Java Building

Princípios

- Herança
- Nomenclatura

Polimorfismo

Polimorfismos

Polimorfismo é uma característica importante em qualquer linguagem, especialmente se a linguagem é orientada a objetos, e especificamente em Java. O polimorfismo é muitas vezes confundido com o próprio conceito de orientação a objetos, mas embora dependa dele em certa medida, é, na realidade, um conceito separado.

O polimorfismo está presente de diferentes maneiras. Estas são aquelas relevantes para Java mas quase todas se aplicam a outras linguagens.

Variável Polimórfica

Este tipo de polimorfismo permite atribuir objetos de tipos (classes, interfaces, ...) diferentes a uma mesma variável. Para que isto seja possível deve existir uma relação de hiearquia entre o tipo da variável e o tipo do objeto, de tal modo que o tipo da variável seja igual ou mais abstrato que o tipo do objeto.

Este tipo do polimorfismo é na realidade uma forma de encapsulamento que, por sua vez, é uma concretização do Princípio de Separação de Responsabilidade; e é necessária para uma linguagem orientada a objeto completa.

No exemplo a seguir a variável texto não é polimórfica já que `String` é uma classe final. Não é possível estender `String` e, portanto, não é possivel criar tipos menos abstratos que `String`.

```
1.String texto;
2.texto = "Apenas uma String";
```
Código 1:

No exemplo seguinte, a varável texto é polimórfica já que `CharSequence` é uma interface, o que, por definição, significa que é possivel que exista um conjunto de possibilidades para os objetos que podem ser atribuidos a essa variável.

```
1.CharSequence texto;
2.texto = "Apenas uma String" ;
3.texto = new StringBuffer ( "Ou um StringBuffer" ) ;
4.texto = new StringBuilder ( "Ou um StringBuilder" ) ;
5.
```
Código 2:

Utilizar interfaces e classes abstratas como os tipos das variáveis garante explicitamente que a variável é polimórfica. Utilizar classes abstratas e sobretudo interfaces como tipos de variáveis é considerado uma boa prática e é conhecido pela expressão "programar para interfaces". Na realidade, isto apenas significa : "utilize variáveis polimórficas por padrão e sempre que possível".

Sombreamento

Sombreamento (shadowing) é a capacidade de poder definir duas, ou mais, variáveis com o mesmo nome em escopos diferentes. O código a seguir apresenta o exemplo clássico:

```
01.public class UmaClasse {
02.
03.         String nome; // variável no escopo "classe"
04.
05.         public void setName ( String nome ){ // variável no escopo "método"
06.                 this .nome = nome;
07.         }
08.}
09.
```
Código 3:

O sobreamento permite que o mesmo nome seja utilizado para duas variávels diferentes. No caso, a variável "nome" definida na classe e a variável "nome" definida no método. O detalhe com o uso de sobreamento é que as variáveis de maior escopo podem interagir com as de menor escopo. Contudo, como elas têm o mesmo nome, é necessário destingui-las. Para isso, é utilizada a palavra reservada `this` que representa o objeto corrente e contém, portanto, variáveis de escopo de classe. Caso o `this` não fosse utilizado junto de ?nome?, o compilador assume que você está se referindo à variável de menor escopo; no caso a definida no método. Isso não é uma falha. É a utilidade do sobreamento.

O compilador Java é um tanto esperto e avisa o programador de falhas básicas. Uma delas é a tentativa de atribuir uma variável a ela própria. Isso é um código que não tem nenhum propósito e o compilador o avisará quando detectar essa situação. Por isso se você escrever o código seguinte:

```
01.public class UmaClasse {
02.
03.         String nome; // variável no escopo "classe"
04.
05.         public void setName ( String nome ){ // variável no escopo "método"
06.                 nome = nome;
07.         }
08.}
09.
```
Código 4:

O compilador apresentará um aviso na linha 6 dizendo que a variável está sendo atribuída a ela própria. Isso demonstra que o compilador escolhe sempre a variável de menor escopo.

Sobrecarga

Sobrecarga (overload) é a capacidade de poder definir dois, ou mais métodos, numa mesma classe, ou suas derivadas, com o mesmo nome.

Para que exista sobecarga não é necessário que a linguagem seja orientada a objetos e, por isso, à semelhança do sombreamento a sobrecarga é normalmente entendida com uma característica da linguagem e não como uma forma de polimorfismo.

Embora os métodos possam ter o mesmo nome, eles têm obrigatoriamente que ter uma assinatura diferente. Eis alguns exemplos:

```
1.public int calculaIdade ( int ano , int mes, int dia ) ;
2.public int calculaIdade ( Date data ) ;
3.public int calculaIdade ( Calendar data ) ;
4.
```

Código 5:

Sobrescrita

Sobrescrita (overriding) é a capacidade de poder redefinir a implementação de um método que já foi definido e implementado em uma classe superior na hierarquia de herança.

