SCC0504 - Programação Orientada a Objetos

Hierarquia de Classes

Prof.: Leonardo Tórtoro Pereira
leonardop@usp.br

Objetivos

- ➔ Revisar Classes e Objetos
- ➔ Entender o que é herança
- ➔ Apresentar a transitividade de propriedades para heranças em múltiplos níveis
- ➔ Como lidar com a necessidade de herança múltipla nas principais linguagens orientadas a objetos
- ➔ Apresentar o conceito de classe abstrata e sua aplicação
- ➔ Estudar um caso mais próximo da realidade

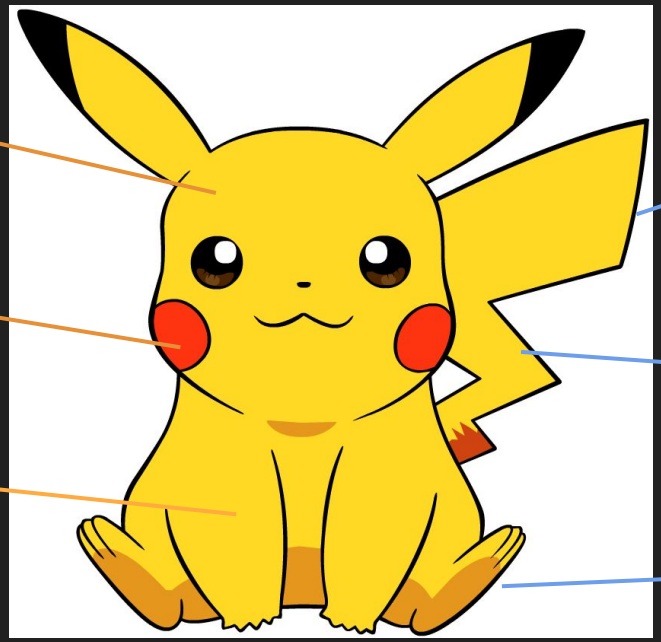
Revisão e Contextualização

Objeto = Características + Comportamento

Nome:
Pikachu

Tipo:
Elétrico

Cor:
Amarelo



Thunder()

TailWhip()

Run()

Objeto

Objeto = Características + Comportamento

normalmente são representadas como variáveis

```
skills = ["fire blast",
     "scratch"];

 sAttack = 300;


 hp = 280;


 status = "healthy"
```

Habilidades

Ataque Especial

HP

Estado

Os valores podem se alterar: poison, receber dano, aprender habilidade nova, paralisado.

Objeto

Objeto = Características + Comportamento

normalmente são representadas como funções

Poison

```
poison(target){
     target.sufferDamage(
          target.getHP() * 0.05);
     target.setStatus("poisoned");
}
```

Synthesis

```
synthesis(){
     self.recoverHP(self.maxHP*0.5);
}
```

Altera as variáveis de seus atributos e também de outros objetos!

Classe

- Alguns objetos podem ser agrupados em um mesmo tipo, pois possuem características e comportamentos em comum.
- Classes servem como um molde para a criação de objetos similares.

Classe

```
class WaterPokemon {
  int hp, atk, def;
  String type = "water";
  void takeDamage(amount,
enemy_type){
    if(enemy_type == "fire"){
      hp = hp - (amount/2);
    }
    else{
      hp = hp - amount;
    }
  }
}
```

```
class FirePokemon {
  int hp, atk, def;
  String type = "fire";
  void takeDamage(amount,
enemy_type){
    if(enemy_type == "water"){
      hp = hp - (amount * 2);
    }
    else{
      hp = hp - amount;
    }
  }
}
```

Classe

```
class PokemonWorld{
  void main(){
    WaterPokemon squirtle;
    FirePokemon charmander;
    squirtle.takeDamage(20, "fire");
  }
}
```

Vamos observar de novo as classes...

