Aula 05

Herança

O conceito de `herança` surge a partir da generalização que podemos fazer de uma `classe`, e, a partir dessa classe, podemos criar outras mais específicas.

Por exemplo, podemos ter uma classe para `Pessoa Física` e outra para `Pessoa Jurídica`. Ambas terão alguns atributos e métodos em comum. Exemplo:

```
public class PessoaFisica{
    public String nome;
    public int idade;
    public Conta contaBancaria;
    public String cpf;

    //... getters & setters
}

public class PessoaJuridica{
    public String nome;
    public int idade;
    public Conta contaBancaria;
    public String cnpj;

    //... getters & setters
}
```

Existe alguma forma de declarar os mesmos atributos e métodos somente uma vez? Sim, através da `herança`.

Podemos então criar uma classe chamada Pessoa, a qual terá todos os atributos e métodos comuns entre Pessoa Física e Pessoa Jurídica. Essas duas classes terão apenas de declarar atributos e métodos próprios. Por exemplo, toda pessoa (física ou jurídica) possui:

- Nome;
- Idade;
- Conta Bancária.

Uma Pessoa Física, além desses três atributos acima, terá um próprio seu, o CPF. Por outro lado, uma Pessoa Jurídica terá um atributo somente seu, chamado CNPJ. Portanto, podemos ter uma `classe mãe` chamada `Pessoa`, e `subclasses` chamadas `PessoaFisica` e `PessoaJuridica`, as quais herdarão os atributos e métodos de `Pessoa`. Vejamos como isso é feito:

```
public class Pessoa{
    public String nome;
    public int idade;
    public Conta contaBancaria;

    public Pessoa(){}

    public Pessoa(String nome, int idade, Conta contaBancaria){
        this.nome = nome;
        this.idade = idade;
        this.contaBancaria = contaBancaria;
    }

    public String getNome(){
        return nome;
    }

    //... getters & setters
}

public class PessoaFisica extends Pessoa{
    public String cpf;

    public PessoaFisica(){}

    public PessoaFisica(String cpf, String nome, int idade, Conta contaBancaria
        super(nome, idade, contaBancaria);
        this.cpf = cpf;
    }

    public String getCpf(){
        return cpf;
    }
```

```
        public void setCPF(String cpf){
            this.cpf = cpf;
        }
    }

    public class PessoaJuridica extends Pessoa{
        public String cnpj;

        public PessoaJuridica(){}

        public PessoaJuridica(String cnpj, String nome, int idade, Conta contaBanca
            super(nome, idade, contaBancaria);
            this.cnpj = cnpj;
        }

        public String getCnpj(){
            return cnpj;
        }

        public void setCnpj(String cnpj){
            this.cnpj = cnpj;
        }
    }
```

No exemplo acima foram criadas três classes: `Pessoa`, `PessoaFisica` e `PessoaJuridica`. A primeira é a classe mãe, ou em termos mais técnicos a `superclasse`. As duas seguintes são as classes filhas ou `subclasses`. Perceba que na criação de `PessoaFisica` e `PessoaJuridica` temos a palavra reservada `extends`. A partir dessa palavra é que dizemos ao compilador que essa classe é subclasse de outra. No nosso caso, são subclasses de `Pessoa`. Dessa forma, todos os atributos e métodos de `Pessoa` serão herdados por suas subclasses. Exemplo:

```
public class TesteHeranca{
    public static void main(String[] args) {
        PessoaFisica pf = new PessoaFisica(); // utilizando o construtor padrão

        // A classe PessoaFisica não tem os métodos getNome ou setNome nela, mas he
        pf.setNome("Fulano");
        System.out.println("O nome da pessoa é : " + pf.getNome());

        // O mesmo acontece para PessoaJuridica
        PessoaJuridica pj = new PessoaJuridica();
        pj.setNome("Faculdade Uninassau");
```

```
        System.out.println("O nome da empresa é: " + pj.getNome());
    }
}
```

Exercícios de Herança

1. Crie uma superclasse chamada `ContaBancaria`. Depois crie duas novas classes chamadas `ContaCorrente` e `ContaPoupanca`, as quais deverão ser subclasses de `ContaBancaria`. Crie uma classe com o método `main` para instanciar objetos das 3 classes, utilizando todos os seus métodos implementados.
2. Crie uma classe chamada `Pessoa`, com atributos e métodos comuns a qualquer tipo de pessoa. Depois crie as subclasses `PessoaFisica` e `PessoaJuridica`, ambas subclasses de `Pessoa`. Por fim, crie outra classe chamada `Funcionario`, a qual deverá ser subclasse de `PessoaFisica`

Polimorfismo

Grosso modo, o `polimorfismo` é uma técnica de *sobrescrever* os métodos de uma `superclasse`. O uso de `polimorfismo` pressupõe duas condições:

1. A existência de herança, e
2. A redefinição de métodos em todas as classes (as que estão na estrutura da herança).

Não confunda **sobrecarga** com **polimorfismo**!!!

> **Sobrecarga**: métodos com mesmo nome, porém de assinatura diferente, implementados na mesma classe.

> **Polimorfismo**: métodos com mesmo nome e assinatura, porém implementados em classes diferentes, no contexto de herança.

