `A`, `B` e `C`. Mas deve o objeto ter um ou dois conjuntos de dados de `A`? Afinal, a classe `D` herda `A` por dois caminhos diferentes. Em alguns casos, seria melhor `D` ter dois conjuntos de dados de `A`. Em outros, é melhor ter apenas um conjunto. Veja estes exemplos:

- a classe `Pessoa` é herdada por `Estudante` e `Atleta`, que são herdadas por `BolsistaAtleta`.[5] Neste caso, deve-se ter um único nome em objetos da classe `BolsistaAtleta`;
- a classe `Trabalhador` é herdada por `Professor` e `Gerente`, que são herdados por `ProfessorGerente`.[6] Neste caso, deve-se ter os dados do trabalhador, como tempo de serviço e nome do empregador, duplicados em objetos de `ProfessorGerente`. Seria interessante que `Trabalhador` herdasse de `Pessoa`. Assim, um objeto de `ProfessorGerente` teria apenas uma variável para nome e outros dados básicos.

Algumas linguagens optam por uma destas opções enquanto que outras permitem que se escolha uma delas no momento da herança.

Praticamente todas as linguagens orientadas a objeto definem uma classe que é superclasse de todas as outras. Esta classe é chamada de `Object` em Smalltalk, Java e C#. Nesta classe são colocados métodos aplicáveis a todos os objetos, como `equals`, `clone`, `toString` e `hashCode`.

3.3 Polimorfismo

Se o tipo de uma variável `w` for uma classe `T`, a atribuição

```
w = null;
```

estará correta qualquer que seja `T`. Isto é possível porque `null` é um valor polimórfico: ele pode ser usado onde se espera uma referência para objetos de qualquer classe. Polimorfismo quer dizer faces múltiplas e é o que acontece com `null`, que possui infinitos tipos.

Em Java, uma variável cujo tipo é uma classe pode referir-se a objetos de subclasses desta classe. O código

```
void main() {
    Figura f;
    Circuo c;
    c = new Circulo();
    c.init( 20.0, 30, 50 );
    f = c;
    f.imp();
}
```

está correto. A atribuição

```
f = c;
```

atribui uma referência para `Circulo` a uma variável de `Figura`.

[5] O atleta ganha uma bolsa de estudos por ser atleta.

[6] O professor trabalha em tempo parcial e também é um gerente.

Java permite atribuições do tipo

```
Classe = Subclasse
```

como a acima, que é

```
Figura = Circulo
```

Uma variável cujo tipo é uma classe `A` sempre será polimórfica, pois ela poderá apontar para objetos de `A` ou qualquer objeto de subclasses de `A`.

Agora, qual método

```
f.imp()
```

irá invocar? `f` referencia um objeto de `Circulo` e, portanto, seria natural que o método invocado fosse "`Circulo::imp`". Contudo, o tipo de `f` é "`Figura`" e "`f.imp()`" também poderia invocar `Figura::imp`.

O envio de mensagem "`f.imp()`" invocará o método `imp` de `Circulo`. Será feita uma busca em *tempo de execução* por método `imp` na classe do objeto apontado por `f`. Se `imp` não for encontrado nesta classe, a busca continuará na superclasse, superclasse da superclasse e assim por diante. Quando o método for encontrado, ele será chamado. Sendo a busca feita em tempo de execução, será sempre chamado o método mais adequado ao objeto, isto é, se `f` estiver apontando para um círculo, será chamado o método `imp` de `Circulo`, se estiver apontando para um retângulo, será chamado `Retangulo::imp` e assim por diante.

A instrução "`f.imp()`" causará uma busca em *tempo de compilação* por método `imp` na classe *declarada* de `f`, que é `Figura` (`f` é declarado como "`f : Figura`"). Se `imp` não fosse encontrado lá, a busca continuaria na superclasse de `Figura` (se existisse), superclasse da superclasse e assim por diante. Se o compilador não encontrar o método `imp`, ele sinalizará um erro. Isto significa que uma instrução

```
f.setRaio(10);
```

será ilegal mesmo quando tivermos certeza de que `f` apontará em tempo de execução para um objeto de `Circulo` (que possui método `setRaio`). A razão para esta restrição é que o compilador não pode garantir que `f` apontará para um objeto que possui método `setRaio`. A inicialização de `f` pode depender do fluxo de controle:

```
void m(int i) {
    Figura f, aFig;
    Circulo aCir;

    aCir = new Circulo(20.0, 50, 30);
    aFig = new Figura(30, 40 );
    if ( i > 0 )
      f = aFig;
    else
      f = aCir;
    f.setRaio(10);
    ...

}
```

Se este método fosse legal, `f` poderia ser inicializado com `aFig`. Em tempo de execução, seria feita uma busca por método `setRaio` na classe `Figura` e este método não seria encontrado, resultando em um erro de tempo de execução com o término do programa.

Como resultado da discussão acima, temos que

```
f.imp()
```

será válido quando `imp` pertencer à classe declarada de `f` ou suas superclasses (se existirem). Já que `f` pode apontar para um objeto de uma subclasse de `Figura`, podemos garantir que a classe deste objeto possuirá um método `imp` em tempo de execução? A resposta é "sim", pois `f` pode apontar apenas para objetos de `Figura` ou suas subclasses. O compilador garante, ao encontar

```
f.imp()
```

que a classe declarada de `f`, `Figura`, possui método `imp` e, como todas as subclasses herdam os métodos das superclasses, as subclasses de `Figura` possuirão pelo menos o método `imp` herdado desta classe. Assim, `f` apontará para um objeto de `Figura` ou suas subclasses e este objeto certamente possuirá um método `imp`.

Polimorfismo é fundamental para o reaproveitamento de *software*. Quando um método aceitar como parâmetro um objeto de `Figura`, como

```
void m( Figura f )
```

podemos passar como parâmetro objetos de qualquer subclasse desta classe. Isto porque uma chamada

```
m(aCir);
```

envolve uma atribuição "`f = aCir`", que será correta se for da forma

```
Classe = Subclasse
```

Então, podemos passar como parâmetro a `m` objetos de `Circulo`, `Retangulo`, etc. Não é necessário construir um método `m` para objetos de cada uma das subclasses de `Figura` — um mesmo método `m` pode ser utilizado com objetos de todas as subclasses. O código de `m` é *reaproveitado* por causa do polimorfismo.

Admitindo que as classes `Retangulo` e `Triangulo` existam e são subclasses de `Figura`, o código a seguir mostra mais um exemplo de polimorfismo.

```
void impVet( Figura []v ) {
    int i;
    for (i = 0; i < v.length; ++i)
        v[i].imp();
}


void main() {
    Circulo c1;
    Retangulo r1, r2;
    Triangulo
    Figura vetFig = {
        new Circulo(5, 80, 30),
        new Retangulo(30, 50, 70, 60),
        new Triangulo(10, 18, 30, 20, 40, 25),
        new Retangulo(20, 100, 80, 150)
    };
    impVet( vetFig );
}
```

A função `impVet` percorre o vetor `v` enviando a mensagem `imp` a cada um de seus elementos. O método `imp` executado dependerá da classe do objeto apontado por "`v[i]`".

Existe uma outra forma de polimorfismo em que será mostrada acrescentando-se métodos nas classes `Figura` e `Circulo`:

```
class Figura {
```

```
    public void desenhe() { }
    public void apague() { }
    public void mova( int nx, int ny ) {
        apague();
        x = nx;
        y = ny;
        desenhe();
    }
    private int x, y;
}


class Circulo extends Figura {
    ... // métodos definidos anteriormente
    public void desenhe() {
        // desenhe um círculo
    }
    public void apague() {
        // apague um círculo
    }
    private float raio;
}
```

Os métodos `desenhe` e `apague` de `Figura` não fazem nada porque esta classe foi feita para ser herdada e não para se criar objetos dela.[7] O método `mova` apaga o desenho anterior, move a figura e a desenha novamente. Como `desenhe` e `apague` são vazios em `Figura`, `mova` só faz sentido se `desenhe` e `apague` forem redefinidos em subclasses. Em

```
Circulo c;
c = new Circulo(10.0, 50, 30 );
c.mova( 20, 80 );
...
```

o método invocado em "`c.mova(20, 80)`" será "`Figura::mova`". Este método possui um envio de mensagem

```
apague();
```

que é o mesmo que

```
this.apague();
```

que envia a mensagem `apague` ao objeto que recebeu a mensagem `mova`, que é "`c`". Então, a busca por método `apague` será iniciada em `Circulo` (classe do objeto `c`), onde `Circulo::apague` será encontrado e executado. Da mesma forma, a instrução

```
desenhe()
```

em `Figura::mova` invocará `Circulo::desenhe`.

Observando este exemplo, verificamos que não foi necessário redefinir o método `mova` em `Circulo` — o seu código foi reaproveitado. Se tivermos uma classe `Retangulo`, subclasse de `Figura`, precisaremos de definir apenas `desenhe` e `apague`. O método `mova` será herdado de `Figura` e funcionará corretamente com retângulos. Isto é, o código

[7] Seriam métodos abstratos.

```
Retangulo r;
r = new Retangulo( 30, 50, 70, 20 );
r.mova( 100, 120 );
...
```

invocará os métodos `desenhe` e `apague` de `Retangulo`.

As redefinições de `apague` e `desenhe` em `Circulo` causaram alterações no método `mova` herdado de `Figura`, adaptando-o para trabalhar com círculos. Ou seja, `mova` foi modificado pela redefinição de outros métodos em uma subclasse. Não foi necessário redefinir `mova` em `Circulo` adaptando-o para a nova situação. Dizemos que o código de `mova` foi *reaproveitado* em `Circulo`. O método `mova` se comportará de maneira diferente em cada subclasse, apesar de ser definido uma única vez em `Figura`.