Classe

```
class WaterPokemon {
  int hp;
  String type = "water";
  void takeDamage(amount,
enemy_type){
    if(enemy_type == "fire"){
      hp = hp - (amount/2);
    }
    else{
      hp = hp - amount;
    }
  }
}
```

```
class FirePokemon {
  int hp;
  String type = "fire";
  void takeDamage(amount,
enemy_type){
    if(enemy_type == "water"){
      hp = hp - (amount * 2);
    }
    else{
      hp = hp - amount;
    }
  }
}
```

Contextualização

- ➔ É muito comum encontrarmos classes muito similares entre si, que necessitam apenas de alguns atributos ou métodos diferentes, mas mantém grande parte do resto.
- ➔ O mesmo acontece com TADs da programação estruturada.
- ➔ Re-escrever todas as partes em comum para uma nova classe/TAD e alterar apenas o que é necessário gera muito código extra, dificulta entendimento e reuso...

Contextualização

- ➔ Esse mesmo problema ficou claro na década de 80 pelos desenvolvedores.
- ➔ A maior reutilização de software era uma das melhores maneiras de aumentar a produtividade
- ➔ Os TADs eram as unidades ideais para reutilizar
- ➔ Mas as características e capacidades dos tipos existentes não eram adequadas para novo uso.

Contextualização

- O tipo requeria pelo menos algumas modificações, que podiam ser difíceis. Ou era necessário alterar todos os programas clientes.
- Todas as definições eram independentes e estavam no mesmo nível. Isso dificulta a estruturação do problema.
- Geralmente, os problemas possuem categorias de objetos relacionados, como “irmãos” (similares) e “pais e filhos” (algum tipo de subordinação).

Como Resolver?

Herança!

Herança

➔ Existem muitas semelhanças entre Pokémon de tipos diferentes:

Flareon : FirePokemon
- HP
- Attack
- Defense
- Special Attack
- Special Defense
- Speed

Vaporeon: WaterPokemon
- HP
- Attack
- Defense
- Special Attack
- Special Defense
- Speed

➔ Nesse caso, seria interessante ter uma classe "**Pokemon**".

Herança

Pokemon
- HP, Attack, Defense, Special Attack, Special Defense, Speed
- Tackle()

FirePokemon
- Ember()
- Flamethrower()

WaterPokemon
- Bubble()
- WaterGun()

Herança

Nesse caso, cada pokémon possui os **mesmos atributos**, mas cada tipo tem **habilidades diferentes**! As classes de cada tipo **herdam** da classe principal e se especializam.

Pokemon
- HP, Attack, Defense, Special Attack, Special Defense, Speed
- Tackle()

FirePokemon
- Ember()
- Flamethrower()

WaterPokemon
- Bubble()
- WaterGun()

Herança

A classe *Pokemon* é a chamada **classe base**, **superclasse** ou **classe pai/mãe**.

As classes *FirePokemon* e *WaterPokemon* são chamadas de **classe derivada**, **subclasse**, ou **classe filho/filha**

Pokemon
- HP, Attack, Defense, Special Attack, Special Defense, Speed
- Tackle()

FirePokemon
- Ember()
- Flamethrower()

WaterPokemon
- Bubble()
- WaterGun()

Herança

Quando ocorre a herança, as classes filhas como FirePokemon e WaterPokemon são **capazes de utilizar os atributos e métodos da classe mãe** Pokemon

Pokemon
- HP, Attack, Defense, Special Attack, Special Defense, Speed
- Tackle()

FirePokemon
- Ember()
- Flamethrower()

WaterPokemon
- Bubble()
- WaterGun()

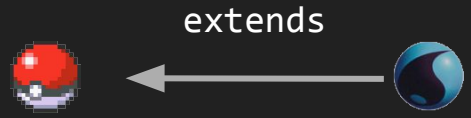
Herança

- *"A herança é uma forma de reutilização de software em que o programador cria uma classe que absorve dados e comportamentos de uma classe existente e os aprimora com novas capacidades. A reusabilidade de software economiza tempo durante o desenvolvimento de programa. Ela também encoraja a reutilização de softwares de alta qualidade já testados e depurados, o que aumenta a probabilidade de um sistema ser eficientemente implementado."*
  - DEITEL, H. M.; DEITEL, P. J. C++ Como Programar: 5 ed. São Paulo: Bookman, 2006. 1208 p.

