FACULDADES DA INDÚSTRIA SISTEMA FIEP	**2ª APS**	Atividade Prática Supervisionada **Conceitos de Orientação a Objetos**
Curso: **Bacharelado em Sistemas de Informação / Engenharia de Software**		
Disciplina: **Programação Orientada a Objetos**		Período: **3º / 4º**
Professor: **Eunelson José da Silva Júnior**		Data: **07/10/2020**

Estudante: Clístenes Grizafis Bento

Hoje, a maioria das linguagens de programação são orientadas a objetos como Java, C#, Python e C++ e, apesar de terem algumas diferenças na implementação, todas seguem os mesmos princípios e conceitos. Muitos programadores, apesar de utilizarem linguagens orientadas a objetos, não sabem utilizar alguns dos principais conceitos desse paradigma orientado a objetos e, por isso, desenvolvem sistemas com alguns erros conceituais e acabam escrevendo mais código que o necessário, não conseguindo reutilizar o código como seria possível.

Já abordamos, durante o 1º bimestre, os conceitos de Classes, Objetos, Associação, Encapsulamento e Construtores. Agora é hora de aprofundar os estudos.

Esta pesquisa deverá lhe ajudar a compreender os próximos conceitos de orientação a objetos. Durante as próximas aulas iremos trabalhar com diversos exemplos de implementação destes conceitos de orientação a objetos em Java.

Pesquise sobre cada um dos tópicos abaixo, estudando os conceitos e exemplos de implementação, a fim de compreender seu funcionamento e suas aplicações.

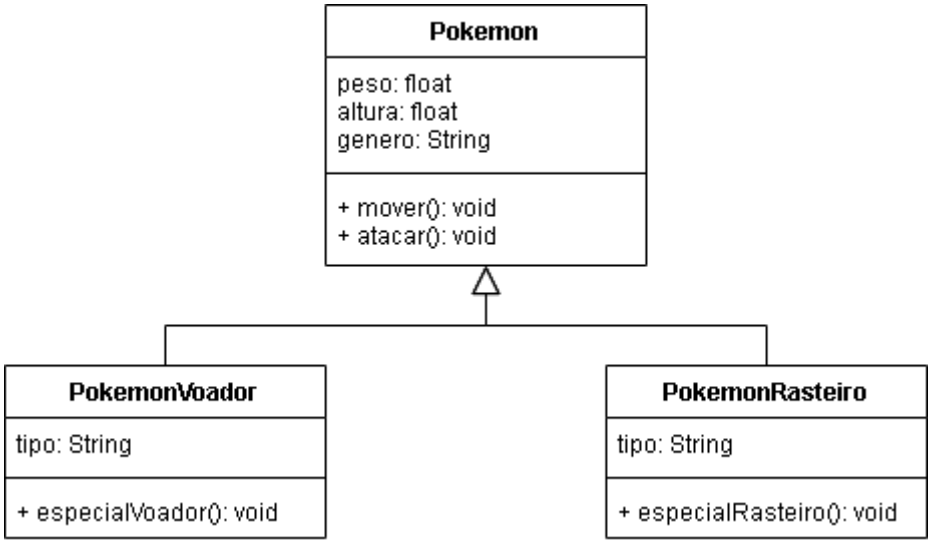
Tópico 01 – Herança, reutilização de código e reescrita de métodos.

A herança em Programação Orientada a Objetos (POO) é uma técnica utilizada para aproveitar atributos e métodos de uma classe genérica em classes mais

específicas, evitando assim ter que reescrevê-los. Além te permitir a reutilização dos códigos em diversas subclasses.

Exemplo: Podemos ter uma classe genérica chamada “Pokemon” que possuí os seguintes atributos: nome, peso, altura, gênero; e os seguintes métodos: mover e atacar. Agora podemos ter duas outras classes, uma chamada “PokemonVoador” e outra “PokemonRasteito”. Note que ambas as classes adicionadas posteriormente irão obter os mesmos atributos e métodos da classe “Pokemon”. Então em POO poderemos fazer com que as classes “PokemonVoador” e “PokemonRasteiro” sejam subclasses de “Pokemon”, herdandos seus atributos e métodos.

No diagrama de classes ficaria assim:



Em linguagem de programação Java seria necessário criar a classe “mãe” chamada “Pokemon” junto com seus atributos e métodos e posteriormente criar as subclasses, onde ao cria-las seria necessário adicionar a palavra-chave *extends* seguido do nome da classe em que está herdando.

Exemplo: public class PokemonVoador extends Pokemon{

}

Assim a classe “PokemonVoador” irá herdar todos os atributos e métodos de “Pokemon”.

Tópico 02 – Polimorfismo (abstrações de comportamentos).