É possível declarar diversos métodos com mesmo nome mas com parâmetros diferentes em diversas linguagens orientas a objeto. Então podemos ter

```
class Output {
    public void print(int n)   { ... }
    public void print(char ch) { ... }
    public void print(float f) { ... }

}
```

Este tipo de construção é chamado de "sobrecarga de métodos" e considerado um tipo de polimorfismo por alguns autores, embora não haja nenhum reuso de código.

3.4 Redefinição de Métodos

Algumas linguagens orientadas a objeto exigem que, ao redefinir um método na subclasse, se use a palavra-chave `override` (ou semelhante). Em Java, usa-se a anotação `@Override`:

```
class Person {
    public void print() { ... }
    ...
}

class Worker extends Person {
    ...
    @Override public void print() { ... }
}
```

Há bons motivos para se usar `override` ao redeclarar um método:

(a) o compilador irá emitir um erro se o programador está redefinindo, sem saber, um método da superclasse;

(b) o compilador emitirá um erro se o programador cometer algum erro no nome do método ou tipos dos parâmetros e tipo de retorno. Isto é, quando ele pensa que está redefinindo um método quando na verdade ele não está;

(c) se um método é adicionado à superclasse e alguma subclasse define um método com o mesmo nome, haverá um erro de compilação. O programador será avisado de que há uma redefinição do método;

(d) fica claramente documentado que o método em questão é uma redefinição.

Algumas linguagens permitem que, em uma redefinição de um método na subclasse, os tipos dos parâmetros e o tipo de retorno sejam diferentes daqueles do método da superclasse. Para que a linguagem seja estaticamente tipada, o tipo `B` de um parâmetro do método na superclasse pode ser substituído por um supertipo `A` de `B`. E o tipo do valor de retorno `C` da superclasse pode ser substituído por um subtipo `D` de `C`. Então se um método é declarado na superclasse como

`C m( B x )`

ele pode ser redeclarado na subclasse como

`D m( A x )`

Em desenho,

`↓m( ↑ )`

Uma regra de tipos como ↓ é chamada de co-variante. A regra do tipo ↑ é chamada de contra-variante (porque ela segue a direção contrária à herança).

Se esta regra não for obedecida, haverá um erro de tipos em execução. No exemplo abaixo, suponha que `D` herde de `C` que herde de `B` que herde de `A`. A classe `X` define apenas um método `mx` no qual `x` é o nome da classe em minúsculo. Assim a classe `C` define apenas um método `mc`.

```
class T {
    public C m( B x ) {
        x.mb();
        return new C();
    }
}

class R extends T {
    public B m( C x ) {
        x.mc();
        return new B();
    }
}

class Test {
    public void error() {
        T t = new R();
        t.m( new B() );        // 1
        C c = t.m( new C() );  // 2
        c.mc();                // 3
    }
}
```

Não há erros de compilação no código acima, assundo que o compilador permite esta redefinição do método `m` que não obedece à regra acima. Contudo, na execução da instrução `// 1`, um objeto de `B` será passado ao método `R::m` — haverá uma atribuição, nesta passagem de parâmetro, do tipo "`superclasse = classe`". Dentro deste método, mensagem `mc` é enviado ao objeto referenciado por `x` do tipo `B`. Esta classe não possui método `mc`, que está apenas presente em `C` e `D`. Em `// 2`, o tipo que o compilador deduz para o retorno do envio de mensagem é `C`, pois o tipo de `t` é `T` e esta classe define um método `m` cujo tipo de retorno é `C`. Mas o objeto retornado é do tipo `B`, pois o método chamado em execução é `R::m`. Este objeto é associado a `c` que recebe à mensagem `mc` em `// 3`. Há um erro de execução, pois `B` não possui este método.

3.5 Classes e Métodos Abstratos

Um método abstrato é declarado com a palavra-chave `abstract`. O corpo da classe não deve ser fornecido. Uma classe também pode ser declarada como `abstract`. Neste caso, ela pode ter zero ou mais métodos abstratos:

```
abstract class Expr {
    abstract public void genJava();
    public Type getType() {
        return type;
    }
    private Type type;
}
```

Classes abstratas são úteis para representar elementos comuns a várias classes em uma superclasses sendo que esta superclasse não representa um elemento do domínio do programa. Por exemplo, a classe `Figura` utilizada anteriormente deveria ter sido declarada como abstrata. Existem triângulos, retângulos, círculos etc, mas não existe uma figura geomética "figura". A classe `Expr` acima pode ser utilizada como superclasse de classes que representam expressões de uma linguagem qualquer. O compilador de Cyan utiliza uma classe semelhante a esta. Todas as expressões têm que ter um tipo (variável de instância `type`) e todas têm que gerar código em Java (método `genJava`). Há classes como `ExprLiteralInt`, subclasse de `Expr`, para representar um inteiro literal. Esta classe não é abstrata. Portanto ela tem que definir todos os métodos abstratos herdados de `Expr` — neste caso, apenas `genJava`. E não há em Cyan um elemento que seja representado por `Expr`, esta funciona apenas como uma superclasse. Por este motivo, não se pode criar objetos de classes abstratas. Mas pode-se utilizá-las como tipo de variáveis e parâmetros, tipo de retorno de métodos e com o operador `instanceof`.

Em resumo, classes abstratas devem obedecer algumas regras:

(a) métodos abstratos não devem definir um corpo e só podem ser colocados em classes abstratas;

(b) uma classe abstrata pode declarar tudo o que uma classe regular pode mais métodos abstratos (possivelmente nenhum método abstrato);

(c) uma classe que herda de uma classe abstrata deve ser declarada abstrata se ela não implementa todos os métodos abstratos herdados ou define novos métodos abstratos;

(d) não se pode criar objetos de uma classe abstrata, mesmo se ela não define nenhum método abstrato.

3.6 Modelos de Polimorfismo

Esta seção descreve quatro formas de suporte a polimorfismo empregado pelas linguagens Smalltalk, POOL-I, Java e C++. Naturalmente, os modelos de polimorfismo descritos nas subseções seguintes são abstrações das linguagens reais e apresentam diferenças em relação a elas.

3.6.1 Smalltalk

Smalltalk [5] é uma linguagem tipada dinamicamente, o que quer dizer que na declaração de uma variável ou parâmetro não se coloca o tipo. Durante a execução, uma variável irá se referir a um objeto e terá o tipo deste objeto. Conseqüentemente, uma variável pode se referir a objetos de tipos diferentes durante a sua existência.

No exemplo abaixo,

```
var a, b;
a = 1;
b = new Janela(a, 5, 20, 30);
a = b;
a.desenhe();
...
```

se a instrução "`a.desenhe()`" for colocada logo após "`a = 1`", haverá o envio da mensagem `desenhe` a um número inteiro. Como a classe dos inteiros não possui um método `desenhe`, ocorrerá um erro de tipos e o programa será abortado.

Considere agora um método

```
void m(y) {
    y.desenhe();
    y.mova(10, 20);
}
```

de uma classe `A`. Assuma que exista uma classe `Janela` em Smalltalk, que é aquela do exemplo abaixo sem os tipos das declarações de variáveis.

```
class Janela {
    public Janela(int px, int py) { x = px. y = py; }
    private int x, y;
    public void desenhe() { ... }
    public void mova(int novo_x, int novo_y) {
        this.x = novo_x;
        this.y = novo_y;
        this.desenhe();
    }
}

class JanelaTexto extends Janela {
    ...
    public void desenhe() { ... }
}
```

Esta classe possui métodos `desenhe` e `mova`, sendo que este último não causa erros de tipo se os seus dois parâmetros são números inteiros.

Se um objeto de `Janela` for passado com parâmetro a `m`, como em

```
a = new A();
a.m( new Janela() );
```

não haverá erros de tipo dentro deste método. Se a `m` for passado um objeto de uma subclasse de `Janela`, também não haverá erros de tipo. A razão é que uma subclasse possui pelo menos todos os métodos da superclasse. Assim, se um objeto de `Janela` sabe responder a todas as mensagens enviadas a ele dentro de `m`, um objeto de uma subclasse também saberá responder a todas estas mensagens. Estamos admitindo que, se `mova` for redefinido em uma subclasse, ele continuará aceitando dois inteiros como parâmetros sem causar erros de tipo.

De fato, o método `m` pode aceitar como parâmetros objetos de *qualquer* classe que possua métodos `mova` e `desenhe` tal que `mova` aceita dois inteiros como parâmetros e `desenhe` não possui parâmetros. Não é necessário que esta classe herde de `Janela`. Este sistema de tipos, sem restrição nenhuma que não seja a capacidade dos objetos de responder às mensagens que lhe são enviadas, possui o maior grau possível de polimorfismo.

Se `m` for codificado como

```
void m(y, b) {
    y.desenhe();
    y.mova(10, 20);
    if ( b )
        y.icon();
}
```

o código

```
a = new A();
a.m( new Janela(), false );
```

não causará erro de tipos em tempo de execução, pois a mensagem `icon` não será enviada ao objeto de `Janela` em execução. Se fosse enviada, haveria um erro já que a classe `Janela` não possui método `icon`.

Em geral, o fluxo de execução do programa, controlado por `if`'s, `while`'s e outras estruturas, determina quais mensagens são enviadas para cada variável. E este mesmo fluxo determina a capacidade de cada variável de responder a mensagens. Para compreender melhor estes pontos, considere o código

```
if ( b > 0 )
    a = new Janela();
else
    a = 1;
if ( c > 1 )
    a.desenhe();
else
    a = a + 1;
```

O primeiro `if` determina quais as mensagens `a` pode responder, que depende da classe do objeto a que `a` se refere. O segundo `if` seleciona uma mensagem a ser enviada à variável `a`. Em Smalltalk, "`+ 1`" é considerado um envio de mensagem.

