PROGRAMAÇÃO COM OBJECTOS

O.1 - INTRODUÇÃO

A programação orientada aos objectos goza das seguintes características:

Programação com objectos

Objecto = dados + operações

Um objecto tem estado interno não volátil e é capaz de responder a perguntas e efectuar operações sobre esse estado.

Programação é simulação

Os programas não são mais do que simulações de problemas a resolver.

Os objectos modelam entidades do mundo real.

Programação por mensagens

Cada objecto é que decide como responder quando recebe uma mensagem.

A mesma mensagem enviada para objectos distintos, pode originar respostas diferentes.

Programação por reutilização

Um objecto pode ser definido à custa de outro, isto é, o objecto novo herda as características (dados + operações) de outro já existente, que podem ser alteradas ou acrescentadas.

O.2 – PORQUÊ OBJECTOS?

Modelo natural

- As pessoas comunicam por mensagens, ninguém mexe no cérebro do parceiro.
- Só o receptor decide o que fazer quando recebe uma mensagem.
- Ninguém pode saber o que pensa outra pessoa nem o que fará quando receber uma dada mensagem

As entidades do mundo real comportam-se como objectos.

Na realidade, a programação orientada por objectos surgiu em simulação (SIMULA).

O modelo orientado por objectos facilita:

- A especificação do problema a resolver na linguagem de programação.
- A introdução de alterações e melhoramentos.

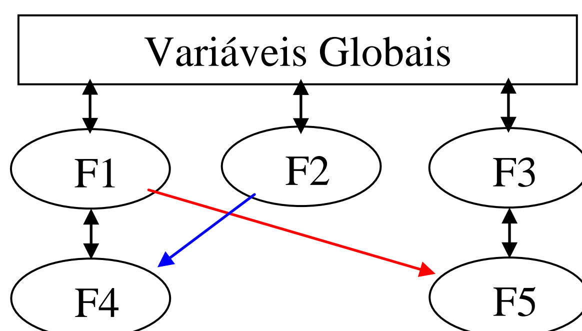
Isto porque o modelo de programação está mais perto da realidade.

O.3 – TIPOS DE PROGRAMAÇÃO

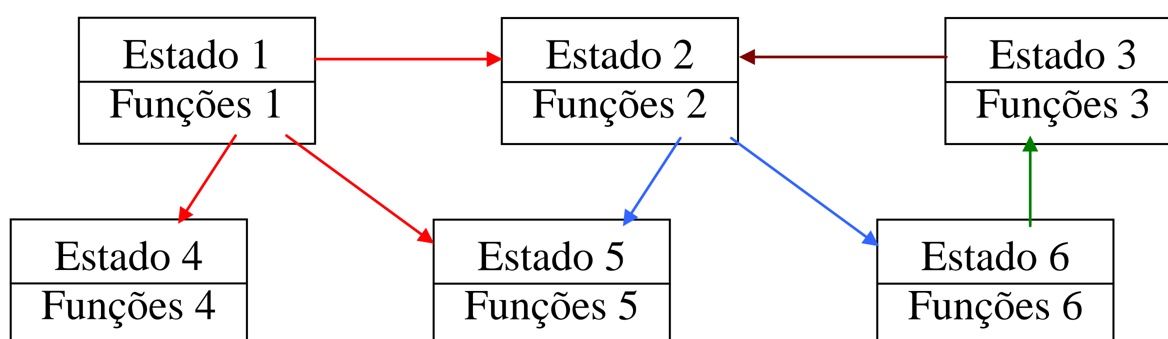
Linguagens	Organização	Base	Programação
Assembly FORTRAN 4	Código global Dados globais	Instrução	Não estruturada (70's e antes)
Pascal C	Código estruturado Dados globais	Procedimento Função	Estruturada (80)
Smalltalk-80 C++ Eiffel	Código + dados encapsulados	Objectos	Orientada para objectos (90's)

O.4 – PERSPECTIVA TRADICIONAL VS OBJECTOS

Perspectiva tradicional: programação orientada para funções



Perspectiva orientada para objectos



O.5 – CONCEITOS BÁSICOS

Objectos = Dados + operações.

