Introdução

Nesse módulo, nós estendemos mais nosso conhecimento sobre herança e como podemos criar objetos que se comportam dinamicamente.

Além disso, aprendemos a criar métodos que sabem tratar diferentes tipos de dados por meio do polimorfismo.

Polimorfismo

Polimorfismo ("múltiplas formas") é um conceito da orientação a objetos que permite ao programador criar diferentes comportamentos para um mesmo objeto ou método. Existem quatro tipos de polimorfismo: Sobrecarga, Coerção, Paramétrico e Subtipagem. Os dois primeiros podem ser categorizados como polimorfismo "ad hoc" e os dois últimos, como "universal".

Polimorfismo Universal

Paramétrico

O polimorfismo paramétrico permite criar métodos que conseguem lidar com diferentes tipos de argumentos:

```
import java.util.ArrayList;
import java.util.List;

public class Zoologico {
    private List<Animal>  animais = new ArrayList<>();

    public void cadastrarAnimal(Animal novoAnimal) {
        animais.add(novoAnimal);
    }

    public void listarAnimais() {
        System.out.println("Há " + animais.size() + " cadastrados:
");

        for (Animal animal :  animais) {
            System.out.println(animal.getEspecie());
        }
    }
}
```

O método "**cadastrarAnimal**" pode receber qualquer objeto que seja do tipo "**Animal**", como "**Cachorro**", "**Passaro**" e "**Tubarao**":

```
public class Animal {
    private String especie;
    private String formaDeLocomocao;

    public Animal(String especie, String formaDeLocomocao) {
        this.especie = especie;
        this.formaDeLocomocao = formaDeLocomocao;
    }

    public String getEspecie() {
        return especie;
    }

    public String getFormaDeLocomocao() {
        return formaDeLocomocao;
    }
}

public class Cachorro extends Animal {
    public Cachorro() {
        super("Cachorro", "Cachorros andam");
    }

    public String latir() {
        return "Wooooof! Wooooooof!";
    }
}

public class Passaro extends Animal {
    public Passaro() {
        super("Pássaro", "Pássaros voam");
    }
}

public class Tubarao extends Animal {
    public Tubarao() {
        super("Tubarão","Tubarões nadam");
    }
}
```

Dessa forma, podemos invocar o método "**cadastrarAnimal**" da classe "**Zoologico**" passando cada um dos animais criados:

```
Zoologico zoologico =  new Zoologico();

zoologico.cadastrarAnimal( new Tubarao());
zoologico.cadastrarAnimal( new Cachorro());
zoologico.cadastrarAnimal( new Passaro());

zoologico.listarAnimais();

// Há 3 cadastrados:
// Tubarão
// Cachorro
// Pássaro
```

Subtipagem

É o tipo de polimorfismos que já utilizamos nas classes "**Animal**", "**Cachorro**", "**Passaro**" e "**Tubarao**". Nesse caso, podemos criar uma lista de animais de diferentes tipos:

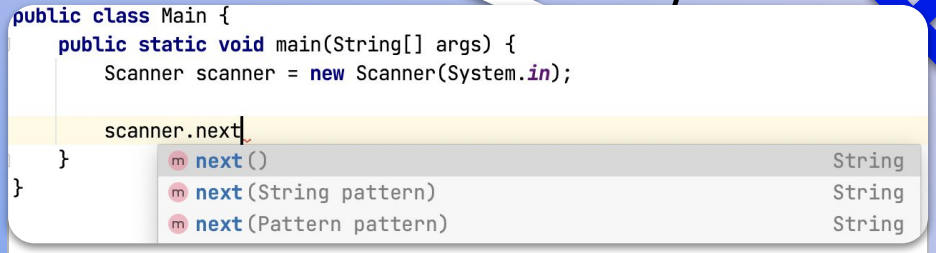
```
List<Animal> animais =  new ArrayList<>();

animais.add( new Passaro());
animais.add( new Tubarao());
animais.add( new Cachorro());

for (Animal animal : animais) {
   System.out.println(animal.getFormaDeLocomocao());
}
// Pássaros voam
// Tubarões nadam
// Cachorros andam
```

Polimorfismo Ad Hoc

Sobrecarga

Polimorfismo de sobrecarga é usado quando queremos definir comportamentos diferentes para um método baseado no tipo dos argumentos que ele recebe.

A linguagem Java implementa esse tipo de polimorfismo em algumas de suas classes, como a "Scanner", onde o método "**next**" tem diferentes comportamentos:



Na imagem acima, temos três implementações do **método** "**next**": uma que não recebe argumentos, outra que recebe um argumento do tipo "**String**" e outra do tipo "**Pattern**".