Então, o fluxo de execução determina a corretude de tipos de um programa em Smalltalk, o que torna os programas muito inseguros. Alguns trechos de código podem revelar um erro de tipos após meses de uso. Note que, como é impossível prever todos os caminhos de execução de um programa em tempo de compilação, é também impossível garantir estaticamente que um programa em Smalltalk é corretamente tipado.

A linguagem Smalltalk (a linguagem real) emprega *seletores* para a definição de métodos e envios de mensagem. Um método unário, sem parâmetros, consiste de um nome simples como `name` ou `age`. Já um método com parâmetros deve ter um seletor para cada parâmetros, sendo que cada seletor é seguido, sem espaços em branco, por ':'. Então um método para inicializar o nome e a idade de um objeto pessoa poderia ser chamado como

```
pessoa name: 'Isaac Newton' age: 25.
```

Cyan utiliza uma sintaxe semelhante com duas diferenças: (a) um seletor com '`:`' não precisa ser seguido por parâmetro e (b) cada seletor pode ter mais de um parâmetro.

Um método que inicializa um protótipo[8] `Circle` em Cyan pode ser declarado como

```
object Circle
    fun x: (Int nx) y: (Int ny) radius: (Int nr) {
        x = nx;
        y = ny;
        radius = nr;
    }
    ...
end
```

Este método é chamado pela instrução

```
Circle x: 10 y: 40 radius: 5;
```

Uma conseguência desta sintaxe é que os programas se tornam muito legíveis. Neste último envio de mensagem sabe-se claramente qual o `x`, o `y` e o raio do círculo. Contraste esta instrução com a equivalente em outras linguagens:

```
circle.set(10, 50, 5);
```

3.6.2 POOL-I

Esta seção descreve o modelo das linguagens POOL-I [1] e Green [8] [7]. Como o sistema de tipos de Green foi parcialmente baseado no de POOL-I, este modelo será chamado de modelo POOL-I. Green e POOL-I são linguagens estaticamente tipada, pois todos os erros de tipo são descobertos em compilação.

Neste modelo, o tipo de uma classe é definido como o conjunto das interfaces (assinaturas ou *signatures*) de seus métodos públicos (construtores excluídos). A interface de um método é o seu nome, tipo do valor de retorno (se houver) e tipos de seus parâmetros formais (o nome dos parâmetros é desprezado). Por exemplo, o tipo da classe `Janela` dada anteriormente é

```
{ desenhe(), mova(int, int) }
```

sendo que { e } são utilizados para delimitar os elementos de um conjunto, como em matemática. Um tipo U será subtipo de um tipo T se U possuir pelo menos as interfaces que T possui. Isto é, $T \subset U$. Como exemplo, o tipo da classe `JanelaProcesso` é um subtipo do tipo da classe `Janela`.

```
class JanelaProcesso {
    public void desenhe() { ... }
    public void mova( int nx, int ny )   { ... }

    public void iniciaProcesso()         { ... }
    public void setProcesso( String s )  { ... }
}
```

Como abreviação, dizemos que a classe `JanelaProcesso` é subtipo da classe `Janela`.

Quando uma classe `B` herdar de uma classe `A`, diremos que `B` é *subclasse* de `A`. Neste caso, `B` herdará todos os métodos públicos de `A`, implicando que `B` é *subtipo* de `A`.[9] Observe que toda subclasse é também subtipo, mas é possível existir subtipo que não é subclasse — a classe `JanelaProcesso` da é subtipo mas não subclasse de `Janela`.

[8] Como veremos em breve, Cyan declara protótipos e não classes.

[9] A linguagem POOL-I, ao contrário deste modelo, permite subclasses que não são subtipos. Em Green, todas as subclasses são subtipos.

Neste modelo, uma atribuição

```
t = s
```

estará correta se a classe declarada de `s` for subtipo da classe declarada de `t`. As atribuições do tipo

```
Tipo = SubTipo;
```

são válidas.

Esta restrição permite a detecção de todos os erros de tipo em tempo de compilação, por duas razões:

- Em um envio de mensagem
  `t.m(`b_1`, `b_2`, ... `b_n`)`
  o compilador confere se a classe com que `t` foi declarada possui um método chamado `m` cujos parâmetros formais possuem tipos T_1, T_2, ... T_n tal que o tipo de b_i é subtipo de T_i, $1 \leq i \leq n$. A regra "`Tipo = Subtipo`" é obedecida também em passagem de parâmetros.

- Ao executar este envio de mensagem, é possível que `t` não se refira a um objeto de sua classe, mas de um subtipo do tipo da sua classe, por causa das atribuições do tipo `Tipo = SubTipo`, como `t = s`. De qualquer forma, não haverá erro de execução, pois tanto a sua classe quanto qualquer subtipo dela possuem o método `m` com parâmetros formais cujos tipos são T_1, ... T_n.

Em uma declaração

```
var A a
```

a variável `a` é associada ao *tipo* da classe `A` e não à classe `A`. Deste modo, `a` pode se referir a objetos de classes que são subtipos sem serem subclasses de `A`. Este é o motivo pelo qual a declaração da variável `a` não aloca memória automaticamente para um objeto da classe `A`.

3.6.3 C++

C++ [17] é uma linguagem estaticamente tipada em que todo subtipo é subclasse. Portanto, as atribuições válidas possuem a forma

```
Classe = Subclasse
```

Assume-se que a variável do lado esquerdo da atribuição seja um ponteiro e que o lado direito seja uma referência para um objeto:

```
Figura *p;
...
p = new Circulo(150, 200, 30);
```

Não há polimorfismo em C++ quando não se utiliza ponteiros. Se `p` fosse declarado como "`Figura p;`", ele poderia receber apenas objetos de `Figura` em atribuições. Neste modelo assume-se que não há variáveis cujo tipo sejam classes, apenas ponteiros para classes.

O motivo pelo qual o modelo C++ exige que subtipo seja também subclasse é o desempenho. Uma chamada de método é feita através de um vetor de ponteiros para funções e é apenas duas ou três vezes mais lenta do que uma chamada de função normal.

C++ suporta métodos virtuais e não virtuais, sendo que nestes últimos a busca pelo método é feita em compilação — a ligação mensagem/método é estática. Nesta subseção, consideramos que todos os métodos são virtuais.

3.6.4 Java

Java [13] [11] suporta apenas herança simples. Contudo, a linguagem permite a declaração de *interfaces* que podem ser utilizados em muitos casos em que herança múltipla deveria ser utilizada. Uma interface *declara* assinaturas (*signatures* ou interfaces) de métodos:

```
interface Printable {
   void print();
}
```

Uma assinatura de um método é composto pelo tipo de retorno, o nome do método e os parâmetros e seus tipos (sendo os nomes dos parâmetros opcionais).

Uma classe pode *implementar* uma interface:

```
class Worker extends Person implements Printable {
   ...
   public float getSalary() { ... }
   void print() { ... }
}
```

Quando uma classe implementa uma interface, ela é obrigada a definir (com o corpo) os métodos que aparecem na interface. Se a classe `Worker` não definisse o método `print`, haveria um erro de compilação. Uma classe pode herdar de uma única classe mas pode implementar várias interfaces diferentes.

Este modelo considera as interfaces como classes de uma linguagem com herança múltipla exceto que as interfaces não podem *definir* métodos. Interfaces são similares a classes abstratas[10] e tudo se passa como se o modelo admitisse herança múltipla onde todas as classes herdadas são completamente abstratas (sem nenhum corpo de método) exceto possivelmente uma delas. Um *tipo* neste modelo é uma classe ou uma interface. Subtipo é definido indutivamente como:

(a) uma classe `C` é subtipo dela mesma;

(b) uma interface `I` é subtipo dela mesma;

(c) se uma classe `B` herda de uma classe `A`, `B` é subtipo de `A`;

(d) se uma interface `J` herda de uma interface `I`, `J` é subtipo de `I`;

(e) se `R` é subtipo de `S` e `S` é subtipo de `T`, então `R` é subtipo de `T`.

Em resumo, para descobrir as relações de subtipo desenhe um grafo no qual os vértices são as classes e arestas e no qual há aresta de `X` para `Y` se `X` herda de `Y` ou `X` implementa `Y`. Então `X` é subtipo de `Y` se há um caminho de `X` para `Y`.

As atribuições válidas em Java são

```
Tipo = subtipo
```

Pode-se declarar uma variável cujo tipo é uma interface. Como exemplo, o código abaixo é válido.

[10] A diferença é que classes abstratas podem declarar variáveis de instância e o corpo de alguns métodos. E podem possuir métodos privados. Em uma interface, todos os métodos são públicos.

```
Printable p;
Person person;
person = new Worker();  // cria objeto de Worker
p = person;
p.print();
p = new NightWorker(); // NightWorker é subclasse de Worker
```

Java é estaticamente tipada. Então, se o tipo de uma variável é uma interface, apenas métodos com assinaturas declaradas na interface e métodos da superclass `Object` podem ser chamadas por meio da variável. Por exemplo, por meio de `p` acima pode-se chamar apenas o método `print` e aqueles de `Object`.

Interfaces em Java são uma forma de adicionar os benefícios de herança múltipla à linguagem mas sem alguns dos problemas desta facilidade (como duplicação dos dados de objetos herdados por mais de um caminho — veja página 40).

3.6.5 Comparação entre os Modelos de Polimorfismo e Sistema de Tipos

Agora podemos comparar o polimorfismo dos modelos de linguagens descritos acima. Considere o método `Q` no modelo C++.

```
public void Q( Janela x, int y ) {
    x.desenhe();
    if ( y > 1 )
        x.mova(20, 5);
}
```

Ele pode receber, como primeiro parâmetro (`x`), objetos da classe `Janela` ou qualquer *subclasse* desta classe.

