Programação I

Polimorfismo e Genéricos

Samuel da Silva Feitosa

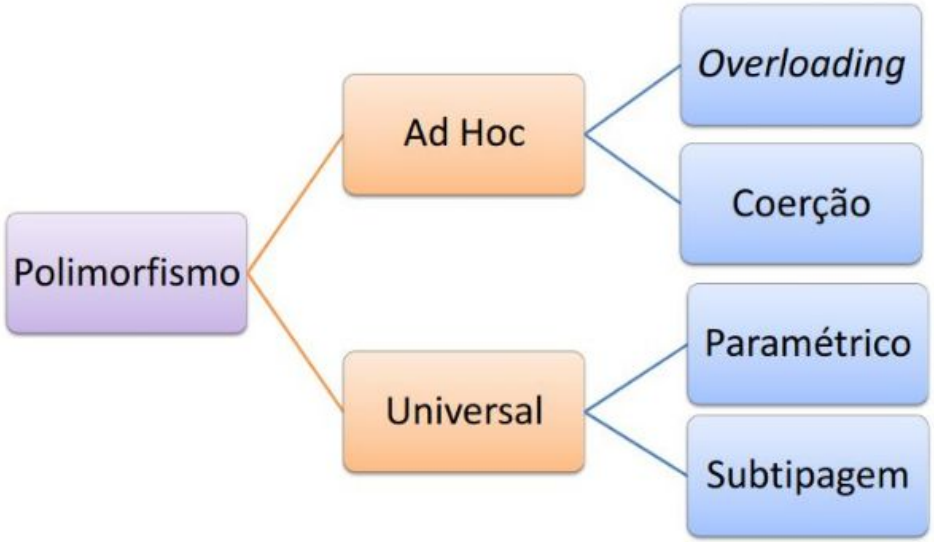
Aula 12

Polimorfismo

- Termo originário do grego: “Muitas formas”.
  - Poli = Muitas, Morphos = formas.
- Em POO, indica a realização de uma tarefa de formas diferentes.
- Visto claramente na chamada de métodos:
  - Objetos de tipos diferentes (porém relacionados) são capazes de decidir qual método acionar, produzindo resultados diferentes.
  - Resolução de chamada de métodos ou construtores sobrecarregados.

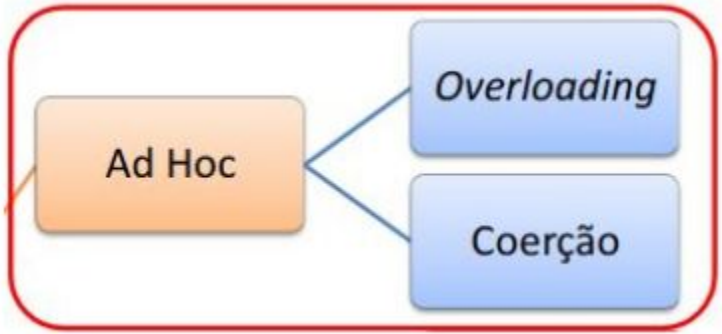
Polimorfismo no Java

- Em sua versão atual, o Java apresenta diferentes formas de polimorfismo.



Polimorfismo Ad-hoc

- Número finito de variações.
  - Sobrecarga (*overloading*)
  - Coerção
- Resolvido de forma estática, ou seja, em tempo de compilação.



Sobrecarga - Exemplo

- O que acontece se repetirmos o tipo do parâmetro ‘a’ nos dois métodos?

```
public class Quadrado {
    public static int quadrado(int a ) {
        System.out.println("Quadrado do int: " + a);
        return a * a;
    }

    public static double quadrado(double a) {
        System.out.println("Quadrado do double: " + a);
        return a * a;
    }

    public static void main(String[] args) {
        quadrado(1);
        quadrado(1.0);
    }
}
```

Coerção de tipo - Exemplo

- Quais dessas chamadas são válidas?

