APOSTILA

Módulo 1:
Metodologias de Desenvolvimento

Curso de Treinamento Específico para Programadores

Sumário

1. Metodologia de Desenvolvimento

1.1. Programação Estruturada

Programação estruturada é uma forma de programação de computadores que preconiza que todos os programas possíveis podem ser reduzidos a apenas três estruturas: **sequência, decisão e iteração** (repetição), desenvolvida por Michael A. Jackson no livro "*Principles of Program Design*" de 1975.

Tendo, na prática, sido transformada na **programação modular**, a programação estruturada orienta os programadores para a criação de estruturas simples nos programas, usando as **sub-rotinas** e as **funções**. Foi a forma dominante na criação de software anterior à programação orientada por objetos.

a) Exemplos de linguagens que utilizam programação estruturada

Cobol:

```
//COBUCLG  JOB CLASS=A,MSGCLASS=A,MSGLEVEL=(1,1)

//HELOWRLD EXEC COBUCLG,PARM.COB='MAP,LIST,LET'
//COB.SYSIN DD *
  001  IDENTIFICATION DIVISION.
  002  PROGRAM-ID.  'HELLO'.
  003  ENVIRONMENT DIVISION.
  004  CONFIGURATION SECTION.
  005  SOURCE-COMPUTER.  IBM-360.
  006  OBJECT-COMPUTER.  IBM-360.
  0065 SPECIAL-NAMES.
  0066     CONSOLE IS CNSL.
```

```
  007  DATA DIVISION.
  008  WORKING-STORAGE SECTION.
  009  77  HELLO-CONST   PIC X(12) VALUE 'HELLO, WORLD'.
  075  PROCEDURE DIVISION.
  090  000-DISPLAY.
  100      DISPLAY HELLO-CONST UPON CNSL.
  110      STOP RUN.
//LKED.SYSLIB DD DSNAME=SYS1.COBLIB,DISP=SHR
//            DD DSNAME=SYS1.LINKLIB,DISP=SHR
//GO.SYSPRINT DD SYSOUT=A
//
```

C

```c
# include <stdio.h>


struct Pessoa
{
   char nome[64]; // vetor de 64 chars para o nome
   unsigned short int idade;
   char cpf[13];
};


int main()
{
 // declaração da variável "exemplo"
   struct Pessoa exemplo = {"Fulano", 16, "00.000.000-00"};


   printf("Nome: %s\n", exemplo.nome);
   printf("Idade: %hu\n", exemplo.idade);
   printf("CPF: %s\n", exemplo.cpf);


   getchar(); // desnecessário, mas comumente utilizado em
ambientes windows para "segurar" o terminal aberto
   return 0;
}
```

PHP

```php
<?php


  $vet01 = array();
  $vet01[] = "Sistemas operacionais";
  $vet01[] = "Compiladores";
  $vet01[] = "Bancos de dados";


  $vet02 = array(1, 2, 3, 4, 5);
```

```php
$vet03 = array( 0 => 0, 2 => 3, 10 => "item 10");

for ($i = 0; $i < count($vet01); $i++) {
  echo $vet01[$i] . "<br />";
}

?>
```

b) Por que PHP?

É uma linguagem de programação baseada em software livre, sem custos para o usuário.

É o padrão de linguagem de **programação web** definida pela Prefeitura de Juiz de Fora.

Possui uma comunidade de programadores cada vez maior.

c) O PHP

PHP quer dizer **P**HP: **H**ypertext **P**reprocessor (PHP: Processador de Hipertexto).

Esta linguagem nasceu pela mão de Rasmus Lerdof em 1994, como um CGI escrito em Linguagem C que inicialmente interpretava muito facilmente formulários.

A primeira designação dada foi de FI (Form Interpreter) porém, devido à criação de inúmeras funções pela comunidade, que se ia desenvolvendo pela internet, a linguagem teve que, em 1997, ser redenominada como PHP.

O PHP é uma das mais abrangentes ferramentas que o homem possui atualmente.

Por ser uma linguagem **server-side**, ou seja roda direto do servidor e só mostra ao usuário o resultado já processado.

É possível a criação de uma grande variedade de coisas com o PHP, tendo em vista que esta é uma linguagem que podemos definir como inteligente, pois se não existir uma classe para uma determinada função podemos criá-la.