Em Java, se `Janela` é uma interface, o primeiro parâmetro passado a `Q` pode ser objeto de quaisquer classes que implementem esta interface ou que herdem das classes que implementam esta interface. As classes que implementam uma interface geralmente não têm nenhuma relação de herança entre si. Se quisermos passar um objeto de uma classe A para `Q`, basta fazer com que `A` implemente a interface `Janela`. Isto é, `A` deveria implementar os métodos definidos em `Janela` e que possivelmente são utilizados no corpo de `Q`.

Em uma linguagem com herança simples e que não suporte interfaces (como definidas em Java), apenas objetos de `Janela` e suas subclasses poderiam ser passados como primeiro parâmetro (assumindo então que `Janela` é uma classe e não um interface). Para passar objetos de uma classe `A` como parâmetros, deveríamos fazer esta classe herdar de `Janela`, o que não seria possível se `A` já herdasse de uma outra classe.

Se `Janela` for uma classe, poderão ser passados a `Q`, como primeiro parâmetro, objetos da classe `Janela` ou qualquer *subclasse* desta classe, como em C++.

Em POOL-I, os parâmetros passados a `Q` podem ser de qualquer *subtipo* de `Janela`. Todas as classes que herdam de `Janela` (subclasses) são subtipos desta classe e há subtipos que *não* são subclasses. Ou seja, o conjunto dos subtipos de `Janela` é potencialmente maior que o de subclasses de `Janela`. Conseqüentemente, em POOL-I o procedimento `Q` pode ser usado com mais classes do que em C++, pois o conjunto de classes aceito como parâmetro para `Q` nesta última linguagem (subclasses) é potencialmente menor que o conjunto aceito por POOL-I (subtipos).

Em C++, Java e POOL-I, o compilador confere, na compilação de `Q`, se a classe/interface de `x`, que é `Janela`, possui métodos correspondentes às mensagens enviadas estaticamente a `x`. Isto é, o compilador confere se `Janela` possui métodos `desenhe` e `mova` e se `mova` admite dois inteiros como

parâmetros. Estaticamente é garantido que objetos de `Janela` podem ser passados a `Q` (como primeiro parâmetro) sem causar erros de tipo. Em tempo de execução, objetos de subclasses ou subtipos de `Janela` serão passados a `Q`, por causa de atribuições do tipo `Tipo = SubTipo`. Estes objetos saberão responder a todas as mensagens enviadas a eles dentro de `Q`, pois: a) eles possuem pelo menos todos os métodos que objetos da classe `Janela` possuem; b) objetos da classe `Janela` possuem métodos para responder a todas as mensagens enviadas ao parâmetro `x` dentro de `Q`.

Smalltalk dispensa tipos na declaração de variáveis e, portanto, o procedimento `Q` neste modelo seria

```
public void Q( x, y ) {
    x.desenhe();
    if ( y > 1 )
        x.mova(20, 5);
}
```

Como nem `x` nem `y` possuem tipos, não se exige que o objeto passado como primeiro parâmetro real a `Q` possua métodos `desenhe` e `mova`. De fato, na instrução

```
Q(a,0)
```

é enviada mensagem `desenhe` ao objeto referenciado por `x` (e também por `a`), mas não é enviada a mensagem `mova`.

Como conseqüência, esta instrução pode ser executada com parâmetros `a` de qualquer classe que possua um método `desenhe` sem parâmetros. Ao contrário de POOL-I, Java e C++, a classe do parâmetro `x` de `Q` não precisa possuir também o método `mova`. Logo, o método `Q` pode ser usado com um conjunto de classes (para o parâmetro `x`) potencialmente maior que o conjunto de classes usadas com o procedimento `Q` equivalente de POOL-I. Portanto, Smalltalk possui mais polimorfismo que POOL-I.

Em linguagens convencionais, uma atribuição `a = b` será correta se os tipos de `a` e `b` forem iguais ou `b` puder ser convertido para o tipo de `a` (o que ocorre com perda de informação se o tipo de `b` for mais abrangente do que o de `a`). Em POOL-I, `a = b` será válido se a classe de `b` for subtipo da classe de `a`. Em C++, se a classe de `b` for subclasse da classe de `a`. Em Java, se a classe de `b` for subclasse da classe de `a` (se o tipo de `a` for uma classe) ou implementar (direta ou indiretamente) a interface que é o tipo de `a` (se o tipo de `a` for uma interface) ou se a interface que é o tipo de `b` herdar da interface que é o tipo de `a`. Em Smalltalk, esta operação será sempre correta. Logo, as linguagens orientadas a objeto citadas estendem o significado da atribuição permitindo um número maior de tipos do seu lado direito. Como em passagem de parâmetros existe uma atribuição implícita, procedimentos e métodos podem aceitar parâmetros reais de mais classes do que normalmente aceitariam, o que é o motivo do reaproveitamento de código. Concluindo, podemos afirmar que a mudança do significado da atribuição é o motivo de todo o reaproveitamento de *software* causado pelo polimorfismo descrito neste artigo. Quando mais liberal (Smalltalk — nenhuma restrição ao lado direito de `=`) é a mudança, maior o polimorfismo.

Suponha que estejamos construindo um novo sistema de janelas e seja necessário construir uma classe `Window` que possua os mesmos métodos que `Janela`. Contudo, `Window` possui uma aparência visual e uma implementação bem diferentes de `Janela`. Certamente, é interessante poder passar objetos de `Window` onde se espera objetos de `Janela`. Todo o código construído para manipular esta última classe seria reusado pela classe `Window`.

Em C++, `Window` deve herdar de `Janela` para que objetos de `Window` possam ser usados onde se espera objetos de `Janela`. Como `Window` possui uma implementação completamente diferente de `Janela`, as variáveis de instância da superclasse `Janela` não seriam usadas em objetos de `Window`.

Então, herança estaria sendo utilizada para expressar *especificação* do problema e não *implementação*. *Especificação* é o que expressa a regra "um objeto de uma subclasse é um objeto de uma superclasse". *Implementação* implica que uma subclasse possui pelos menos as variáveis de instância da sua superclasse e herda algumas ou todas as implementações dos métodos.

Em POOL-I, este problema não existe, pois a *especificação* e *implementação* são desempenhados por mecanismos diferentes. A saber, subtipagem e herança. Em Java, o programador poderia fazer `Janela` e `Window` herdarem de uma interface com todos os métodos originais de `Janela`. Mas isto só será possível se o código fonte de `Janela` estiver disponível. Em POOL-I, isto não é necessário.

Existe um problema ainda maior em C++ por causa da ligação subtipo-subclasse. Considere que a classe `Janela` possua um método

```
public wrong( Janela outra ) {
    ...
    w = outra.x;
    ...
}
```

Dentro deste método é feito um acesso à variável `x` do parâmetro `outra`. Esta variável de instância foi, naturalmente, declarada em `Janela`. Se este parâmetro refere-se a um objeto de `Window`, subclasse de `Janela`, então `outra.x` não foi inicializado. A razão é que a classe `Window` não utiliza as variáveis herdadas de `Janela` e, portanto, não as inicializa. Note que o mesmo problema ocorre com a linguagem Java.

3.7 Herança Mixin

Uma classe mixin é um tipo de classe suportada por algumas linguagens que permite herança múltipla sem alguns dos problemas associados a este tipo de construção. O suporte a mixins varia largamente entre linguagens. Descreveremos estas classes em uma linguagem hipotética.

```
mixin class AgeMix {
    private Int age;
    public String getAge() { return age; }
    public void setAge(int age) { this.age = age; }
}

mixin class NameMix {
    public abstract String getFirstName();
    public abstract String getLastName();
    public String getName() {
        return getFirstName() + " " + getLastName();
    }
}

class Person with NameMix AgeMix {
    private String firstName, lastName;
    public String getFirstName() { return firstName; }
    public String getLastName() { return lastName; }

}
```

A herança `mixin` é feita com a palavra-chave `with` seguida das classes mixin. À classe `Person` são adicionados os métodos e variáveis de instância das classes mixin (elas são *mixed into* `Person`). A diferença com herança múltipla é que classes mixin não podem ser usadas para criar objetos. Então elas podem ser incompletas, podem possuir métodos abstratos que são fornecidos pela classe ao qual elas são acopladas. No exemplo acima, `NameMix` exige que a classe ao qual ela é acoplada tenha métodos `getFirstName` e `getLastName`. Também não há colisão de nomes pois a hierarquia é linearizada. Se ambas as classes `NameMix` e `AgeMix` definissem um método `m`, não haveria colisão, pois `Person` herdaria o método de `NameMix` (pois esta classe está primeiro na lista após a palavra-chave `with`).

A classe `Person` terá métodos `getAge`, `setAge`, `getName`, `getFirstName` e `getLastName`. E variáveis de instância `age`, `firstName` e `lastName`. As classes mixins são classes que não possuem superclasse. Elas não herdam de ninguém.

3.8 Linguagens Baseadas em Protótipos

As linguagens vistas até agora são baseadas em classes, que são utilizadas para estruturar o programa. Classes são esqueletos a partir dos quais objetos são criados. Na maioria das linguagens, elas não existem em tempo de execução — a menos de menção em contrário, assumiremos isto. Objetos existem em execução, classes apenas em compilação.

Nas linguagens baseadas em protótipos não há classes. O papel destas é reservado aos protótipos, que são declarações literais de objetos. Por exemplo, a classe `Store` do início deste capítulo seria transformada, em Cyan, no seguinte protótipo:

```
object Store {
    fun get -> Int {
        return n;
    }
    fun put: Int pn {
        n = pn;
    }
    Int n
end
```

A diferença em relação à classe é que o protótipo é um objeto, ele pode então receber mensagens:

```
Store put: 0;
Out println: (Store get);
```

"`Store put: 0`" é o envio da mensagem "`put: 0`" ao objeto `Store`. "`put:`" é chamado de "seletor" da mensagem. Da mesma forma, "`Store get`" é o envio da mensage `get` ao objeto `Store`. O resultado é passado como parâmetro ao seletor `println:`.