O mesmo acontece para o método "**println**", onde podemos passar diferentes tipos de dados como argumento:

É por isso que não nos preocupamos em imprimir textos ou números no console.

Podemos criar esse tipo de polimorfismo ao criar algumas vezes o mesmo método, alterando apenas os tipos dos seus argumentos:

```
public class Pessoa {
    public void falar(String nome) {
        System.out.println("Meu nome é " + nome);
    }

    public void falar(int idade) {
        System.out.println("Tenho " + idade + " anos de
idade");
    }
}
```

No caso acima a classe “**Pessoa**” possui duas implementações para o método "**falar**", onde uma recebe um **String** e outro um inteiro. O comportamento será diferente de acordo com o tipo do argumento:

```
Pessoa pessoa = new Pessoa();
pessoa.falar("Lucas"); // Meu nome é Lucas
pessoa.falar(25); // Tenho 25 anos de idade
```

Coerção

É um tipo de polimorfismo menos utilizado, mas que permite fazermos **transformações dos dados**. É muito comum em dados numéricos:

```
double temperatura = 23;
```

No caso acima temos uma variável do tipo "**double**", embora o valor atribuído seja do tipo inteiro. No momento da compilação deste código, o valor inteiro será convertido para um "**double**". Podemos fazer explicitamente a coerção ao aplicar o "**casting**" das classes:

```
int temperatura = (int) 22.3;
System.out.println(temperatura);  // 22
```

O casting também se aplica em cenários de herança:

```
public class Animal {
    private String nome;
    private String formaDeLocomocao;

    public Animal(String formaDeLocomocao) {
        this.formaDeLocomocao = formaDeLocomocao;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getFormaDeLocomocao() {
        return formaDeLocomocao;
    }
}

public class Cachorro extends Animal {
    public Cachorro() {
        super("Cachorros andam");
    }

    public String latir() {
        return "Wooooof! Wooooooof!";
    }
}
```

No cenário anterior, temos as classes **Animal** e **Cachorro**, sendo que **Cachorro** é uma subclasse de **Animal**.

Dessa forma, podemos instanciar um objeto de **Animal** sendo seu valor um **Cachorro**:

```
Animal animal = new Cachorro();
```

O objeto "**animal**" só terá disponível os métodos que foram definidos na classe "**Animal**". Dessa forma, o método "**latir**" de **Cachorro** está inacessível. Mas podemos converter o "**animal**" para uma instância de "**Cachorro**" usando o ***casting***, fazendo com que o método "**latir**" fique acessível:

```
Cachorro cachorro = (Cachorro) animal;
System.out.println(cachorro.latir());  // Wooooof! Wooooooof!
```

Interfaces

Interfaces é uma forma de se definir métodos que devem ser implementados por classes que implementam.

Num sistema bancário há diferentes tipos de contas: salário, conta corrente e poupança. Como todas essas contas devem ser capazes de calcular o saldo do cliente, podemos criar uma interface que define o método "**calcularSaldo**":

```
public interface Conta {
    double calcularSaldo();
}
```

Uma interface é semelhante à uma classe, exceto que não possui atributos e seus métodos não possuem implementação.

A implementação dos métodos fica a cargo das classes que implementam a interface. No caso do sistema bancário, as classes "**ContaCorrente**", "**ContaSalario**" e "**ContaPoupanca**" são responsáveis por implementar o método, já que implementam a interface:

```java
import java.util.List;

public class ContaCorrente implements Conta {
   private List<Depositos> depositos;

   @Override
   public double calcularSaldo() {
       double valor = 0.0;
       for(Depositos depositos : depositos) {
           valor += depositos.getValor();
       }
       return valor;
   }
}

public class ContaSalario implements Conta {
   private double saldo;

   @Override
   public double calcularSaldo() {
       return saldo;
   }
}

public class ContaPoupanca implements Conta {
   private double saldo;
   private double taxaDeJuros;

   @Override
   public double calcularSaldo() {
       return saldo * (taxaDeJuros + 1);
   }
}
```


Polimorfismo de subtipagem e paramétrico também se aplicam para interfaces:

```java
List<Conta> contas =  new ArrayList<>();
contas.add(new ContaCorrente());
contas.add(new ContaSalario());
contas.add(new ContaPoupanca());

for (Conta conta : contas) {
   System.out.println(conta.calcularSaldo());
}
```