Basicamente o polimorfismo permite a existência de um método generalizado, o qual pode ser modificado nas subclasses, para se comportar de forma mais específica. A

modificação poderá ser uma **sobrescrita** (conhecida como *override*), ou sobrescrita e **sobrecarga**. Exemplo:

```
public class Animal{
    // atributos ...

    public void mover(){
        // como o objeto se move
    }
}

public class Cachorro extends Animal{
    // atributos ...

    // override
    public void mover(){
        // correr
    }

    // sorbecarga
    public void mover(boolean isAgua){
        if(isAgua){
            // nadar
        }
    }
}

public class Passaro extends Animal{
    // atributos ...

    //override
    public void mover(){
        //voar
    }
}

public class Peixe extends Animal{
    // atributos ...

    public void mover(){
        //nadar
    }
}
```

E se eu estiver escrevendo o método de uma `superclasse` que não deve ser sobrescrito? Eu posso impedir a sobrescrita? Sim, através da palavra reservada `final`. Ex.:

```
public class Animal{
    // atributos ...

    public void mover(){
        // mover
    }

    public final void comer(){
        // esse método será igual para todas as subclasses
    }
}
```

Com a palavra reservada `final`, é possível impedir que uma classe tenha subclasses. Exemplo:

```
public final class Catiorinho{
    // atributos ...
}
```

A palavra reservada `final` também pode ser utilizada em atributos, de forma a transformá-los em constantes. Exemplo:

```
public class Pessoa{
    // atributos ...
    private final int idadeMorte = 100;
}
```

Você deve ter percebido que pela primeira vez eu utilizei `private` em vez de `public` na declaração do atributo. Após os exercícios, vamos entender o porquê.

Exercícios

1. Faça a implementação das classes mostradas nos exemplos, ou seja, `Animal`, `Cachorro`, `Passaro` e `Peixe`, acrescentando pelo menos mais um método para

sofrer `polimorfismo`. Dica: no método você poderá apenas imprimir no console alguma frase; ex.: "O cachorro está correndo".

Encapsulamento

O `encapsulamento` é um princípio de *design* de código geralmente ligado à programação orientada a objetos. A partir desse princípio as classes são implementadas de forma a **esconder** algumas de suas funcionalidades e funcionamentos. Isso possibilita que modificações no sistema possam ser feitas de maneira mais cirúrgica, sem que uma funcionalidade esteja espalhada por diversas partes do sistema.

Quando usar?

Basicamente sempre, pois a forma como as classes e objetos conversam uns com os outros deve estar sempre isolada da forma como executam o que se propuseram a fazer. Além disso aumentamos a segurança de nosso código, impedindo potenciais erros e outras falhas de acontecerem.

São quatro os tipos de encapsulamento: **default**, **public**, **private** e **protected**.

Default

É o encapsulamento padrão, que ocorre quando nenhum tipo é definido. Por exemplo:

```
    class Exemplo {
    int numero;
}
```

Um atributo ou método `default` pode ser acessado por todas as classes que estiverem no **mesmo pacote** da classe que aquele atributo/método.

Public

O mais simples: faz com que o atributo/método esteja visível por todas as classes do projeto.

Private

Um atributo ou método declarado como `private` só é visível dentro da classe. Nem mesmo o `objeto` instanciado tem acesso.

Protected

Quase a mesma coisa de `default`. A diferença é que um atributo/método `protected` estará visível para as subclasses, mesmo que estejam em outro pacote.

Exercícios

1. Faça um programa em Java com as seguintes classes:
    - `Pessoa`
        - Atributos
            - nome
            - endereço
            - telefone
            - email
            - conta corrente ou conta poupança
        - Métodos
            - *getters* e *setters*
    - `PessoaJuridica`
        - Subclasse de Pessoa
        - Atributos
            - tipo da empresa (empreiteira, clínica, etc.)
            - cnpj
            - nome fantasia (esse é o nome conhecido pelos clientes)
        - Métodos
            - *getters* e *setters*
    - `PessoaFisica`
        - Subclasse de Pessoa
        - Atributos
            - cpf
            - cnh
            - rg
        - Métodos
            - *getters* e *setters*

- `Funcionario`
  - Subclasse de PessoaFisica
  - Atributos
    - tipo de funcionário (chefe, arquiteto, engenheiro, pedreiro, secretário, etc.)
    - empresa onde trabalha (dica: deve ser da classe PessoaJuridica)
    - código de funcionário
    - chefe (dica: deve ser do tipo Funcionario)
    - subordinado (dica: deve ser do tipo Funcionario)
  - Métodos
    - *getters* e *setters*
- `ContaBancaria`
  - Atributos
    - titular (dica: deve ser do tipo PessoaFisica)
    - banco (dica: deve ser do tipo PessoaJuridica)
    - agência
    - número
  - Métodos
    - *getters* e *setters*
- `ContaPoupanca`
  - Subclasse de ContaBancaria
  - Atributos
    - juros
  - Métodos
    - *getters* e *setters*
    - aplicaJuros
- `Principal`
  - Classe que terá o método `main` para criar e manipular objetos das demais classes.