Um pouco de código!

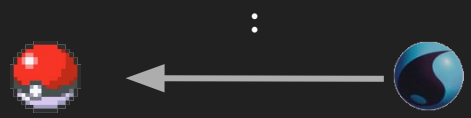
Herança

A palavra-chave extends é utilizada para definir herança em Java.



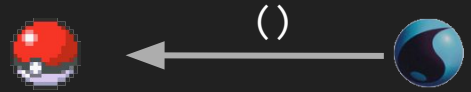
Pokemon WaterPokemon

O caractere “:” é utilizado para definir herança em C++ e C#



Pokemon WaterPokemon

Em Python, a classe mãe é colocada entre parênteses depois do nome da classe.



Pokemon WaterPokemon

Herança

- **Java**

```
class FirePokemon extends Pokemon{
    Métodos e Atributos}
```

- **C++**

```
class FirePokemon : public Pokemon{
    Métodos e Atributos}
```

- **C#**

```
class FirePokemon : Pokemon{
    Métodos e Atributos}
```

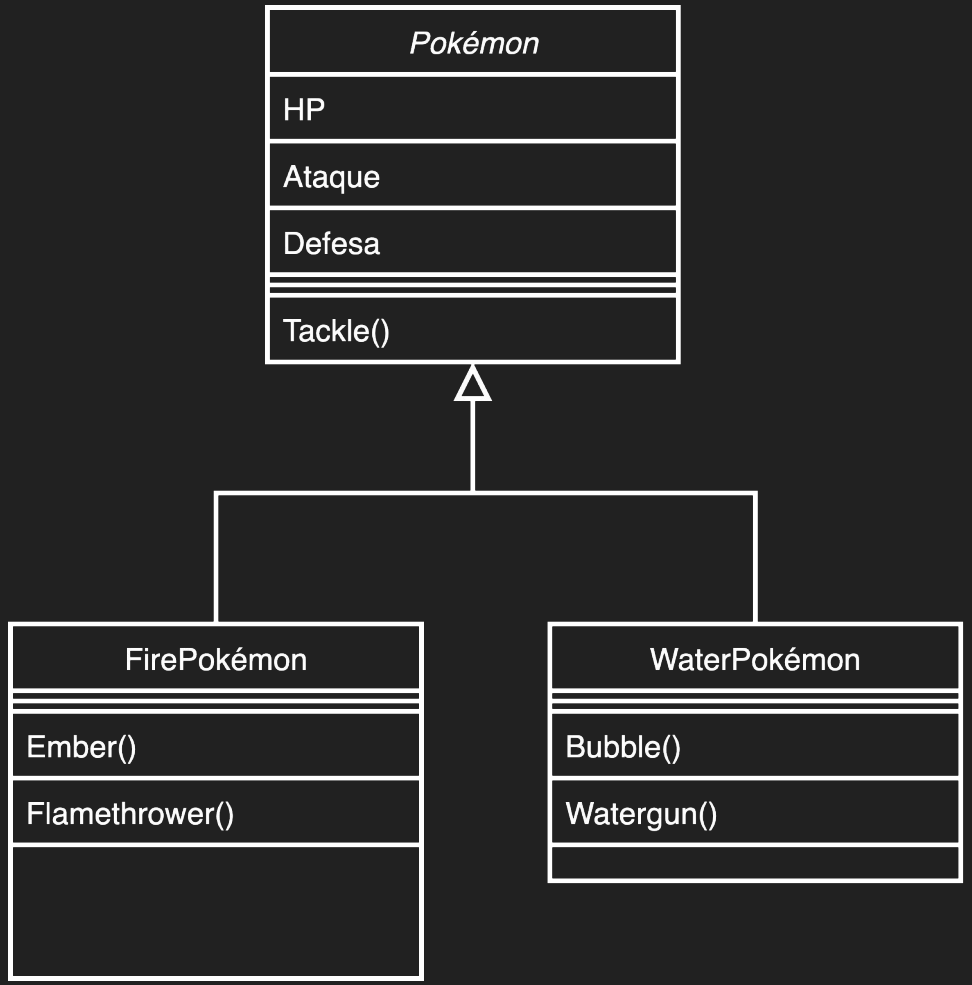
- **Python**

```
class FirePokemon(Pokemon):
    Métodos e Atributos
```