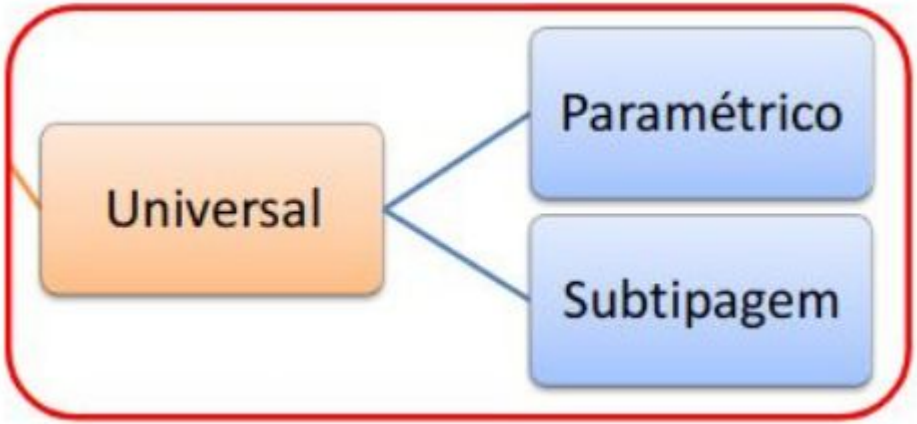
```
public class Coercao {
    public static void f(double x) {
        System.out.println(x);
    }
    public static void main(String args[]) {
        f(3.1416);
        f((byte) 1);
        f((short) 2);
        f('a');
        f(3);
        f(4L);
        f(5.6F);
    }
}
```

Resumo - Polimorfismo Ad-hoc

- Sobrecarga (*overloading*)
  - Utiliza os tipos para escolher a definição.
- Coerção
  - Utiliza a definição para escolher o tipo de conversão.

Polimorfismo Universal

- Número infinito de variações.
  - Polimorfismo de Subtipagem
  - Polimorfismo Paramétrico



Polimorfismo de Subtipagem

- Num sistema bancário, um cliente pode ter cartões de crédito e de débito.
  - Podemos supor que o pagamento em cada modalidade possui diferenças de cobranças pelo banco, por sua utilização.
  - Sendo assim, vamos criar uma classe Cartão, para representar essas diferenças.

```
public class Cartao {
    public Cliente cliente;
    public Conta conta;

    public void pagarCredito(double valor) {
        // implementação
    }

    public void pagarDebito(double valor) {

    }
}
```

Polimorfismo de Subtipagem

- Notamos que a classe apresentada possui dois métodos:
  - O primeiro para registrar o pagamento em cartão de crédito, e o segundo para pagamento em cartão de débito.
  - Caso tivéssemos outro tipo de cartão, precisaríamos criar ainda outros métodos.
  - Isso faria com que a classe tivesse diversos métodos, um para cada tipo de cartão. Imagine se existissem 10 tipos de cartões.
- Além disso, alguns cartões possuem lógicas diferentes entre si.
  - Isso geraria métodos específicos para cada um.

Polimorfismo de Subtipagem

- Para reutilizar código, podemos modelar os diversos tipos de cartões.

```java
public abstract class Cartao {
    public Cliente cliente;
    public Conta conta;

    public abstract void pagar(double valor);
}
```

```java
public class CartaoCredito extends Cartao {
    public String bandeira;
    public int cvv;

    public void pagar(double valor) {
        // Implementação
    }
}
```

```java
public class CartaoDebito extends Cartao {
    boolean exigeSenha;

    public void pagar(double valor) {
        // Implementação
    }
}
```

Subtipagem - Exemplo

- O que acontece quando chamamos um método de uma classe?
  - A subclasse verifica se ela tem ou não um método com este nome e com os mesmos parâmetros.
  - Se não tiver, a classe base imediatamente superior passa a ser a responsável pelo processamento da mensagem.

```
public static void main(String[] args) {
    Cliente c = new Cliente();
    CartaoCredito cc = new CartaoCredito();

    cc.setCliente(c);
}
```

Polimorfismo de Subtipagem

- É possível instanciar um objeto a partir de uma subclasse utilizando o tipo da classe base.

```
Cartao cd = new CartaoDebito();
```

- Devemos observar algumas questões:
  - Uma referência do tipo **Cartao** permite acessar apenas as características definidas em **Cartao** ou nas camadas mais internas.
  - Características das camadas mais externas não poderão ser chamadas.
  - Assim, o polimorfismo é um mecanismo de generalização.

Subtipagem - Ligação Tardia

- Ligação tardia (*late binding*) é a chave para o funcionamento do polimorfismo universal em Java.
- O compilador não gera código em tempo de compilação.
- Cada vez que se invoca um método de um objeto, o compilador gera código para verificar qual método deve ser chamado.

```
CartaoCredito cc = new CartaoCredito();
cc.pagar(50);
Cartao c = new CartaoCredito();
c.pagar(50);
```

Ligação tardia - Exemplo

```
public static void main(String[] args) {
    Cartao cartoes[] = new Cartao[2];

    CartaoCredito cc = new CartaoCredito();

    cartoes[0] = cc;
    cartoes[1] = new CartaoDebito();

    for (int i = 0; i < cartoes.length; i++) {
        cartoes[i].pagar(100);
    }
}
```