O polimorfismo, como o nome diz muda a forma de algo de várias maneiras diferentes. Em POO existem pelo menos seis tipos de polimorfismo, sendo os mais utilizados o de “sobreposição” e o de “sobrecarga” aplicados sobre os métodos das classes.

O polimorfismo de “sobreposição” serve para alterar o comportamento de um método com a mesma assinatura herdado de uma classe mãe:

Exemplo: Seguindo as classes criadas no tópico 01, temos que na classe “Pokemon” o ataque seja “batendo” e na classe “PokemonVoador” o ataque seja bicando, nesse caso é nesse caso é necessário sobrescrever o método herdado. Em Java coloca-se a palavra *@override* sobre o método que será alterado, repete-se a assinatura e escreve-se as alterações dentro do método:

```
@override
public void atacar(){

    System.out.println(“Bicando”);

}
```

O polimorfismo de “sobrecarga” ocorre quando se tem o mesmo método na mesma classe com assinaturas diferentes.

Exemplo:

```
public void atacar(){
    ....
}
public void atacar(float cansaco){
    ....
}
public void atacar(int hits){
    ....
}
```

Tópico 03 – Modificadores de acesso (public, private, protected e sem modificador)

Os modificadores de acesso servem para definir o acesso de atributos e métodos de uma determinada classe sem estar programando esta classe, sendo divididos em public, private, protected e sem modificador.

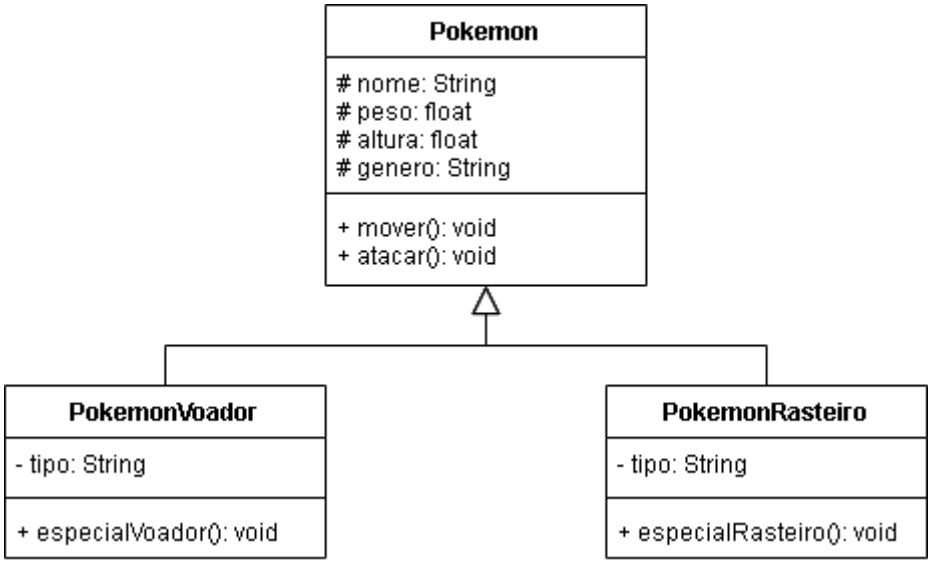
Public: Esse modificador permite o acesso externo aos seus atributos e métodos de qualquer outra classe. Em UML utiliza-se o símbolo “+”.

Private: Esse modificador restringe qualquer acesso externo ao atributo ou método que o tem, nem mesmo as subclasses que os herdam podem acessá-los. Em UML utiliza-se o símbolo “-“.

Protected: Esse modificador permite que suas subclasses acessem aos seus atributos e métodos. Em UML utiliza-se o símbolo “#“.

Sem modificador: Permite que qualquer classe do mesmo pacote acesse seus atributos e métodos. Em UML utiliza-se o símbolo “~“.

O exemplo do tópico 01 em diagrama e classes utilizando modificadores de acesso ficariam assim:



Nesse exemplo os atributos da classe “Pokemon” estão como *protected* os atributos de suas subclasses estão como *private* e todos os métodos como *public*. Em POO normalmente, devido ao encapsulamento, utiliza-se os atributos como *private* e os métodos como *public*.

Implementação: a implementação da classe “Pokemon” ficaria:

```
public class Pokemon{
    protected String nome;
```

```
        protected float peso;
        protected float altura;
        protected String gerero;

        public void mover(){
                ...
        }
        public void atacar(){
                ...
        }
}
```

Tópico 04 – Métodos estáticos e atributos estáticos

Os métodos e atributos estáticos são aqueles que utilizados pela classe e são comuns a todos os objetos.

Exemplo: Se quisermos que a classe “Pokemon” contabilize cada novo pokemon cadastrado, podemos criar um atributo estático chamado numPokemon e um método estático AddPokemon:

```
public class Pokemon{
        protected String nome;
        protected float peso;
        protected float altura;
        protected String gerero;
        private static int numPokemon;

        public Pokemon(){
                addPokemon();
        }
        Public static void addPokemon(){
                Pokemon.numPokemon++;
        }
        public void mover(){
```

```
            ...
        }
        public void atacar(){
            ...
        }
}
```

Nesse caso para cada novo Pokémon instanciado o numPokemon será acrescentado uma unidade.