Herança

➔ **Java**

```
class FirePokemon extends Pokemon{
    “Métodos e Atributos”
    private string type = “Fire”
    public int Flamethrower(Pokémon target){
        private int power = 90;
        private int acc = 100;
        private int burnChance = 10;
        if(AccuracyCheck(acc)){
            target.ApplyDamage(power*atk, type);
            if(BurnCheck(10))
                target.ApplyStatus(“Burn”);
        }
    }
}
```


Pokémon
HP
Ataque
Defesa
Tackle()
FirePokémon
Ember()
Flamethrower()
WaterPokémon
Bubble()
Watergun()

E se eu precisar de mais uma herança?

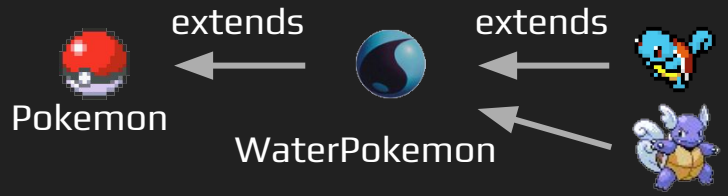
Herança

Os pokémons dentro de uma **mesma classe** podem ter diferenças entre si.

Cada pokémon possui **características únicas** que diferem de seu tipo como uma **imagem** diferente e **quais habilidades** podem aprender e em qual nível.

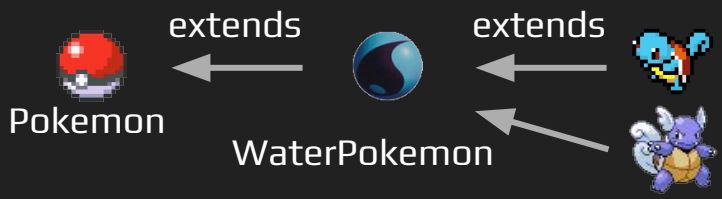
Transitividade da Herança

- ➔ Portanto, cada espécie de Pokémon deveria ter uma classe para si.
- ➔ As propriedades da classe pai são passadas para a filha, a “neta”, “bisneta”, e assim sucessivamente.
- ➔ Cada instância de Pokémon encontrada seria um objeto da classe do Pokémon específico, que por sua vez herda da classe de seu tipo, e que herda da classe base Pokémon.



Transitividade da Herança

- ➔ Nesse caso, a classe *Pokémon* é chamada da classe base **indireta** das classes Squirtle e Wartortle
- ➔ E *WaterPokémon* é a classe base **direta** das classes Squirtle e Wartortle



Herança

- **Java**

```
class Squirtle extends WaterPokemon{
    Métodos e Atributos}
```