Classe – Descrição de objectos com características comuns.

Instância – Objecto descrito por uma classe. As instâncias duma classe têm todas as mesmas operações e a mesma estrutura de dados, mas valores próprios diferentes.

Abstracção – Descrição formal do comportamento de uma dada entidade. Uma classe concretiza uma dada abstracção com um dado código (nas operações).

Encapsulamento – Os dados de um objecto só podem ser acedidos pelas suas operações.

Método – Operação.

Polimorfismo – Mecanismo pelo qual uma variável pode conter instâncias das subclasses da classe com que a variável foi declarada. Neste caso, os métodos invocados são os das subclasses e não os da classe da variável.

Mensagem = Selector + argumentos.

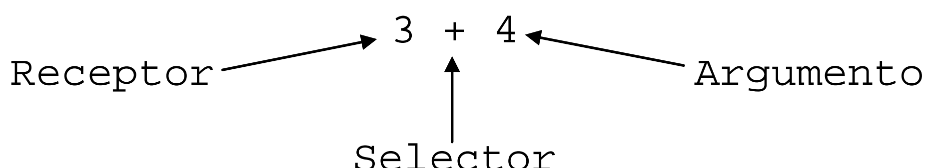
Selector – Especifica qual o método a invocar no objecto receptor

Receptor – Objecto que recebe uma mensagem.

A recepção de uma mensagem invoca um dado método.

Os objectos interagem sempre por mensagens.

Exemplo:



A diferença em relação à perspectiva tradicional é que quem comanda a operação é o receptor (3) e não o "+" (que é selector e não operador).

É sempre o receptor que decide o que fazer. A mesma mensagem enviada para objectos diferentes origina

normalmente reacções (métodos invocados) diferentes.

Exemplo:

```
Inteiro  ──→
Matriz   ──→  a + 3
String   ──→
```

Exemplo: Definição da classe ContaBancária

saldo	estado (variáveis)
depositar: valor saldo <- saldo + valor **levantar: valor** saldo <- saldo - valor **saldo** ^ saldo	comportamento (métodos)

Criação de uma conta (instância da classe ContaBancária):

```
minhaConta <- ContaBancária new
```

Uso da minha conta:

```
minhaConta depositar: 1000
minhaConta levantar: 500
saldoCorrente <- minhaConta saldo
```

O.5.1 - Objectos

Um objecto tem:

- Estado (variáveis)
- Comportamento (métodos)
- Identidade única

Os objectos conhecem a identidade de outros e o seu comportamento.

Os objectos interagem por mensagens.

Um objecto corresponde muitas vezes a identidades do mundo real, mas também pode representar entidades abstractas (Ex.: o número 3).

O.5.2 – Classes

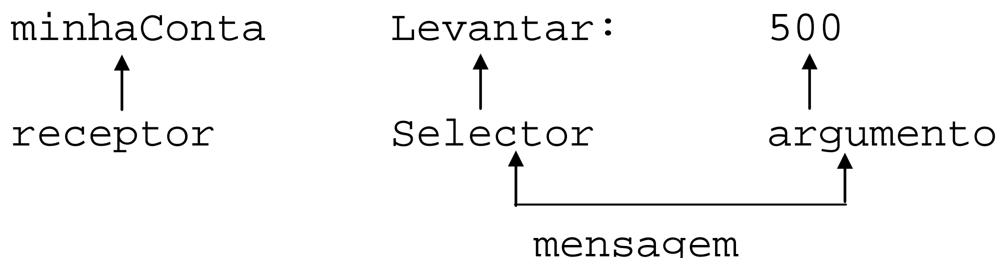
A classe descreve a **estrutura comum** a vários objectos, mas cada instância tem o seu próprio estado.