Para que exista sobre-escrita é necessário que o método seja definido com a exata assinatura que existe na classe superior.

```
01.public class Somador {
02.
03.          public int calculaSoma ( int inicio, int fim ){
04.
05.                    int soma = 0 ;
06.                    for ( int i = inicio ; i <= fim ; i++ ){
07.                      soma += i;
08.                    }
09.                    return soma;
10.          }
11.
12.}
13.
14.public class SomadorInteligente extends Somador {
15.
16.          public int calculaSoma ( int inicio, int fim ){
17.
18.                    int umAteInicio = inicio ( inicio + 1 ) / 2 ;
19.                    int umAteFim = fim ( fim + 1 ) / 2 ;
20.
21.                    return umAteFim ? umAteInicio;
22.          }
23.
24.   }
25.
```

Código 6:

O método `calculaSoma` em `SomadorInteligente` sobrescreve o método `calculaSoma` em `Somador` redefinindo a logica de soma.

Tipo Genérico

Tipos genéricos permitem estabelecer relações fortes entre os tipos, mas sem especificar o tipo real que será utilizado. Tipos genéricos são uma inovação recente da linguagem Java mas já conhecidos e utilizados em linguagems anteriores.

Esta forma de parametrização possibilita que classe sejam criadas utilizando uma outra classe ou grupo de classes sem necessidade de fazer casting explícito e possibilitando maior controle sobre o funcionamento da classe. Tipo genérico é uma parametrização do tipo e portanto é utilizando em qualquer lugar onde tipo é usado. Por exemplo, na definição de uma variável.

```
1.List<Number> numbers = new ArrayList<Number> () ;
2.
```

Código 7:

Tipo generico, à semelhança da variável polimorfica, é também um forma de encapsulamento.

Auto-boxing e Auto-unboxing

Java suporta tipos primitivos,ou seja, tipos de variáveis que não são objetos. Em algumas situações é necessário converter esses valores primitivos para objetos. Isso é conhecido como boxing (colocar em caixas). O processo inverso é chamado unboxing (retirar das caixas). Por exemplo, converter um `int` para um `Integer` pode ser feito assim:

```
1.int inteiroPrimitivo = 5;
2.Integer inteiroObjecto = Integer.valueOf(inteiroPrimitivo);
```

Código 8:

Auto-boxing e Auto-umboxing acontece quando próprio compilador faz essa operação. Este recurso foi adicionado à linguagem Java a partir da sua versão 5.

Numero de argumentos indefinido (Var args)

Algumas vezes é útil passar vários argumentos de um certo tipo para um método. Normalmente isso é feito pela utilização de arrays ou coleções. Contudo, nem sempre isso é prático do ponto de vista do programador.

O ideal seria passar os argumentos como se fossem argumentos individuais e capturá-los depois como um array. Essa funcionalidade conhecida como var args é uma forma de polimorfismo já que a forma como o programador invoca o método é diferente da forma com que ele trabalha o resultado, contudo os dados são os mesmos.

```
01.public class Vector {
02.
03.         public void setElements ( int ? elements ){
04.
05.              for ( int i = 0 ; i < elements.length; i++ ){
06.                // faz algo com o elemento
07.              }
08.         }
09.
10.}
11.
12.  // uso
13.
14.  Vector v;
15.  v.setElements ( 4 , 8 , 15 , 16 , 23 , 42 )
```

Código 9:

Categorias de Polimorfismo

Alguns dos tipos de polimorfismo escondem o real funcionamento do programa por serem forma de encapsulamento. Para alguns tipos de polimorfismo única forma de saber exatamente o que está acontecendo é

analisar o programa enquanto está funcionando. Estes tipos de polimorfismo formam uma categoria designada: Polimorfismo Dinâmico.

Os outros tipos cujo efeito no programa é claro mesmo quando o programa não está funcionando, ou seja, pode ser compreendido diretamente da análise do código compõem a categoria designada: Polimorfismo Estático.

Muitas das funcionalidades do polimorfismo estático não dependem do conceito de objeto e podem ser encontrados em outras linguagens, mesmo nas não orientadas a objetos. Talvez por isso não seja comum encontrar referência a essas capacidades como tipos de polimorfismo e são normalmente apresentadas como funcionalidades da linguagem. Em contrapartida, as funcionalidades de polimorfismo dinâmico dependem, quase todas, dos conceitos de objeto e herança (ou alguma forma de hierarquia). Muitas vezes elas se confundem com os próprios conceitos de herança e orientação a objetos e são normalmente apresentadas como parte integrante desse paradigma.

Eis um sumário da características integrantes do polimorfismo dinâmico:

- Variáveis Polimórficas
- Tipo genérico
- Polimorfismo Estático

Eis um sumário da caracteristicas integrantes do polimorfismo estático:

- Sobrecarga
- Sobrescrita
- Sombreamento
- Auto-(un)boxing
- Var args

Referências

[1] **Design Patterns: Elements of Reusable Object-Oriented Software**

Livro:Design Patterns: Elements of Reusable Object-Oriented Software

[2] **Design Patterns Java Workbook**

Livro:Design Patterns Java Workbook