- **C++**

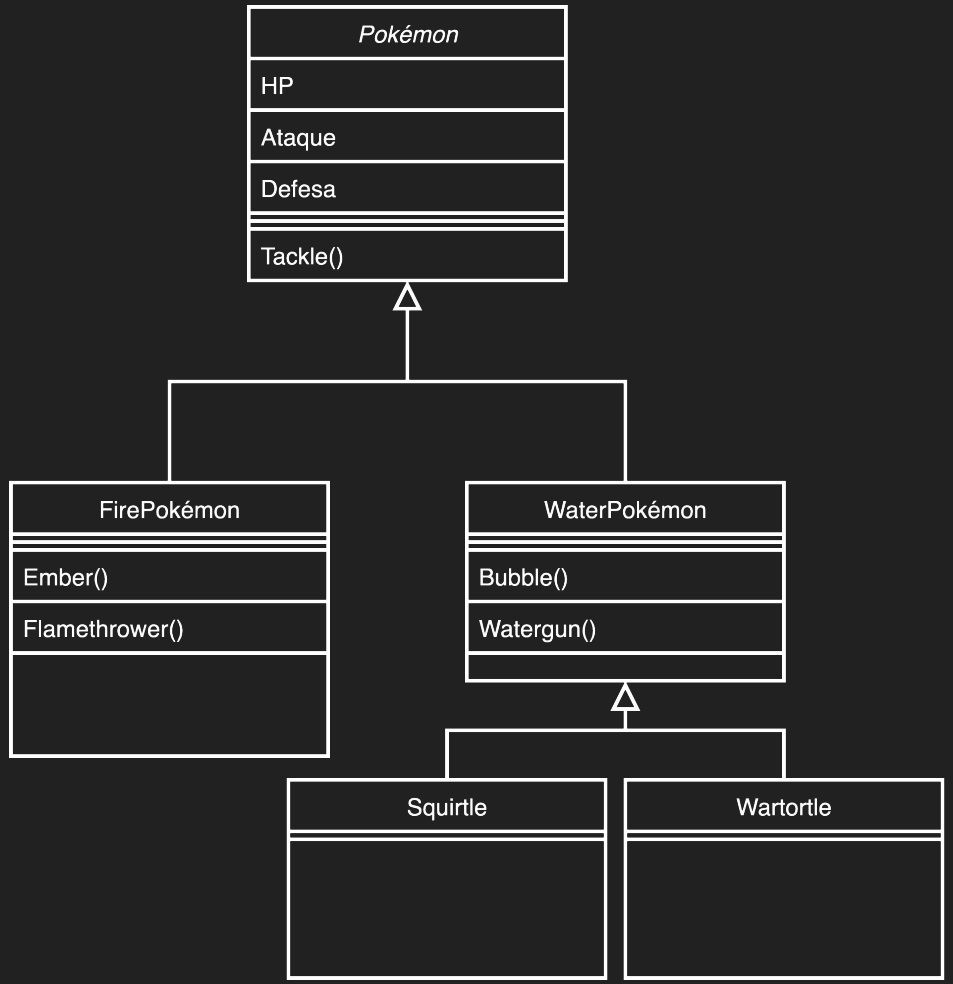
```
class Squirtle : public WaterPokemon{
    Métodos e Atributos}
```

- **C#**

```
class Squirtle : WaterPokemon{
    Métodos e Atributos}
```

- **Python**

```
class Squirtle (WaterPokemon):
    Métodos e Atributos
```


Pokémon
HP
Ataque
Defesa
Tackle()
FirePokémon
Ember()
Flamethrower()
WaterPokémon
Bubble()
Watergun()
Squirtle
Wartortle

E se eu precisar herdar de mais de uma classe?

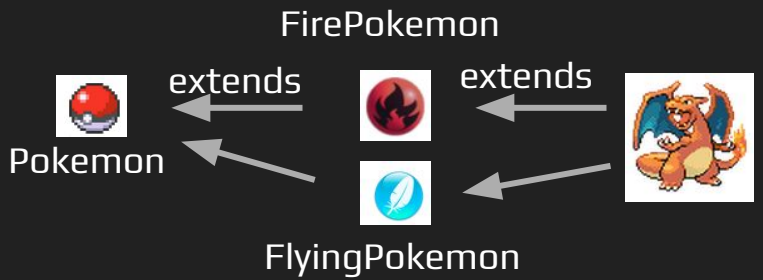
Herança

Charizard é do tipo **Voador** e **Fogo**.

Python e C++ possuem suporte para

Herança Múltipla

Portanto, Charizard pode herdar das duas classes!