A classe é o que o programador especifica, e define a interface e a implementação dos objectos.

A classe é a unidade de modularidade na programação orientada por objectos.

Um objecto não é uma classe (mas em algumas linguagens uma classe é um objecto).

O.5.3 – Mensagens



Mensagem = selector + argumentos

Pedido de activação de comportamento (no receptor, a recepção de uma mensagem invoca o método correspondente ao selector da mensagem).

Envio de mensagem = chamada de função com escolha automática no receptor.

A interface de um objecto (o seu protocolo) é definida em termos de mensagens (aquelas a que ele sabe responder).

O.5.4 – Métodos

```
saldo
depositar: valor
  saldo <- saldo + valor

levantar: valor
  saldo <- saldo - valor

saldo
  ^ saldo
```

Método = código que implementa comportamento.

Um método equivale a um procedimento ou função.
Cada método inclui, usualmente, várias instruções de envio de mensagens.

Algumas instruções são normalmente primitivas e fazem parte integrante da sintaxe da linguagem (Ex.: atribuição).

O.5.5 – Abstracção

Abstracção – Forma de descrever algo pelas suas **propriedades "essenciais"**, ignorando o detalhe desnecessário.

É um conceito fundamental, não exclusivo do mundo dos objectos.

É uma forma usual de lidar com a complexidade.

0.5.6 - Encapsulamento

Embalagem de estado + comportamento

Estado acessível apenas através de operações (invocadas por mensagens).
Interface externa do objecto separada da sua implementação.

- Pode mudar-se a implementação sem alterar a interface.

- Ajuda a reduzir o efeito das alterações no programa.

- Quem usa o objecto não precisa de conhecer os **detalhes de implementação**.

0.6 - BENEFÍCIOS DA PROGRAMAÇÃO ORIENTADA AOS OBJECTOS

Produtividade - Construir software é mais fácil e barato.

Robustez - O software é mais fiável (por reutilização e melhores regras.

Manutenção - Muito mais fácil fazer melhorias e encontrar bugs.

Extensibilidade - Muito mais fácil acomodar novos requisitos.

Modularidade - Abstracção e encobrimento de informação.

Naturalidade - Programação orientada por abstracções do mundo real.

0.7 - DIFICULDADES NA UTILIZAÇÃO DA PROGRAMAÇÃO ORIENTADA AOS OBJECTOS

Linguagens diferentes usam conceitos e terminologias não totalmente coincidentes.

Não há um modelo semântico de objectos standard e universal.

Não há metodologias de análise e projecto universalmente aceites.

Técnicas convencionais de análise estruturada não são aplicáveis.

Ainda é uma área de investigação activa.

Formação

Linguagens POO são conceptualmente diferentes das tradicionais.

É preciso desaprender alguma coisas.

Ler sobre o assunto não basta, é essencial programar.

Tempo de aprendizagem não é zero.

O.8 - O QUE É QUE INTERESSA?

Embora haja muitas linguagens de POO e muitas variantes, existe um conjunto de conceitos comum que lhes serve de base.

O importante é:

Dominar bem o conjunto de conceitos base.

Saber aplicá-los (Object Oriented Design).

Infelizmente, a maior parte dos programadores não vai além da primeira parte.

O.9 - EXEMPLOS

O.9.1 - Smalltalk-80

Selectores

Selectores unários - apenas um identificador

```
minhaConta saldo
alfa sin
```

Selectores binários - um ou dois caracteres alfanuméricos

```
saldo + valor
3 - 4
saldo >= 0
```

Selectores de palavras-chave - uma ou mais palavras-chave (identificador com ":"), aparecendo cada uma na mensagem seguida de um argumento.