Objetos podem ser criados em execução atráves do método `clone` que todos os objetos possuem:

```
var s = Store clone;
s put: 5;
Out println: (s get);
```

Então `clone` faz o papel de `new` em linguagens com classes. Em Cyan podem existir, para dado protótipo, métodos `new` ou `new:` para a criação de objetos.

A maioria das linguagens baseadas em protótipos é dinamicamente tipada e permite alterações dinâmicas nos protótipos e outros objetos em execução.[11] Então, por exemplo, pode-se adicionar uma variável de instância em execução. Ou um método. Pode-se também adicionar herança, remover herança, remover métodos e variáveis de instância, protótipos etc. Note que modificar um protótipo pode significar modificar todos os objetos criados a partir daquele protótipo. Ou todos os objetos criados a partir dele a partir da modificação. Modificar um objeto que não é protótipo altera apenas o objeto.

Métodos, protótipos, praticamente tudo são objetos. Então pode-se facilmente passar um método de um protótipo `A` como parâmetro para um método de um objeto `B` que irá adicioná-lo a `B`.

Há vários mecanismos para implementar herança neste tipo de linguagem. Tipicamente, um objeto possui uma variável de instância `parent` que referencia um objeto que desempenha o papel de "superclasse". Quando um objeto recebe uma mensagem, ele procura pelo método correspondente nele mesmo. Se não encontra, ele delega a mensagem para o objeto referenciado por `parent`. Mas para `parent` também é passada uma referência ao objeto original. Se há um envio de mensagem para `self` (ou `this`) no objeto referenciado por `parent`, a busca pelo método começa no objeto original. É um mecanismo praticamente igual ao de linguagens baseadas em classes.

Mudar a herança de um objeto é simplemente fazer a variável de instância `parent` referenciar outro objeto.

Há várias vantagens em linguagens baseadas em protótipos (LBP):

(a) elas tornam a programação mais concreta. O que o programa declara são realmente entidades existentes como uma pessoa, um elefante etc. Uma classe é mais abstrata pois é um molde utilizado para criar objetos. O molde e o objeto existem em tempos diferentes: um em compilação e outro em execução. Herança só é feita em compilação, classes não podem ser modificadas, objetos só adquirem vida em execução. As LBP eliminam esta distinção: objetos existem na compilação, herança pode ser feita em execução também;

(b) é fácil criar um novo protótipo em execução. Pode-se clonar um protótipo existente e adicionar um método a ele em seguida. Em programas que exigem uma grande quantidade de classes ou protótipos, isto é realmente útil;

(c) em algumas LBP, a proteção de informação existe até mesmo dentro do próprio objeto. Isto é, o acesso a variáveis de instância é feito atráves de métodos. Então pode-se facilmente modificar a representação do objeto sem alterar o próprio objeto;

(d) objetos únicos, como `Terra`, dos quais devem existir apenas um único exemplar são fáceis de serem criados (este é o padrão de projetos "Singleton");

(e) não há necessidade de meta-classes. Em linguagens baseadas em classes que consideram classes como objetos, as classes têm que ter uma classe, chamada de meta-classe. Mas esta meta-classe também é uma classe e portanto um objeto. E que tem que ter uma classe, que é uma meta-meta-classe. Há uma regressão potencialmente infinita que não é resolvida satisfatoriamente, no nosso ponto de vista, em nenhuma linguagem;

(f) o programa pode alterar a si mesmo em execução. Esta capacidade pode ser utilizada para fazer programas que seriam difíceis de fazer em linguagens sem esta característica.[12]

Há inúmeras críticas a linguagens baseadas em protótipo também:

[11] Aparentemente, em 2014, só existem duas linguagens que podem ser consideradas baseadas em protótipos e estaticamente tipadas: Cyan e Omega. Cuidado: há duas linguagens que se chamam Omega.

[12] Para saber mais, estude "programação adaptativa" ou "adaptive programming".

(a) humanos tendem a abstrair as entidades encontradas no mundo. Então a existência de classes é natural, pois estas são abstrações de entidades encontradas no domínio do sistema que está sendo implementado;

(b) a facilidade de alterar protótipos e objetos em execução torna os programas difíceis de entender. O código que se vê no monitor do computador não é aquele que será executado. O código executado depende do fluxo de execução do próprio código;

(c) a imensa maioria das LBP são dinamicamente tipadas e trazem consigo todas as desvantagens desta tipagem: detecção de erros de tipo somente em execução, desempenho ruim. Além disto, é difícil otimizar o código pois este pode ser alterado em execução.

Como exemplos de linguagens baseadas em protótipos podemos citar Self, Javascript, Cecil, Omega e Cyan.

3.9 Classes parametrizadas

A classe `Store` em Smalltalk permite o armazenamento de objetos de qualquer tipo.

```
class Store {
    private n;
    public put( i ) {
        n = i;
    }
    public get() { return n; }
}
```

Um objeto de `Store` guarda um outro objeto através do método `put` e retorna o objeto armazenado através de `get`.

A classe `Store` em Java é mostrada abaixo com o nome de `StoreInt`.

```
class StoreInt {
    private Int n;
    public void put( int i ) {
        n = i;
    }
    public int get() { return n; }
}
```

Como cada variável possui um tipo nesta linguagem, a classe `Store` se torna restrita — só pode armazenar inteiros. Se quisermos armazenar objetos de outros tipos, teremos que construir outras classes semelhantes a `Store` — uma classe para cada tipo, como `StoreBoolean`, `StoreFloat`, etc.

Em Smalltalk, a classe `Store` é utilizada para todos os tipos, causando reaproveitamento de código. Nesta linguagem, podemos ter uma árvore binária ou lista encadeada genérica, que permite armazenar objetos de qualquer tipo. Os métodos para a manipulação de cada uma destas estruturas de dados é construído uma única vez. Como não há conferência de tipos, é possível inserir objetos de diferentes classes na lista encadeada, criando uma lista heterogênia.

A linguagem Java possui uma construção que oferece um pouco da flexibilidade de Smalltalk, chamada de classes genéricas.

```
public class Store<T> {
    private T n;
    public void put( T i ) {
        n = i;
    }
    public T get() { return n; }
}
```

A classe `Store` acima é genérica e possui um tipo com parâmetro. Na declaração de uma variável desta classe, deve ser especificado o parâmetro `T`:

```
Store<Integer> si = new Store<Integer>();
si.put(0);
System.out.println(si.get());
```

A atribuição de parâmetros reais a uma classe genérica é chamada de instanciação da classe. Em Java, todas as instanciações compartilham o mesmo código. Isto é, há um único código para `Store<Integer>`, `Store<Person>` e assim por diante. Isto limita as operações que podem ser feitas com o tipo dentro da classe genérica. Por exemplo, dentro de `Store` não se pode criar objetos de `T` e não pode existir um cast para `T`. Além disso, exceções não podem ser genéricas, vetores de `Store<Integer>` não podem existir, o operador `instanceof`[13] não pode ser usado com classes genéricas.

Todas estas limitações são causadas porque o código da classe genérica não é duplicada para cada instanciação. Outras linguagens, como Cyan e C++, duplicam o código. Em Cyan não há limitações ao uso dos parâmetros genéricos, embora possa haver erros na compilação quando um parâmetro formal é substituído por um real. Por exemplo, considere o protótipo `Box` nesta linguagem:

```
object Box<T>
    T elem
    fun set: T elem  {
        self.elem = elem;
    }
    fun get -> T { return elem; }
    fun process {
        elem prettyPrint;
    }
end
```

Podemos criar uma instanciação `Box<Int>`. Contudo, haverá um erro de compilação, pois `elem` será do tipo `Int` e este tipo não possui um método `prettyPrint`. Uma mensagem `prettyPrint` é enviada a `elem` no último método.

Observe que, em Cyan, `Box` não é um protótipo, é apenas uma máscara (ou esqueleto — *template* em Inglês) para a criação de protótipos. Os exemplos (instâncias) construídos a partir de `Box`, como `Box<Int>`, são realmente protótipos. Eles podem ser usados como tipo de variáveis, em herança etc. Cada instanciação do protótipo com tipos diferentes causa a criação de um novo código fonte.

Classes ou protótipos genéricos de C++ e Cyan, com duplicação de código, possuem os problemas semelhantes a linguagens dinamicamente tipadas. Alterações na classe ou protótipo genérico pode invalidar instanciações que estavam funcionando adequadamente. Por exemplo, suponha que `Box`

[13] Em java, `x instanceof C` retorna `true` se o objeto referenciado por `x` é uma instância da classe `C`.

tenha sido criado sem o método `process`. Não há nenhum erro na instanciação `Box<Int>`. Contudo, ao acrescentar `process` estaríamos introduzindo um erro nesta instanciação. **O erro não seria detectado na compilação de `Box` e sim na instanciação `Box<Int>`**. Diferente das linguagens dinamicamente tipadas, o erro seria detectado em compilação.

3.10 Closures

Cyan permite a declaração de funções anônimas que podem ter parâmetros e retornar um valor:

```
var b = { (: Int n -> Int :)
    ^ n * n;
};
```

O tipo dos parâmetros e o tipo de retorno (opcional) são declarados entre `(:` e `:)`. Este tipo de função anônima é chamado de `função` ou função anônima em Cyan, que são objetos como todos os outros valores da linguagem.