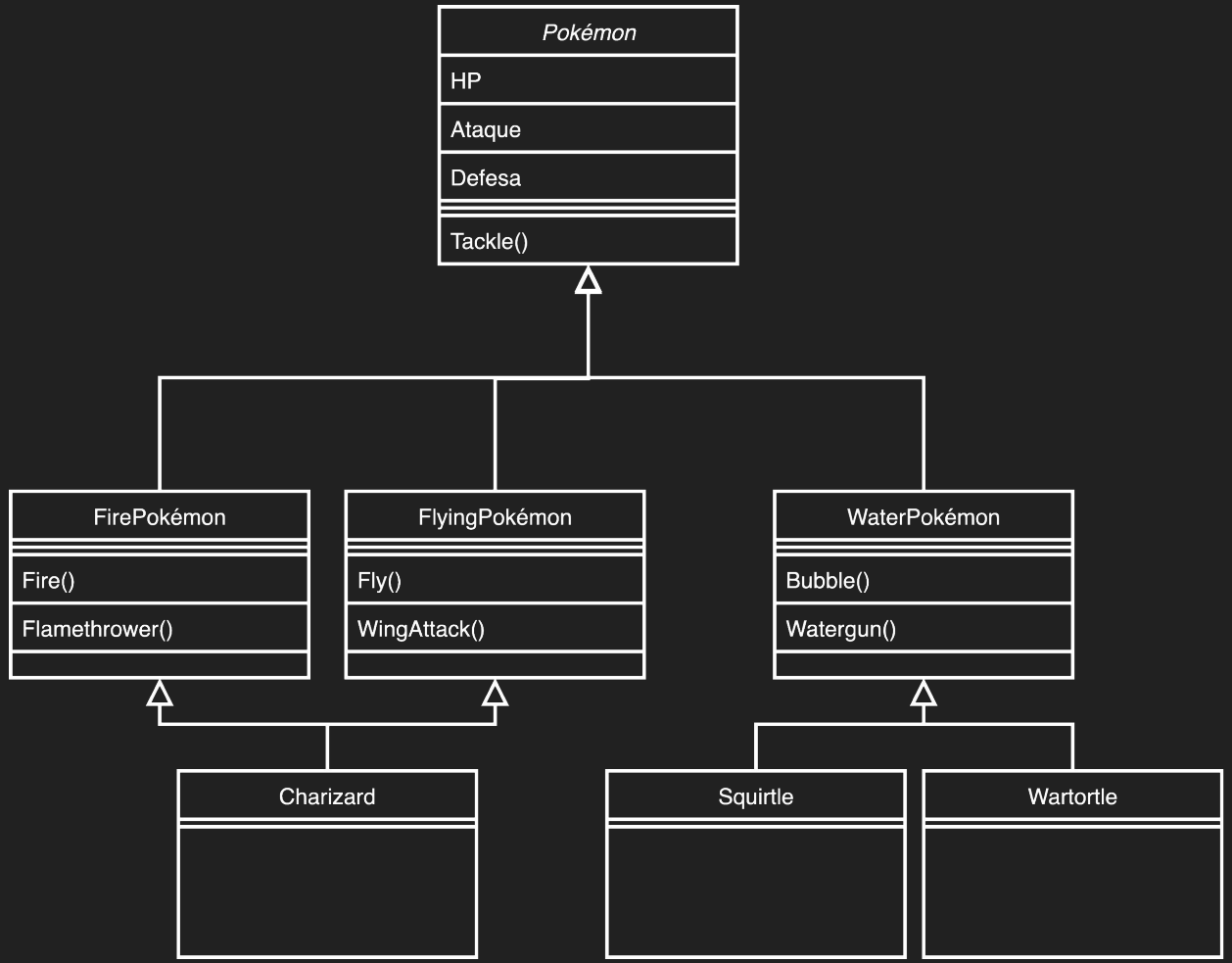
Herança

➔ **C++**

```
class Charizard: public FirePokemon, public FlyingPokemon{
    Métodos e Atributos}
```