Porém, ferramentas que rodam do lado do cliente como a abertura de um pop-up, uma animação de serpentinas exibidas na tela ou um slideshow de imagens não podem ser criados por esta linguagem.

Se procura uma boa linguagem para criar esse tipo de ferramentas procure o **Javascript**. É por isso que se diz que o PHP e o Javascript são linguagens que se complementam!

d) Início a programação PHP

Olá mundo:

O código php que se escreve, conforme abaixo, representa apenas uma parte do que é enviado para o cliente.

```
<?php
     echo "Olá mundo";
?>
```

O código abaixo representa a totalidade do conteúdo exibido para o cliente.

```
<html>
 <head>
  <title>Meu primeiro script</title>
 </head>
 <body>
 <?php
   echo "Olá mundo";
 ?>
 </body>
</html>
```

O uso do ponto-e-vírgula

```
<?php
   echo "Olá tudo bem?";
   echo "Como você está?";
?>
```

Não é necessário escrever em linhas diferentes os comandos, mas é recomendável para evitar confusões.

```
<?php
  echo "Olá tudo bem?"; echo "Como você está?";
?>
```

Como o PHP é baseado no C e no C++, ele suporta a sintaxe de comentários das duas linguagens, veja baixo.

Com //comentário:

```php
<?php
    echo "Um comentário!"; //Comentário de um linha só
?>
```

Com /* comentário */:

```php
<?php
    /* Isto é um comentário
       de várias linhas no PHP */
    echo "Outro comentário acima!";
?>
```

As strings passadas para a instrução *echo* também podem conter formatações de texto em HTML.

```php
<?php
    echo "<h2> Título em h2 </h2>";
    echo "<h3> título em h3 </h3>";
    echo "<i>Em itálico</i>";
    echo "<b>Em negrito</b>";
    echo "<strong>Em negrito</strong>";
?>
```

As aspas duplas " podem sempre ser substituídas por apóstrofos/aspas simples '. Ambas as formas estão corretas.

```php
<?php
    echo "Na frase a seguir o nome xpto virá dentro de
    aspas: ";
    echo "O personagem que eu mais gosto é o \"xpto\", sem
    dúvida";
?>
```

Inclusão de trechos de código:

```php
<?php

    include('code.php');      // Inclui e executa um trecho opcional de código

    include 'code.php';       // Maneira alternativa, funciona apenas com include e require.

    require('code.php');      // O mesmo que 'include', porém pára a execução caso o arquivo não seja encontrado

    require_once('code.php'); // O mesmo que require, mas evita que o trecho seja incluído novamente

?>
```

Exemplo de uso do if e eles:

```php
$x=3;

if ($x==2){

  echo "x vale 2";

} else if ($x==3){

  echo "x vale 3";

}else{

  echo "x é diferente de 2 e de 3";

}
```

Exemplo de uso do swtich:

```php
switch ($i) {
    case 0:
        echo "i equals 0";
    break;
    case 1:
        echo "i equals 1";
    break;
    case 2:
        echo "i equals 2";
```

```
    break;
}


switch ($i) {
   case "apple":
        echo "i is apple";
    break;
    case "bar":
        echo "i is bar";
    break;
    case "cake":
        echo "i is cake";
    break;
}
```

Exemplo do uso do for:

```
/* exemplo 1 */
$i = 1;
for (; ; ) {
    if ($i > 10) {
        break;
    }
    echo $i;
    $i++;
}
/* exemplo 2 */
for ($i = 1, $j = 0; $i <= 10; $j += $i, print $i, $i++);
/* exemplo 3 */
```

```php
for ($i = 1; $i <= 10; $i++) {
    echo $i;
}
/* exemplo 4 */
for ($i = 1; ; $i++) {
    if ($i > 10) {
        break;
    }
    echo $i;
}
```

Exemplo do uso do while:

```php
/* example 1 */
$i = 1;
while ($i <= 10) {
    echo $i++;
/* O valor impresso poderá ser $i antes do incremento (pós-incremento*/
}
/* example 2 */
$i = 1;
while ($i <= 10):
    echo $i;
    $i++;
endwhile;
```

Variáveis:

Começam sempre pelo símbolo $ seguido de uma letra.

Podem conter símbolos numéricos (0 - 9) alfanuméricos minúsculos (a - z) e alfanuméricos maiúsculos.