O valor retornado por uma função é dado após "`^`". Ao executar a instrução acima, a função é atribuído a `b`. Mas a função não é executada, o que deve ser feito enviando-se a mensagem `eval:` à variável `b`:

```
var b = { (: Int n -> Int :)
    ^ n * n;
};
Out println: (b eval: 5);
```

É impresso `25` na saída padrão. Sendo uma função um objeto, `b` é uma instância de um protótipo e portanto tem um tipo. O tipo de `b` é

```
Function<Int, Int>
```

O último `Int` é o valor de retorno. Uma função

```
var concat = { (: Int n, Char ch -> String :)
    ^ (ch + n) asString;
};
```

tem o tipo

```
Function<Int, Char, String>
```

Não se preocupe com a instrução de dentro da função. Ela soma um caráter a um número, resultando em um caráter, e converte para `String`.

Funções podem acessar variáveis locais e de instância:

```
int other = 1;
var b = { (: Int n -> Int :)
    ^ n + other;
};
Out println: (b eval: 0);
other = 2;
Out println: (b eval: 0);
```

O valor da variável `other` utilizado é aquela do momento da avaliação da função, quando o método `eval:` é chamado. Então são impressos os números `1` e `2`. Uma função também pode modificar o valor das variáveis externas que ele acessa:

```
int sum = 0;
var b = { (: Int n :) sum = sum + n; };
b eval: 1;
b eval: 2;
Out println: sum;
```

Será impresso "3".

Vetores em Cyan possuem um método chamado `foreach:` que aceita uma função como parâmetro. Este método chama a função para cada elemento do vetor. Então a função deve aceitar um tipo igual ao do elemento do vetor:

```
int sum = 0;
var Array<Int> v = {# 1, 2, 3, 4, 5 #};
v foreach: { (: Int n :)
    sum = sum + n;
};
Out println: sum;
```

Será impresso "15". Um vetor literal em Cyan é dado entre `{#` e `#}`. Em execução, quando a função acima é criada, é feita uma ligação da variável livre `sum`, que não foi declarada na função, e a variável local `sum` — elas passam a ser a mesma variável. Isto é, a função *close over* suas variáveis livres. É interessante notar que esta notação vem da Lógica e, em particular, do Cálculo Lambda. Em Lógica, as fórmulas $\forall x\,(f(x) = y)$ e $f(x) = y$ têm uma e duas variáveis livres: y no primeiro caso e x e y no segundo. Quando se adicionam os quantificadores, $\exists y\,\forall x\,(f(x) = y)$ e $\forall x\,\forall y\,f(x) = y$, as variáveis passam a ser ligadas (ao quantificador) e as fórmulas são chamadas de fechadas (closed).

Uma *closure* é um bloco de código, possivelmente com variáveis *livres*, no qual estas variáveis foram ligadas a variáveis locais, parâmetros ou variáveis de instância em execução. Então uma *closure* é um objeto que existe em tempo de execução, não é a sequência de instruções que está no código. *Closures* são criadas a partir de funções anônimas (como Cyan) ou não. Pode-se ter funções aninhadas em que a função interior acessa variáveis locais declaradas nas funções mais externas.

Veremos outros exemplos de funções.

```
   // imprime os elementos do vetor
var Array<Int> v = {# 1, 2, 3, 4, 5 #};
v foreach: { (: Int n :)  Out println: n };

   // imprime os número de 1 a 10
1..10 foreach: { (: Int n :)  Out println: n };

   // imprime "cinco vezes" cinco vezes
5 repeat: {
    "cinco vezes"  println;
};
( age < 3 ) ifTrue:  { "baby" println }
            ifFalse: { "non-baby" println };
```

`1..10` é um intervalo em Cyan, que possui um método `foreach:`. O protótipo `Int` possui um método `repeat:` que toma uma função como parâmetro. A última instrução é um `if` implementado com envio de mensagens. Duas funções são passados como parâmetro. Em Smalltalk, não há comando `if` ou `while`. Todas as estruturas de repetição são implementadas com envio de mensagens.

3.11 Meta-programação

Meta-programação é programação sobre programas, o que pode ter inúmeros sentidos: programas que manipulam outros programas, um programa que examina a si mesmo, um program que modifica a si mesmo e criação de código em tempo de compilação.

3.11.1 Reflexão Introspectiva

O mais simples tipo de meta-programação acontece quando um programa examina a si mesmo, o que é chamado de *reflexão introspectiva*. Por exemplo, em Java pode-se listar os métodos e variáveis de instância de uma classe:

```
package main;

import java.lang.reflect.Method;

public class Reflect {

    public void r() {
        Class<?> c = this.getClass();
        Method m[] = c.getDeclaredMethods();
        for (int i = 0; i < m.length; i++)
            System.out.println(m[i].toString());
    }

    public int fat(int n) { return n <= 0 ? 1 : n*fat(n-1); }
    public String asString() { return "Reflect"; }
}
```

A chamada `this.getClass()` retorna um objeto da classe `Class` que descreve a classe de this, que neste exemplo é `Reflect` (pois não há subclasses). Usando este objeto, o envio de mensagem

```
c.getDeclaredMethods()
```

retorna objetos descrevendo os métodos de `Reflect`.

Ao se chamar `r`, como em "`(new Reflect()).r()`", será impresso

```
public void main.Reflect.r()
public int main.Reflect.fat(int)
public java.lang.String main.Reflect.asString()
```

Pode-se invocar um método pelo seu nome, como no código

```
Reflect reflect = new Reflect();
Method m = reflect.getClass().getMethod("asString");
System.out.println( (String) m.invoke(reflect) );
```

O método `getMethod` retorna um objeto que descreve o método cujo nome é o parâmetro. Pode-se fornecer os tipos dos parâmetros. Neste caso, isto não é necessário pois `asString` não possui parâmetros. Neste exemplo, o método `invoke` de `Method` chama o método do objeto `reflect` cujo nome é `asString`.

Em Cyan, pode-se chamar um método cujo nome está em uma string usando-se o operador ‘ (*backquote*, ASCII 96):

```
var String s = "name";
    // cria um objeto Person
var Person p = Person("Newton", 85);
   // envia a p a mensagem 'name'
Out println: ( p `s );
s = "age";
   // envia a p a mensagem 'age'
Out println: ( p `s );
```

3.11.2 Reflexão Comportamental

Linguagens que suportam reflexão comportamental permitem alterar o próprio código do programa durante a execução. Smalltalk, Self, Ruby, Groovy e Cyan são linguagens que suportam este tipo de reflexão, embora os limites do que se possa fazer varie de linguagem a linguagem. Usualmente, pode-se inserir e remover variáveis de instância e métodos em classes, mudar a herança, criar novas classes, eliminar classes etc. As operações de remoção são altamente inseguras e podem causar erros de execução mais facilmente do que as operações de inserção.

Como exemplo de alteração de uma classe, em Groovy pode-se adicionar um método `swapCase` à classe `String` pelo seguinte código, tomado de `http://groovy.codehaus.org/ExpandoMetaClass`.

```
String.metaClass.swapCase = {->
      def sb = new StringBuffer()
      delegate.each {
           sb << (Character.isUpperCase(it as char) ?
                   Character.toLowerCase(it as char) :
                   Character.toUpperCase(it as char))
      }
      sb.toString()
}
```

Depois de executado, temos que `"aBc".swapCase()` retorna `"AbC"`.

Em Cyan, pode-se adicionar ou trocar um método em um protótipo ou objeto através do método `addMethod: ...` definido no super-protótipo `Any` que é herdado por todos os outros. Este é um tipo especial de método chamado de "método de gramática" em que os seletores podem ser descritos utilizando expressões regulares. Para saber mais, consulte o manual da linguagem.

```
public fun (addMethod:
             (selector: String ( param: (Any)+ )?
               )+
              (returnType: Any)?
              body: ContextObject) t
```

Um método `toString` pode ser adicionado a um protótipo `Person` da seguinte forma:

```
Person addMethod:
       selector: #toString returnType: String
       body: { (: Person self :)
           ^ name + " (" + age + ")";
       };
```

A função passada como parâmetro ao seletor `body:` chama-se "função de contexto" e tem `self` como primeiro parâmetro. Dentro desta função as mensagens enviadas a `self`, aquelas sem receptor especificado, devem corresponder a métodos do tipo de `self`, que é `Person`. O código acima está correto se `Person` possui métodos `name` e `age` que retornam objetos do protótipo `String` e `Int`. `+` é a concatenação de uma string com qualquer outro objeto (como em Java).

```
object Person
    fun name -> String { return _name; }
    fun name: (String newName) { _name = newName; }
    fun age -> Int { return _age; }
    fun age: (Int newAge) { _age = newAge; }
    ...
    String _name
    Int _age
end
```

Após a execução do método `addMethod:` pela instrução acima, pode-se enviar a mensagem `toString` para `Person` ou seus objetos:

```
Person name: "Newton";
Person age: 45;
var String s = Person toString;
```

Mas ... o último envio de mensagem, `Person toString` está incorreto se assumirmos que `Person` não tinha um método `toString`. O compilador procurará por um método `toString` em `Person` e não o encontrará, resultando em um erro de compilação. É necessário colocar `?` antes do nome do método. Isto fará com que o compilador não confira se o método existe ou não, exatamente como em linguagens tipadas dinamicamente. O tipo de retorno do método será considerado como `Any`, o super-protótipo de todo mundo.

```
Person name: "Newton";
Person age: 45;
var String s = Person ?toString;
```

O compilador considera que qualquer valor retornado por um método chamado através de `?` é compatível com qualquer tipo. Então o compilador considera que `Person ?toString` retorna uma `String`. Naturalmente, é inserido um teste para, em execução, conferir se este valor pode mesmo ser convertido para `String`. Então esta última linha é equivalente a

```
var String s = String cast: (Person ?toString);
```

Em Cyan pode-se adicionar variáveis de instância a qualquer protótipo que herde o protótipo mixin `AddFieldDynamicallyMixin`. Qualquer protótipo que herde deste mixin pode introduzir variáveis através do envio de mensagens prefixadas por `?`. Por exemplo, considere um protótipo `Pessoa` que não possui nenhum método ou variável de instância, mas que herda de `AddFieldDynamicallyMixin`.

```
object Person mixin AddFieldDynamicallyMixin
end
```

Pode-se adicionar uma variável de instância enviando-se uma mensagem a `Person`:

```
Person ?name: "Newton";
Out println: (Person ?name);
```

Este envio de mensagem cria dois métodos: `name: String` e `name -> String`. O primeiro método inicializa uma variável de instância (que é guardada em uma tabela hash) e o segundo retorna o valor da variável.