➔ **Python**

```
class Charizard(FirePokemon, FlyingPokemon):
    Métodos e Atributos
```


Pokémon
HP
Ataque
Defesa
Tackle()
FirePokémon
Fire()
Flamethrower()
FlyingPokémon
Fly()
WingAttack()
WaterPokémon
Bubble()
Watergun()
Charizard
Squirtle
Wartortle

Herança Múltipla

- A herança múltipla pode aumentar significativamente a complexidade do código. Por isso, é sempre bom tomar cuidado ao usá-la nas linguagens que a possuem.
- Caso ambas as classes base tenham um atributo ou método com o mesmo identificador, ocorrerão erros.
- Isso dificulta o reuso de código, uma vez que agora o programador precisa saber os identificadores de uma classe base antes de herdar de outra. E, caso haja conflitos, re-escrever o código.

E nas outras linguagens? O que eu posso fazer?

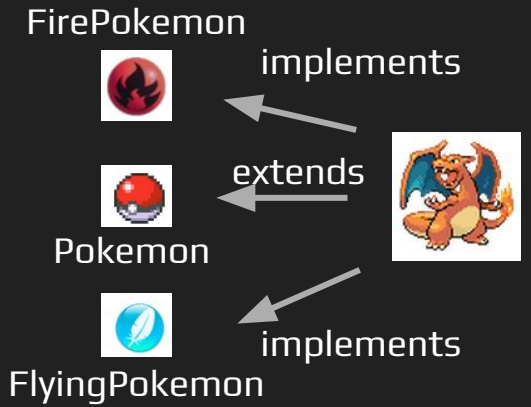
Interface!

Charizard é do tipo **Voador** e **Fogo**.

Java e C# possuem suporte para

Interface!

Charizard pode **implementar** quantas interfaces quiser!



Interface

➔ **Java**

```
private class Charizard: extends Pokemon implements
FirePokemon, FlyingPokemon{
    Métodos e Atributos
    PRECISA IMPLEMENTAR MÉTODOS DAS INTERFACES!*
}
```

Java 8 possui métodos *default* em interfaces

➔ **C#**

```
private class Charizard: Pokemon, FirePokemon,
FlyingPokemon{
    Métodos e Atributos
    PRECISA IMPLEMENTAR MÉTODOS DAS INTERFACES!
}
```

Classe PRECISA vir antes!