Não podem conter espaços! Se tiverem mais do que uma palavra deverão ser interligadas por um underscore _.

Por exemplo: $total_variavel.

Alguns exemplos de variáveis:

```php
<?php
    $minha_variavel=4;
    $minha_string="super ";
    echo "$minha_string "." $minha_variavel";
?>
```

O ponto . após a variável $minha_string concatena as variáveis.

Deverá escrever na tela do navegador: super 4.

As variáveis são no PHP representadas por um $ seguido do nome a ela atribuído. É necessário ter em conta alguns cuidados quando criamos variáveis:

```php
<?php
    $var = "Bom ";
    $Var = "dia!";
    echo $var . $Var;   // Exibe "Bom dia!"
```

```php
$25arrobas = "Ainda por cumprir!";
                    // INVÁLIDO - Nenhum nome de variável pode
                         // começar por uma expressão numérica


$_25arrobas = "Quase na linha!"; // VÁLIDO - Nome começa por _

?>
```

A inclusão de variáveis funciona de várias maneiras.

```php
<?php

    $carro = "Mercedes";

    echo "Ele comprou um bonito $carro";      // funciona

    echo "Eles compraram vários {$carro}s";   // funciona

    echo "Eu faço um ${carro}";               // funciona

?>
```

Podemos também modificar Strings.

```php
<?php

    $str = "Olha que alí há mal";

    // Apresenta "Olha que alí há mal"

    $str{strlen($str)-1} = "r";

    // Apresenta "Olha que alí há mar"

?>
```

ou

```php
<?php

    $str = "Pegue isto";         // Apresenta "Pegue isto"

    $str{strlen($str)-10} = "S";  // Apresenta "Segue isto"

?>
```

Manipulação de Tipos:

No PHP não é necessário definir o tipo que queremos usar, ou seja, este é determinado pelo contexto em que é usado. Por exemplo, $var = "string" é uma string, já $var = 12, é um inteiro.

Para alternarmos entre os tipos usamos uma sintaxe de moldagens:

```
<?php
    $foo = 5;                   // $foo é um inteiro
    $bar = (boolean) $foo;      // $bar é um boleano
?>
```

Moldagens permitidas:

int ou *integer*: moldar para inteiro.

bool ou *boolean*: moldar para booleano.

float, double ou *real*: moldar para número de ponto flutuante.

string: moldar para string.

array: moldar para array.

object: moldar para objeto.

Exemplos de moldes:

```
<?php
    $str = "Eu sou string";
    $int = 12;
    $num = 25/85;
```

```php
    $a = (boolean) $str;
    $b = (string) $int;
    $c = (int) $num;

    echo $a . $b . $c;
?>
```

Arrays, exemplo:

```php
<?php
    $arr = array(1 => "um", 2 => "dois", 3 => TRUE);
    echo $arr[1];  // Imprime "um"
    echo $arr[3];  // Imprime "TRUE"
?>
```

Aqui nos é apresentado 3 chaves (1, 2 e 3) e 3 valores ("um", "dois" e TRUE).

Então podemos pegar uma chave de um array para imprimirmos o valor correspondente. Podemos também definir um array como vários array:

```php
<?php
    $arr = array("versão" => array ("beta"    => "0.5x",
                                    "alpha"   => "0.1x",
                                    "release" => "0.8x",
                                    "final"   => "1.x"
                                    ),
                 "nome" => "PHP software",
```

```
                    "SO" => array("win" => "Windows",
                                  "lin" => "Linux",
                                  "mac" => "MacOS"
                                  )
                    );


        // Vamos agora remover um elemento do array, visto que o
        nosso
        // software já passou a fase alpha
        unset($arr["versão"]["alpha"]);


        // Também podemos apagar o array inteiro. Já não vamos
        // disponibilizar o nosso software
        unset($arr);
?>
```

Podemos especificar apenas valores num array, sendo que o interpretador irá tornar cada um dos valores com uma chave a partir do zero.

```
<?php
        $arr = array(1,45,23,68);
        echo $arr[1]; // imprime 45
        // Vamos agora apagar um valor e reindexar o nosso
        array:
        unset($arr[1]);
        $arr = array_values($arr);
        // Não podemos imprimir o nosso array através de echo.
        Temos
        // que fazer isso com print_r():
```

```
        print_r($arr);
    ?>
```