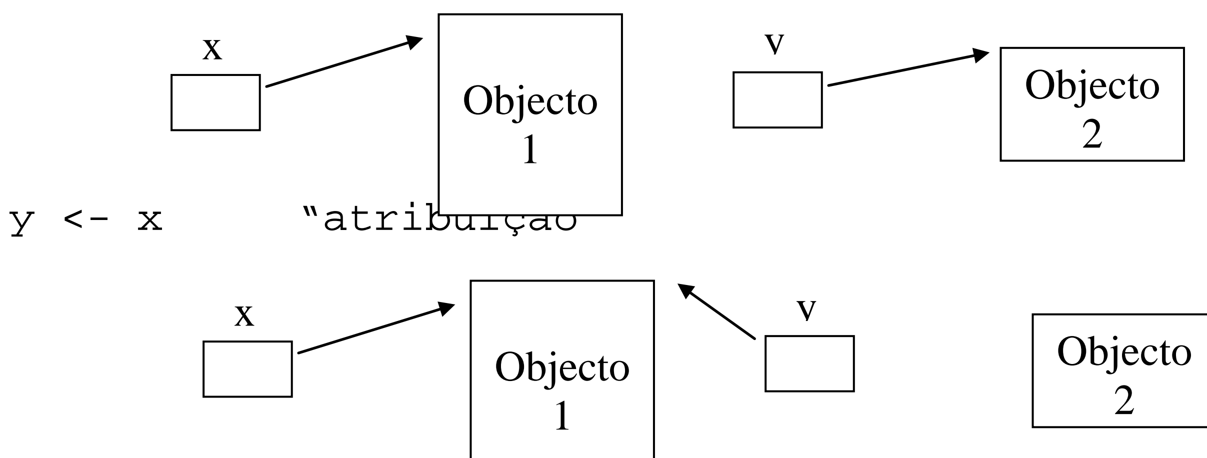
```
minhaConta depositar: 1000
minhaConta depositar: 1000 comJuro: 10
```

Atribuição

A **atribuição** muda o objecto que a variável referencia.

```
x <- 0
x <- #(1 2 3 )      "polimorfismo"
x <- y <- 0         "atribuição múltipla"
```

Mudança de referência:




```
y <- x      "atribuição
```

Os objectos que não são referenciados por nenhum outro são inúteis (dizem-se mortos).

Os objectos que morrem são reciclados automaticamente (recolecção automática de lixo).

Bloco

Bloco – Conjunto de instruções diferida (entre parêntesis rectos).

Um bloco é substituído por uma referência para um objecto que contém as suas instruções. Estas instruções só são executas quando o objecto receba a mensagem **value**.

```
saldo <- 0

[saldo <- 0] value

aux <- [saldo <- 0]
aux value

carrasco executa: [saldo <- 0]

executa: umBloco
  umBloco value
```

Selecção

Pascal	Smalltalk-80
If expressão-booleana **Then** instrução1 **Else** instrução2	Expressão-booleana **ifTrue**: [bloco1] **ifFalse**: [bloco1]

Herança

A subclasse herda as variáveis e os métodos da superclasse (excepto se redefinidos na subclasse).

As instâncias da subclasse serão idênticas às instâncias da superclasse excepto nas diferenças estabelecidas pela definição da subclasse.

Uma classe pode:

- Acrescentar variáveis.
- Acrescentar métodos.
- Redefinir métodos.

Superclasse	Object
Classe	ContaBancária
Variáveis	saldo
Métodos	**depositar: valor** saldo <- saldo + valor **levantar: valor** saldo >= valor **ifTrue:**[saldo <- saldo - valor. ^valor] **ifFalse**:[self **error**: 'Querias!'. ^0] **saldo** ^ saldo

Superclasse	ContaBancária
Classe	DepósitoAOrdem
Variáveis	Saldo
Métodos	**depositar: valor** **levantar: valor** **saldo** **multibanco**: **valor** valor <= 40000 **ifTrue:**[^self **levantar**: valor] **ifFalse**:[self **error**: 'Tanta massa?'. ^0]