O leitor interessado deve procurar uma outra forma de reflexão comportamental chamada de metaobjetos de tempo de execução. Um metaobjeto é um objeto que pode ser acoplado ao outro objeto em execução. Todas as mensagens enviadas ao objeto são redirecionadas ao metaobjeto. Este pode fazer algum processamento e enviar a mensagem original ao objeto. Este tipo de comportamento é implementado por mixins em Groovy e Ruby.

3.11.3 Metaobjetos de Tempo de Compilação

Um metaobjeto de tempo de compilação é um objeto cujos métodos são executados em compilação. O compilador inclui o metaobjeto ao seu próprio código. Métodos do metaobjeto podem modificar como a compilação é feita. Eles podem fazer conferências adicionais, inserir métodos e variáveis de instância em classes e protótipos, modificar métodos existentes etc. Exemplificaremos este conceito usando a linguagem Cyan. Não serão dados detalhes de como os metaobjetos são implementados, apenas detalhes de alto nível em relação à implementação.

Metaobjetos em Cyan são *chamados* em tempo de compilação usando-se `@` como em

```
package people;

object Person
    @init(name, age)
    String name;
    Int age;
end
```

O Metaobjeto `init` é chamado com os parâmetros `name` e `age`. Metaobjetos são declarados em *packages*, mais especificamente, no diretório `meta` de um *package*. Em Cyan, eles são feitos atualmente em Java. O *package* `cyan.lang` é incluído automaticamente por qualquer arquivo fonte em Cyan e, junto com ele, são incluídos os metaobjetos padrões como `init` (usado acima), `prototypeCallOnly`, `annot`, `doc`, `text`, `checkIsA` etc. Para exemplificar, no diretório `cyan\lang\meta` há um arquivo `CyanMetaobjectInit.java` que implementa o metaobjeto `init`. O compilador carrega esta classe e cria um objeto dela, que chamaremos de `metaInit`.

Quando o compilador Cyan encontra `@init`, ele solicita ao metaobjeto `metaInit` que gere o código que deve substituir `@init(name, age)`. Isto é, a mensagem `cyanCode(...)` é enviada a `metaInit`. Obviamente, os parâmetros são passados ao método `cyanCode` como strings. Este método retorna uma string que substitui `@init(name, age)`. O código resultante fica:

```
package people;

object Person
    fun init: String name, Int age {
        self.name = name;
        self.age = age;
    }
    String name;
    Int age;
end
```

Mas como o método `cyanCode` de `CyanMetaobjectInit.java` sabe que `name` é do tipo `String` e `age` é do tipo `Int`? Estas informações são passadas a `cyanCode` através de um objeto `context`:

```
metaInit.cyanCode(parameters, context)
```

Este objeto possui muitas das informações que o compilador possui. Através dele pode-se saber as variáveis de instância do protótipo, os seus métodos, de quem o protótipo herda etc. Como um outro exemplo, suponha a existência de um metaobjeto `beforeAfter` que modifica métodos:

```
package zoo;

object Animal
    @beforeAfter fun print {
        name println;
    }
    ...
end
```

Este metaobjeto poderia introduzir código no método ao qual ele está acoplado. O resultado poderia ser:

```
package zoo;

object Animal
    fun print {
        "before print" println;
        name println;
        "after print" println;
    }
    ...
end
```

Então o metaobjeto `beforeAfter` modifica o próprio método. Isto é feito modificando-se a Árvore de Sintaxe Abstrata (ASA) do método por um método do metaobjeto. A ASA de um método é a representação em forma de objetos do texto do método. Assim, um método é representado por um objeto da classe `MethodDec` (em Java) que possui variáveis de instância para representar os seletores dos métodos, os parâmetros (nome e tipo), tipo de retorno, se é público, privado ou protegido e suas instruções. Um dos métodos do metaobjeto pode modificar ou conferir qualquer aspecto deste objeto. Poderia, por exemplo, mudar o tipo de retorno ou conferir se os seletores estão começando por letra minúscula.

O metaobjeto `prototypeCallOnly` deve ser acoplado a um método. Ele confere se o método foi chamado através de um protótipo. Se não foi, ele sinaliza um erro.

```
package data;

object Date
    @prototypeCallOnly
    fun getCurrentTime -> Long { ... }
    ...
end
```

Pode-se usar

```
ct = Date getCurrentTime;
```

Mas a instrução seguinte causa um erro de compilação

```
ct = (Date new) getCurrentTime;
```

Este metaobjeto declara um método `checkMessageSend` que é chamado pelo compilador em todos os envios de mensagem em que este método possivelmente seria chamado. O compilador passa como parâmetros o receptor da mensagem e os parâmetros. E o método confere se o receptor é um protótipo. Se não for, pede ao compilador para sinalizer um erro.

Poder-se-ia definir um metaobjeto `memoized` (como em Groovy) para memorizar o resultado de um método que realiza alguma computação cara em termos de tempo. Este metaobjeto iria inserir no protótipo uma tabela hash para guardar os resultados já obtidos e modificar o método de tal forma que ele verifique na tabela se a computação com o valor pedido (parâmetro) já não foi realizada antes.

```
package myMath;

object MyMath
    @memoized
    fun fibonacci: (Int n) -> Int { ... }
    ...
end
```

3.11.4 Macros

Macros são funções que são executadas em tempo de compilação e que produzem código que é então compilado.

Para exemplificar este tópico, mostraremos como macros serão definidos em Cyan.[14] Nesta linguagem, um macro pode ter mais de um "seletor", cada um deles com parâmetros. Os seletores não são terminados por ":". Pode-se declarar uma função macro usando-se a palavra-chave `macro` antes de "`fun`":

```
macro fun "unless" (String expr) "do" (String b)  -> String
          where Expr expr, expr getType  == Boolean
          where Function<Nil> b
{
   return "if ! #{expr} #{b}";
}
```

Este macro acrescenta à linguagem Cyan duas novas palavras reservadas: `unless` e `do`. É como se estas duas palavras fossem seletores deste método, cada um tomando uma `String` como parâmetro. O tipo de retorno também é `String`. Assuma que sempre será assim: os tipos dos parâmetros e do retorno serão `Strings`. A cláusula `where` na segunda linha restringe o que pode ser o primeiro parâmetro: `expr` pode ser um objeto do tipo `Expr` e o tipo de `expr` deve ser booleano. A próxima cláusula restring `b` para um objeto de `Function<Nil>`.

`Expr`, `Boolean` e `Function` são classes da Árvore de Sintaxe Abstrata (AST) do compilador Cyan. Estas classes são utilizadas para representar o programa Cyan. `Expr` representa uma expressão. Esta classe possui um método `getType` que retorna o tipo da expressão. Então "`expr getType`" é o envio da mensage `getType` ao objeto referenciado por `expr`.

[14] Este conceito ainda não faz parte da linguagem.

Em Cyan, se existe uma variável `n`, então `"valor = #{n}"` é o mesmo que
`("valor = " + n)`
Isto é utilizado no código acima.

Quando o compilador encontra um uso de `unless`, como em

```
unless n >= 0 do {
    Out println: "Please type an integer >= 0";
};
```

ele procura por um macro que comece por "`unless`". De fato, ele faz outras buscas que não nos interessam antes disto.

O macro declarado anteriormente é encontrado. O compilador confere se que o que está entre `unless` e `do` é uma expressão e se esta expressão é do tipo `Boolean` (cláusula `where` do macro). Após isto é conferido se depois de "`do`" se segue uma função sem parâmetros ou valor de retorno. Nenhuma das conferências resulta em erro. Então o compilador chama a função macro passando como parâmetros as seguintes strings:

```
"n >= 0"
"{\n    Out println: "Please type an integer >= 0";\n}"
```

O macro retorna uma string

```
"if  ! (n >= 0) {\n    Out println: "Please type an integer >= 0";\n}"
```

que o compilador coloca no lugar de

```
unless n >= 0 do {
    Out println: "Please type an integer >= 0";
}
```

Isto é, a "chamada do macro" é apagada do código e o retorno do método macro é inserido. Note que esta mudança no código afeta apenas os dados internos do compilador. O código fonte do usuário não é afetado.

Para entender melhor como macros funcionam, considere o macro

```
macro fun "test"
{
   Out println: "Em compilação";
   return "Out println: \"Em execução\"";
}
```

Em um programa Cyan

```
package main
object Main
    fun run {
        test;
    }
end
```

O compilador iria imprimir `"Em compilação"` na saída padrão (é o que faz o método `println:` de `Out`). Ao executar o método `run`, seria impresso

```
"Em execução"
```

3.12 Linguagens Específicas de Domínio

Uma linguagem específica de domínio[15] (LED) é uma linguagem adequada para um domínio específico. Uma LED pode ser uma linguagem de programação (o mais usual) ou não. No último caso, ela pode apenas descrever dados ou não ter todos os comandos necessários para que seja Turing-completa.

Existe uma enorme quantidade de LED´s disponíveis. Mostraremos algumas delas.