Operadores de comparação em PHP

O PHP apresenta um operador chamado idêntico, que compara não somente se o valor de uma variável é igual ao valor em comparação, mas também se os tipos são iguais, uma vez que os tipos em PHP são definidos de forma dinâmica esse operador costuma ser muito bem-vindo, outros exemplos de operadores estão descritos na tabela abaixo:

Nome	Exemplo	Significado
Igual	$a == $b	Verdadeiro se $a é igual a $b
Idêntico	$a === $b	Verdadeiro se $a igual a $b e do mesmo tipo
Diferente	$a != $b	Verdadeiro se $a diferente de $b
Diferente	$a <> $b	Verdadeiro se $a diferente de $b
Não idêntico	$a !== $b	Verdadeiro se $a diferente de $b, ou não são do mesmo tipo
Menor que	$a < $b	Verdadeiro se $a menor que $b
Maior que	$a > $b	Verdadeiro se $a maior que $b
Menor ou igual	$a <= $b	Verdadeiro se $a menor ou igual a $b
Maior ou igual	$a >= $b	Verdadeiro se $a maior ou igual a $b

1.2. Programação Orientada a Objetos

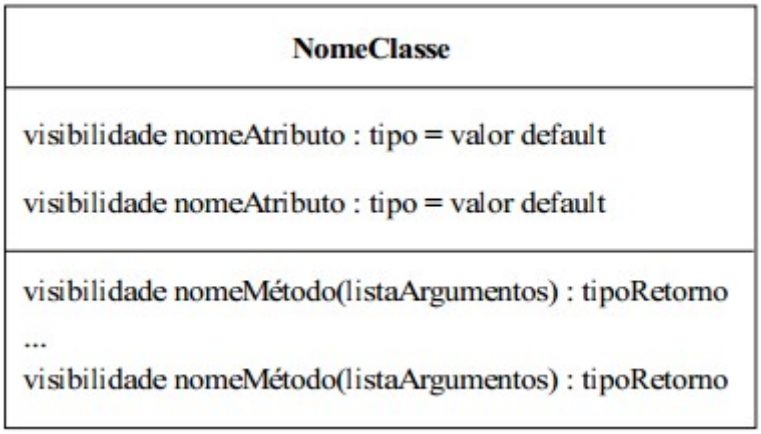
Nesta apostila são apresentados os conceitos básicos que permeiam o uso das técnicas de orientação a objetos na programação, utilizando a linguagem Java como motivador. Objetos são instâncias de classes, que determinam qual informação um objeto contém e como ele pode manipulá-la.

Um dos grandes diferenciais da programação orientada a objetos em relação a outros paradigmas de programação que também permitem a definição de estruturas e operações sobre essas estruturas está no conceito de herança, mecanismo através do qual definições existentes podem ser facilmente estendidas. Juntamente com a herança deve ser enfatizada a importância do polimorfismo, que permite selecionar funcionalidades que um programa irá utilizar de forma dinâmica, durante sua execução.

a) Classes

A definição de classes e seus inter-relacionamentos é o principal resultado da etapa de projeto de software. Em geral, esse resultado é expresso em termos de alguma linguagem de modelagem, tal como UML. Uma classe é um "gabarito" para a definição de objetos. Através da definição de uma classe, descreve-se que propriedades — ou atributos — o objeto terá. Além da especificação de atributos, a definição de uma classe descreve também qual o comportamento de objetos da classe, ou seja, que funcionalidades podem ser aplicadas a objetos da classe. Essas funcionalidades são descritas através de métodos.

Um método nada mais é que o equivalente a um procedimento ou função, com a restrição que ele manipula apenas suas variáveis locais e os atributos que foram definidos para a classe. Uma vez que estejam definidas quais serão as classes que irão compor uma aplicação, assim como qual deve ser sua estrutura interna e comportamento, é possível criar essas classes em Java. Na *Unified Modeling Language* (UML), a representação para uma classe no diagrama de classes é tipicamente expressa na forma gráfica, como mostrado na figura abaixo.



Como se observa nessa figura, a especificação de uma classe é composta por três regiões: o nome da classe, o conjunto de atributos da classe e o conjunto de métodos da classe.