O.9.2 - C++

Declarações podem aparecer em qualquer sítio.

```
for(int i=0; i<N; i++) …
```

Constantes

```
const int x = 2;          // x = …; é erro
```

```
const char* nome = “Zeca”;         // nome[0]=’L’;  erro
                                   // nome = “Manel”;  ok
```

Argumentos de funções

```
float soma (float vector[ ], int tamanho) {
        // corpo da função
}
…
float a [20];
int b = 20;
float resultado;
…                              // inicialização do array a
resultado = soma (a,b);
```

O compilador verifica que o número e tipo dos argumentos actuais corresponde aos dos formais, algo que muitos compiladores de C não fazem.

Argumentos por omissão

Correcto

```
int função (int um,
            float dois,
            char* três = “olá”,
            int quatro = 1 )
```

Errado

```
int função (int a = 0,
            char b,
            float c = 3.5 )
```

Overloading de nomes de funções

```
void print (int n) {
        printf(“inteiro = %d\n”, n)
}

void print (char* s) {
        printf(“string = %s\n”, s)
}

void main( ) {
        printf(2);                // invoca: void print (int n)
        printf(“olá”);            // invoca: void print (char* s)
}
```

Exemplo de definição de uma classe

```
class Ponto {
        int x,y;                        //coordenadas
public:
        Ponto ( )                       { x = y = 0;}
        Ponto ( int nx, int ny)         { x= nx; y = ny; }
        int x ( ) const                 { return x; }
        int x ( int nx )                        { return x = nx; }
        int y ( ) const                 { return y; }
        int y ( int ny )                        { return y = ny; }

        Ponto operator+ (const Ponto& p) const
                { return Ponto ( x + p.x, y + p.y); }

        Ponto& operator+= (const Ponto& p) const
                { x += p.x( ); y += p.y( ); return this;}
        void print (ostream& strm = cout) const;
};

void Ponto::print (ostream& strm = cout) const {
        strm << '(' << x << ',' << y << ')';
}

ostream& operator << (ostream& strm, const Ponto& p) {
        p.print (strm);
        return strm;
}
```

O.9.3 - Java

Exemplo de polimorfismo

As linguagens de programação orientadas a objectos conseguem reconhecer qual o tipo de objecto associado a uma variável e invocar o método da classe correspondente.

```
public class Polimorfismo {
        public static void main(String[] args) {
                Figura c1;
                Figura r1;
                c1 = new Circulo(10,2,2);               // Objecto tipo Figura contém Circulo
                r1 = new Rectangulo(3,3,10,5);          // Objecto tipo Figura contém Rectangulo
                System.out.println("Area c: " + c1.Area());
                System.out.println("Posx c: " + c1.pos.x);
                System.out.println("Posy c: " + c1.pos.y);
                System.out.println();
```

```
            System.out.println("Area q: " + r1.Area());
            System.out.println("Posx q: " + r1.pos.x);
            System.out.println("Posy q: " + r1.pos.y);
        }
}

class Figura {
        public Ponto pos = new Ponto();

        public double Area() {
            return 0.0;
        }
}

class Circulo extends Figura {
        double raio;
        public Circulo(double r) {
            raio = r;
            this.pos.x = 0;
            this.pos.y = 0;
        }

        public Circulo(double r, int x, int y) {
            raio = r;
            this.pos.x = x;
            this.pos.y = y;
        }

        public double Area() {
            return raio * raio * Math.PI;
        }
}

class Rectangulo extends Figura {
        double base;
        double altura;

        public Rectangulo(double b,double h) {
            base = b;
            altura = h;
            this.pos.x = 0;
            this.pos.y = 0;
        }

        public Rectangulo(double b,double h,int x, int y) {
            base = b;
            altura = h;
            this.pos.x = x;
```

```
        this.pos.y = y;
    }

    public double Area() {
        return base*altura;
    }
}

class Ponto {
    int x;
    int y;
}
```