Tópico 05 – Classes abstratas e métodos abstratos.

Classe abstrata é uma classe quem não pode ser instanciada e nem ser herdada de outra classe, mas pode ter subclasses.

A classe “Pokemon” do nosso exemplo poderia ser uma classe abstrata, pois ela não tem “mãe” apenas “filhas” e não faz sentido instanciá-la , servindo apenas de arcabouço para suas “filhas”.

Para classe abstrata coloca-se a palavra-chave *abstract* na construção da classe.

Exemplo:

```
public abstract class Pokemon{
        protected String nome;
        protected float peso;
        protected float altura;
        protected String gerero;
        private static int numPokemon;

        public Pokemon(){
            addPokemon();
        }
        Public static void addPokemon(){
            Pokemon.numPokemon++;
        }
```

```
        public void mover(){
                ...
        }
        public void atacar(){
                ...
        }
}
```

Os métodos abstratos não possuem código implementado e será implementado em alguma classe descendente à classe em que foi criado, através do polimorfismo de sobreposição.

Exemplo: Como a classe abstrata “Pokemon” não será instanciada, e para cada uma de suas filhas os métodos “mover” e “atacar” são diferentes, pode-se criá-los como abstratos e alterá-los em suas filhas.

```
public abstract class Pokemon{
        protected String nome;
        protected float peso;
        protected float altura;
        protected String gerero;
        private static int numPokemon;

        public Pokemon(){
                addPokemon();
        }
        Public static void addPokemon(){
                Pokemon.numPokemon++;
        }
        public abstract void mover();
        public abstract void atacar();
}
```

Tópico 06 – Interfaces (em orientação a objetos)

Uma interface é a relação entre a classe e o mundo exterior. Não possui atributos, apenas métodos abstratos. Tomando como exemplo um controle remoto uma classe seria todas as características e comportamentos do controle e a interface seria os botões que podem ser apertados associados a algum comportamento da classe. Em sua implementação é necessário colocar o nome interface e na classe que a recebe utiliza-se a palavra-chave *implements*.

Exemplo: Se quiséssemos criar uma interface chamada “GritarPokemon” para nossa classe “PokemonVoador” onde nessa interface houvesse os métodos “mover” e “atacar” ela ficaria assim:

```
public interface GritarPokemon{

    public abstract void mover();
    public abstract void atacar();
}
```

E a classe “PokemonVoador” teria que ser criada da seguinte maneira:

```
public class PokemonVoador extends Pokemon implements GritarPokemon{
    ...
}
```

Tópico 07 – Pacotes (organização de classes e interfaces)

Um pacote em POO é um diretório onde as classes e interfaces serão armazenadas no projeto. Os pacotes normalmente são utilizados para manter a organização do projeto, permitindo com que os códigos fiquem mais fácil de serem compreendidos por outros desenvolvedores e reutilizados em outros projetos.

Exemplo: Em nosso projeto criado até o momento, é possível colocar todas as classes e interfaces em um único pacote por se tratar de um projeto pequeno.

```
package pokedex.pokemon
public abstract class Pokemon{
```

```
        protected String nome;
        protected float peso;
        protected float altura;
        protected String gerero;
        private static int numPokemon;

        public Pokemon(){
                addPokemon();
        }
        Public static void addPokemon(){
                Pokemon.numPokemon++;
        }
        public abstract void mover();
        public abstract void atacar();
}
```

Nesse caso, os arquivos ficarão salvos em uma pasta chamada “pokedex” dentro da pasta “pokemon”.

Tópico 08 – Tratamentos de erros (Exceptions)

O tratamento de erros serve para que o programa não seja interrompido devido a algum erro que pode ocorrer, como por exemplo um saque negativo em uma conta bancária, ou dividir algum número por zero. Para esses tipos de situação é utilizado o manipulador de exceções, que pega algum erro apontado pelo sistema e o manipula. O bloco de código onde precisam de tratamento de erros é chamado de região protegida. Ele é indicado pelo comando *try*. Para associar o tratamento a um bloco de código que a manipulará, usa-se um ou mais comandos *catch*.

Exemplo: No caso do nosso projeto do Pokemon pode ocorrer de alguém tentar cadastrar uma altura no formato de letra, nesse casso é necessário tratar o erro que pode ocorrer.

```
public class PokemonVoador extends Pokemon implements GritarPokemon{
        ...
        public void setAltura(float altura){
```

```
            try{
                this.altura = altura;
            }
            catch{
                System.out.println(“altura inválida”);
            }
        }
}
```