O **nome da classe** é um identificador para a classe, que permite referenciá-la posteriormente — por exemplo, no momento da criação de um objeto. O conjunto de **atributos** descreve as propriedades da classe. Cada atributo é identificado por um **nome** e tem um **tipo** associado. Em uma linguagem de programação orientada a objetos pura, o tipo é o nome de uma classe. Na prática, a maior parte das linguagens de programação orientada a objetos oferecem um grupo de tipos primitivos, como inteiro, real e caráter, que podem ser usados na descrição de

atributos. O atributo pode ainda ter um **valor_default** opcional, que especifica um valor inicial para o atributo.

Os **métodos** definem as funcionalidades da classe, ou seja, o que será possível fazer com objetos dessa classe. Cada método é especificado por uma **assinatura**, composta por um identificador para o método (o nome do método), o tipo para o valor de retorno e sua lista de argumentos, sendo cada argumento identificado por seu tipo e nome. Através do mecanismo de **sobrecarga** *(overloading)*, dois métodos de uma classe podem ter o mesmo nome, desde que suas assinaturas sejam diferentes. Tal situação não gera conflito pois o compilador é capaz de detectar qual método deve ser escolhido a partir da análise dos tipos dos argumentos do método.

O **modificador de visibilidade** pode estar presente tanto para atributos como para métodos. Em princípio, três categorias de visibilidade podem ser definidas:

- **público,** denotado em UML pelo símbolo +: nesse caso, o atributo ou método de um objeto dessa classe pode ser acessado por qualquer outro objeto (visibilidade externa total);
- **privativo,** denotado em UML pelo símbolo -: nesse caso, o atributo ou método de um objeto dessa classe não pode ser acessado por nenhum outro objeto (nenhuma visibilidade externa);
- **protegido,** denotado em UML pelo símbolo #: nesse caso, o atributo ou método de um objeto dessa classe poderá ser acessado apenas por objetos de classes que sejam derivadas dessa através do mecanismo de herança.

Relacionamentos Representa a interação entre classes e seus objetos. Pode, por exemplo representar que a alteração de um objeto (o objeto independente) pode afetar outro objeto (o objeto dependente).

Os relacionamentos possuem:

- Nome: descrição dada ao relacionamento (faz, tem, possui,...)
- Sentido de leitura: sentido pelo qual o relacionamento é lido. Por exemplo, pode-se ver que a relação entre a classe Aluno e a classe Professor, pode ser lida de diferentes formas tais quais (i) professor leciona para aluno ou (ii) aluno é ensinado pelo professor.
- Navegabilidade: indicada por uma seta no fim do relacionamento.
- Multiplicidade: 0..1, 0..*, 1, 1..*, 2, 3..7
- Tipo: associação (agregação, composição), generalização e dependência Papéis: desempenhados por classes em um relacionamento

b) Objetos

Objetos são instâncias de classes. É através deles que (praticamente) todo o processamento ocorre em sistemas implementados com linguagens de programação orientadas a objetos. O uso racional de objetos, obedecendo aos princípios associados à sua definição conforme estabelecido no paradigma de desenvolvimento orientado a objetos, é chave para o desenvolvimento de sistemas complexos e eficientes.

Um objeto é um elemento que representa, no domínio da solução, alguma entidade (abstrata ou concreta) do domínio de

interesse do problema sob análise. Objetos similares são agrupados em classes.

No paradigma de orientação a objetos, tudo pode ser potencialmente representado como um objeto. Sob o ponto de vista da programação orientada a objetos, um objeto não é muito diferente de uma variável normal. Por exemplo, quando define-se uma variável do tipo int em uma linguagem de programação como C ou Java, essa variável tem:

- um espaço em memória para registrar o seu estado (valor);
- um conjunto de operações que podem ser aplicadas a ela, através dos operadores definidos na linguagem que podem ser aplicados a valores inteiros.

Da mesma forma, quando se cria um objeto, esse objeto adquire um espaço em memória para armazenar seu estado (os valores de seu conjunto de atributos, definidos pela classe) e um conjunto de operações que podem ser aplicadas ao objeto (o conjunto de métodos definidos pela classe).

Um programa orientado a objetos é composto por um conjunto de objetos que interagem através de "trocas de mensagens" . Na prática, essa troca de mensagem traduz-se na aplicação de métodos a objetos.

As técnicas de programação orientada a objetos recomendam que a estrutura de um objeto e a implementação de seus métodos devem ser tão privativos como possível. Normalmente, os atributos de um objeto não devem ser visíveis externamente. Da mesma forma, de um método deve ser suficiente conhecer apenas sua especificação, sem necessidade de saber detalhes de como a funcionalidade que ele executa é implementada.