Upcasting e Downcasting

- Quando uma referência de uma classe recebe um objeto de suas subclasses ocorre um **upcast**.
  - Coerção de um tipo para outro mais genérico.
  - Permitido quando existe relação de herança.
  - Note que o *upcasting* é uma operação implícita.

```
Cartao c1 = new CartaoCredito(); // upcast de instância
CartaoDebito d = new CartaoDebito();
Cartao c2 = d; // upcast de referência
```

- Se for necessário realizar operações específicas de subclasse, pode ser realizado o downcasting.
  - Esta operação deve ser explicitamente indicada.

```
CartaoCredito cc = (CartaoCredito) c1; // downcasting
```

Polimorfismo Paramétrico

- Funcionalidade introduzida no Java 5.
- Mecanismos para criar tipos parametrizados.
  - Permitem definir classes ou métodos capazes de funcionar com uma variedade de tipos diferentes.
  - O tipo a ser utilizado é fornecido na instanciação.
- Apresenta uma forma que torna a linguagem mais expressiva, mantendo segurança de tipos.
  - Em Java é chamado de **generics**.
  - Similar aos **templates** do C++.

Como funcionava antes?

- Criação de uma lista de objetos utilizando a classe ArrayList.
  - Gostaria de armazenar apenas valores Integer.
  - Este código provocará erro em tempo de execução.
  - Tipos inválidos podem ser inseridos e necessidade de *cast* para obter a informação.

```
public static void main(String[] args) {
     ArrayList lista = new ArrayList();

    lista.add(new Integer(0));
    lista.add("1");
    lista.add(new Integer(2));

    Integer i1 = (Integer) lista.get(0);
    Integer i2 = (Integer) lista.get(1);
}
```

Como funciona agora?

- O uso dos *genéricos* corrige este problema.
  - Indicação de que os elementos devem possuir um tipo específico.
  - Produz um erro de compilação ao tentar inserir uma informação com tipo incorreto.
  - *Casts* não são mais necessários para obter dados.

```
public static void main(String[] args) {
    ArrayList<Integer> lista = new ArrayList<Integer>();

    lista.add(new Integer(0));
    lista.add("1"); // erro de compilação
    lista.add(new Integer(2));

    Integer i1 = lista.get(0);
    Integer i2 = lista.get(1);
}
```

Operador *diamond*

- A partir do Java 7 foi introduzido o operador <> (diamond), que simplifica a instanciação de tipos genéricos.
  - Isto é possível devido ao mecanismo de inferência de tipos.

```
// Instanciação tradicional de genéricos
ArrayList<Integer> lista1 = new ArrayList<Integer>();
// Instanciação de genéricos com inferência de tipos
ArrayList<Integer> lista2 = new ArrayList<>();
```

Definindo uma classe genérica

- Classes ou tipos genéricos possibilitam prover funcionalidades comuns que podem ser empregadas com diferentes tipos de dados.
  - Descrevem conceitos independentes dos tipos envolvidos, como acontece em estruturas lista, pilha ou árvore.
- Uma classe genérica pode ser criada com a inclusão, logo após seu nome, da lista de parâmetros de tipo.
  - No corpo da classe, os parâmetros de tipo podem ser usados na declaração de campos, nos parâmetros de métodos e nos tipos de retorno.

Definindo uma classe Genérica

```java
public class DBGen<T> {
    private int id;
    private T valor;

    private DBGen(int id, T valor) {
        this.id = id;
        this.valor = valor;
    }

    public int getId() { return id; }

    public T getValor() { return valor; }
}
```

Métodos genéricos

- De forma similar às classes, também é possível criar um método genérico.

```
public static <T> boolean equivalente(T a, T b) {
    return a.equals(b);
}
```

- O método funciona com qualquer tipo.

```
public static void main(String[] args) {
    Integer a = 10;
    Integer b = 10;
    if (MetodoGen.equivalente(a, b)) {
        System.out.println("São iguais");
    }
    else {
        System.out.println("São diferentes");
    }
}
```

Resumo - Polimorfismo Universal

- Subtipagem (*late binding* e *overriding*)
  - A partir das definições de herança é possível reescrever métodos e utilizar uma referência a partir da classe base.
- Paramétrico (*generics*)
  - Permite a definição de classes e métodos com tipos genéricos, preservando a segurança de tipos.

Considerações Finais

- Nesta aula estudamos outro pilar da orientação a objetos, o **polimorfismo**.
- Java possui dois tipos de polimorfismo.
  - Ad hoc: sobrecarga e coerção.
  - Universal: subtipagem e paramétrico.
- Permite definir múltiplas implementações para determinado tipo, sem perder as garantias de segurança em tempo de compilação.