(a) JSON, Javascript Object Notation (json.org). É uma linguagem para descrever dados que torna estes fáceis de visualizar. É uma LED, mas não uma linguagem de programação. O exemplo abaixo foi obtido de `http://json.org/example.html`.

```
{
  "menu": {
      "id": "file",
      "value": "File",
      "popup": {
          "menuitem": [
            {"value": "New", "onclick": "CreateNewDoc()"},
            {"value": "Open", "onclick": "OpenDoc()"},
            {"value": "Close", "onclick": "CloseDoc()"}
          ]
      }
  }
}
```

(b) SQL, para gerenciar dados em banco de dados.

```
select *
    from Estudante
    where nota >= 6
    order by nota;
```

SQL é uma linguagem de programação.

(c) linguagens regulares nas quais um conjunto (linguagem) é descrita suscintamente. Esta linguagem não é Turing-completa.

```
[A-Za-z][A-Za-z0-9_]+
1+(a|b)*cc+
```

(d) HTML, XML, que não são Turing-completas. São apenas descritivas;

(e) a linguagem utilizada no programa `make` do Unix;

(f) TEXe LATEXpara processamento de texto;

(g) linguagens de programas que geram compiladores como YACC, Bison, ANTLR etc;

[15] Em Inglês, DSL, Domain Specific Language.

(h) uma linguagem que controla uma máquina industrial, uma linguagem para controlar um carro de brinquedo, para programar um jogo de computador, para criar uma interface gráfica, para representar os movimentos das peças em um jogo de Xadrez etc.

Antes de continuar, apresentaremos uma LED para movimentar um carro de brinquedo programável. Esta linguagem é implementada em Java.

```
class Car {
    public Car start() { ... return this; }
    public Car forward(int cm) { ... return this; }
    public Car left(int degree) { ... return this; }
    public Car right(int degree) { ... return this; }
    public Car stop() { ... return this; }
    ...
}
```

Usualmente estes métodos retornariam `void`. Aqui eles retornam o objeto que recebeu a mensagem, `this`. Isto permite que envios de mensagem possam ser encadeados:

```
Car c = new Car();
c .start()
  .forward(10)
  .left(50)
  .stop();
```

Este encadeamento torna a aparência do código menos parecido com Java e mais com uma LED. O código fica mais natural.

Existem dois tipos de LEDs, as internas e externas. As internas são aquelas implementadas dentro de uma linguagem de programação, usando a sintaxe usual da linguagem (como a LED acima). As externas exigem que se faça um compilador específico para elas e não precisam de uma linguagem de programação hospedeira. As LEDs externas utilizam uma gramática qualquer, não há restrições quanto à sintaxe ou semântica como a LED interna.

Diversas linguagens dão suporte à construção de LED´s internas, como Cyan, Groovy, Ruby e Scala. Este suporte se dá através de diversas características e construções destas linguagens:

(a) suporte a *closures*. Com o uso destas pode-se criar comandos que tomam blocos de código como parâmetros. Uma pequena LED pode ser utilizada para trabalhar com arquivos em Cyan:

```
object MyFile
    fun open: (String name) eachLine: Block<String> b close: {
    }
    ...
end

MyFile open: "data.txt"  eachLine: { (: String line :)
    if ( line contains: "//" )
        line println; // print only lines with comments
}
close: ;
```

(b) tipagem dinâmica também ajuda na construção de LED´s. Há um item a menos a ser fornecido, o tipo, o que torna as LED´s mais fáceis de programar. Este fato se torna mais importante quando os usuários da LED são leigos em Computação. Por exemplo, pode-se ter uma LED para a prescrição de medicamentos ou para implementar regras de negócios. Em ambos os casos o usuário pode não ser da área de Computação;

(c) listas, tuplas, tabelas hash, vetores e intervalos literais. Estes objetos literais facilitam a criação de estruturas complexas de dados. Veja o exemplo abaixo em Cyan.

```
  // uma tupla literal
var t = [. "Isaac Newton", 45 .];
    // 't f2' retorna  45
Out println: "age is " + (t f2);
    // vetor literal
var Array<Int> v = {# 1, 2, 3, 4, 5 #};
    // uma tabela Hash hipotética
var myHash = [*  "Newton" : 1642, "Leibniz" : 1646, "Neumann" : 1903 *];
    // imprime 1903
Out println: (myHash get: "Neumann");
'a'..'z' foreach: { (: Char ch :) ch println };
```

(d) o uso opcional de parenteses e ';' torna as DSL´s mais legíveis:

```
Car c = new Car
c .start
  .foward 10
  .left   50
  .stop
```

(e) inserção de métodos em tipos básicos como `int` ou simulação desta inserção. Isto permite códigos como

```
int n = howMuchTime();
int numDays = 5.days;
int totalTime = 4.hours + n.hours;
```

Em Ruby, novos métodos podem ser inseridos em tipos básicos. Em Groovy pode-se usar Categorias para simular esta inserção ou realmente inseri-los nos tipos básicos;

(f) métodos com nomes de operadores como `+`, `<<` etc.

```
Matriz m, a, b;
...
m = a * b;
Date d = new Date(01, 04, 2014);
d = d + 5; // d mais cinco dias
```

(g) uso de símbolos ao invés de strings. Símbolos em Smalltalk e Cyan são strings especiais iniciadas por `#`

```
house color: #cyan;
car setColor: #red;
Out println: #first, #second;
```

(h) algumas linguagens criam automaticamente uma tabela hash a partir dos parâmetros. Como exemplo, em Groovy[16]

```
take 1.pill
   of: Chloroquinine,
   after: 6.hours
```

Foi definido um método

```
take(Map m, MedicineQuantity mq)
```

`1.pill` é o valor de `mq`. Já "`of: Chloroquinine`" e "`after: 6.hours`" são agrupados em um `map` (tabela hash) e passados como o primeiro parâmetro.

LED´s possuem inúmeras vantagens sobre linguagens de propósito geral (LPG):

(a) o código é mais legível, mais fácil de manter e mais confiável do que código de uma LPG;

(b) o número de linhas de código é muito menor do que o número de linhas necessárias em uma LPG;

(c) pode ser compreendida por não especialistas (se for projetada com esta finalidade).

E algumas desvantagens também:

(a) há uma curva de aprendizado que não existe se uma LPG for utilizada;

(b) a LED precisa ser projetada e implementada;

(c) a LED pode ser mais ineficiente do que o código implementado em uma LPG. Mas em alguns casos pode o código pode até ter execução mais rápida;

(d) pode ser difícil ou impossível integrar código LED com o código normal de uma aplicação;

(e) frequentemente precisa crescer com o acréscimo de novas funcionalidades;

(f) pode não contar com suporte de uma IDE (mas pode também ter este suporte — veja Eclipse XText).

As LEDs internas possuem também vantagens e desvantagens quando comparadas com as LEDs externas. As vantagens de uma são as desvantagens de outra.

Vantagens de LEDs internas sobre as externas:

(a) não necessita de um compilador, sendo portanto mais fácil de fazer do que as LED´s externas;

(b) todas as construções disponíveis para a linguagem estão também disponíveis para a LED;

(c) pode-se utilizar todas as bibliotecas disponíveis para a linguagem;

[16] Exemplo tomado de GR8Conf 2009: Practical Groovy DSL por Guillaume Laforge.

(d) a curva de aprendizado é menor do que a LED externa, pois ela utiliza os recursos já conhecidos pelo programador;

(e) permite o auxílio do IDE pois este pode ajudar a completar o código enquanto o usuário digita (por exemplo).

Vantagens de LEDs externas sobre as internas:

(a) pode utilizar uma gramática qualquer, não relacionada a qualquer linguagem de programação;

(b) pode-se ter auxílio da IDE para a LED externa. Mas frequentemente isto envolve um trabalho extra do criador da LED (ele/ela deveria implementar um plugin para a IDE). Há ferramentas como o XText que já fornecem o suporte à LED automaticamente. Então uma LED externa feita com o XText (ou outras ferramentas semelhantes) já conta com suporte pelo IDE;

(c) bibliotecas externas podem ser utilizadas, mas a LED deve ter construções que permitam a importação e sintaxe que permita o uso destas bibliotecas.

3.13 Discussão Sobre Orientação a Objetos

Dentre todos os paradigmas de linguagens, o orientado a objetos é o que possui melhores mecanismos para representação do mundo real em programas. Os elementos da realidade e as estruturas de dados são representados claramente no programa por meio de classes. Elementos como Pessoa, Governo, Empresa, Balanço de Pagamentos, Texto, Janela, Ícone, Relógio, Carro, Trabalhador, Pilha, Lista, e Fila são representados diretamente por meio de classes. O mapeamento claro entre o mundo real e programas torna mais fácil a compreensão e a manutenção do código. Não só os programas espelham o mundo como é relativamente fácil descobrir o que deve ser modificado no código quando há alguma alteração no mundo real.

Herança permite reaproveitar elegantemente código de superclasses. Uma subclasse define apenas os métodos que devem ser diferentes da superclasse. Hierarquias de herança são criadas incrementalmente com o tempo. As novas subclasses acrescentam funcionalidades ao código existente exigindo poucas ou nenhuma modificação deste.

Polimorfismo é o motivo da alta taxa de reaproveitamento de código encontrada em sistemas orientados a objeto. Código existente pode passar a trabalhar com subclasses sem necessidade de nenhuma alteração. Proteção de informação, estimulada ou mesmo requerida por muitas linguagens orientadas a objeto, impede que modificações nas estruturas de dados de uma classe invalidem outras classes. Este conceito é fundamental para a construção de sistemas, aumentando substancialmente a sua manutenabilidade.

3.14 Exercícios

34. Defina objeto. É um objeto um valor? Isto é, ele se assemelha mais ao valor 5 que está um `i` após a instrução

```
i = 5;
```

ser executada ou ele se assemelha mais ao:

- tipo `int`;
- variável `i`;