- **Encapsulação** é o princípio de projeto pelo qual cada componente de um programa deve agregar toda a informação relevante para sua manipulação como uma unidade (uma cápsula). Aliado ao conceito de ocultamento de informação, é um poderoso mecanismo da programação orientada a objetos.
- **Ocultamento da informação** é o princípio pelo qual cada componente deve manter oculta sob sua guarda uma decisão de projeto única. Para a utilização desse componente, apenas o mínimo necessário para sua operação deve ser revelado (tornado público).

Na orientação a objetos, o uso da encapsulação e ocultamento da informação recomenda que a representação do estado de um objeto deve ser mantida oculta. Cada objeto deve ser manipulado exclusivamente através dos métodos públicos do objeto, dos quais apenas a assinatura deve ser revelada. O conjunto de assinaturas dos métodos públicos da classe constitui sua interface operacional. Dessa forma, detalhes internos sobre a operação do objeto não são conhecidos, permitindo que o usuário do objeto trabalhe em um nível mais alto de abstração, sem preocupação com os detalhes internos da classe. Essa facilidade permite simplificar a construção de programas com funcionalidades complexas, tais como interfaces gráficas ou aplicações distribuídas.

c) Herança

O conceito de encapsular estrutura e comportamento em um

tipo não é exclusivo da orientação a objetos; particularmente, a programação por tipos abstratos de dados segue esse mesmo conceito. O que torna a orientação a objetos única é o conceito de herança.

Herança é um mecanismo que permite que características comuns a diversas classes sejam fatoradas em uma classe base, ou superclasse. A partir de uma classe base, outras classes podem ser especificadas. Cada classe derivada ou subclasse apresenta as características (estrutura e métodos) da classe base e acrescenta a elas o que for definido de particularidade para ela.

Há várias formas de relacionamentos em herança:

- **Extensão:** a subclasse estende a superclasse, acrescentando novos membros (atributos e/ou métodos). A superclasse permanece inalterada, motivo pelo qual este tipo de relacionamento é normalmente referenciado como **herança estrita**.
- **Especificação:** a superclasse especifica o que uma subclasse deve oferecer, mas não implementa nenhuma funcionalidade. Diz-se que apenas a interface (conjunto de especificação dos métodos públicos) da superclasse é herdada pela subclasse.
- **Combinação de extensão e especificação:** a subclasse herda a interface e uma implementação padrão de (pelo menos alguns de) métodos da superclasse. A subclasse pode então redefinir métodos para especializar o comportamento em relação ao que é oferecido pela superclasse, ou ter que oferecer alguma implementação para métodos que a superclasse tenha declarado mas não implementado.

Normalmente, este tipo de relacionamento é denominado herança polimórfica.

d) Polimorfismo

Polimorfismo é o princípio pelo qual duas ou mais classes derivadas de uma mesma superclasse podem invocar métodos que têm a mesma identificação (assinatura) mas comportamentos distintos, especializados para cada classe derivada, usando para tanto uma referência a um objeto do tipo da superclasse. Esse mecanismo é fundamental na programação orientada a objetos, permitindo definir funcionalidades que operem genericamente com objetos, abstraindo-se de seus detalhes particulares quando esses não forem necessários.

Para que o polimorfismo possa ser utilizado, é necessário que os métodos que estejam sendo definidos nas classes derivadas tenham exatamente a mesma assinatura do método definido na superclasse; nesse caso, está sendo utilizado o mecanismo de **redefinição de métodos** *(overriding)*.

No caso do polimorfismo, o compilador não tem como decidir qual o método que será utilizado se o método foi redefinido em outras classes — afinal, pelo princípio da substituição um objeto de uma classe derivada pode estar sendo referenciado como sendo um objeto da superclasse. Se esse for o caso, o método que deve ser selecionado é o da classe derivada e não o da superclasse. Dessa forma, a decisão sobre qual dos métodos que deve ser selecionado, de acordo com o tipo do objeto, pode ser tomada apenas em tempo de execução, através

do mecanismo de **ligação tardia**. O mecanismo de ligação tardia também é conhecido pelos termos em inglês *late binding*, *dynamic binding* ou ainda *run-time binding*.