Interface

➔ **Java**

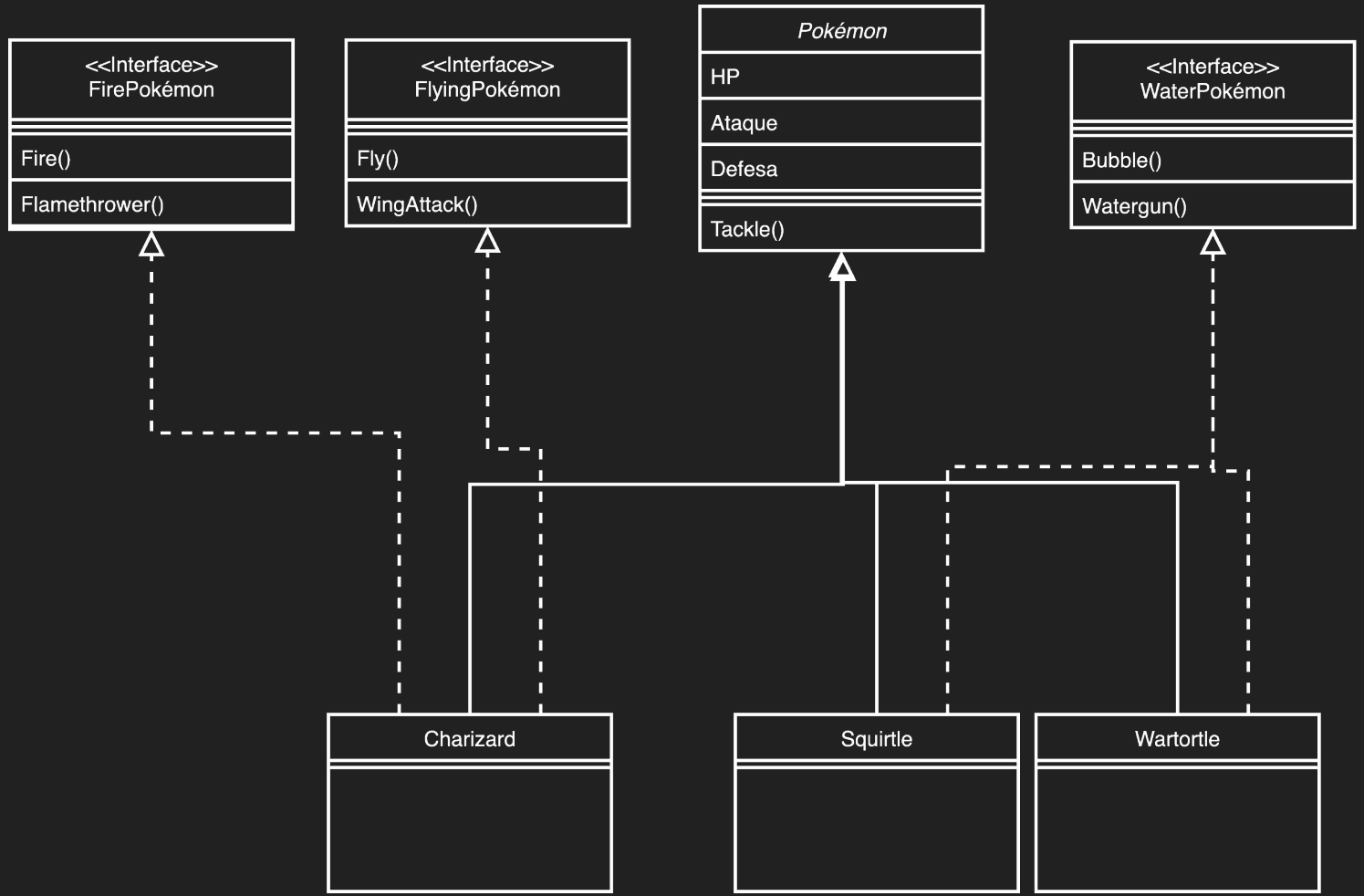
```
interface FirePokemon{
    public int Flamethrower(Pokémon target);
    public int Ember(Pokémon target);
}

interface FlyingPokemon{
    public int Fly(Pokémon target);
    public int WingAttack(Pokémon target);
}
```

Interface

- **Java**

```
private class Charizard: extends Pokemon implements FirePokemon,
FlyingPokemon{
    Métodos e Atributos
    PRECISA IMPLEMENTAR MÉTODOS DAS INTERFACES!
    public int Flamethrower(Pokémon target){
        private int power = 90;
        private int acc = 100;
        private int burnChance = 10;
        if(AccuracyCheck(acc)){
            target.ApplyDamage(power, type)
            if(BurnCheck(10))
                target.ApplyStatus(“Burn”);
        }
    }
}
```


<<Interface>>
FirePokémon
Fire()
Flamethrower()
<<Interface>>
FlyingPokémon
Fly()
WingAttack()
Pokémon
HP
Ataque
Defesa
Tackle()
<<Interface>>
WaterPokémon
Bubble()
Watergun()
Charizard
Squirtle
Wartortle

E se 2 interfaces tiverem o mesmo método?

Interface

➔ **Java**

```
interface FirePokemon{
    public int Flamethrower(Pokémon target);
    public int Ember(Pokémon target);
    public int FlameWings(Pokémon target);
}

interface FlyingPokemon{
    public int Fly(Pokémon target);
    public int WingAttack(Pokémon target);
    public int FlameWings(Pokémon target);
}
```

Interface

➔ **Java**

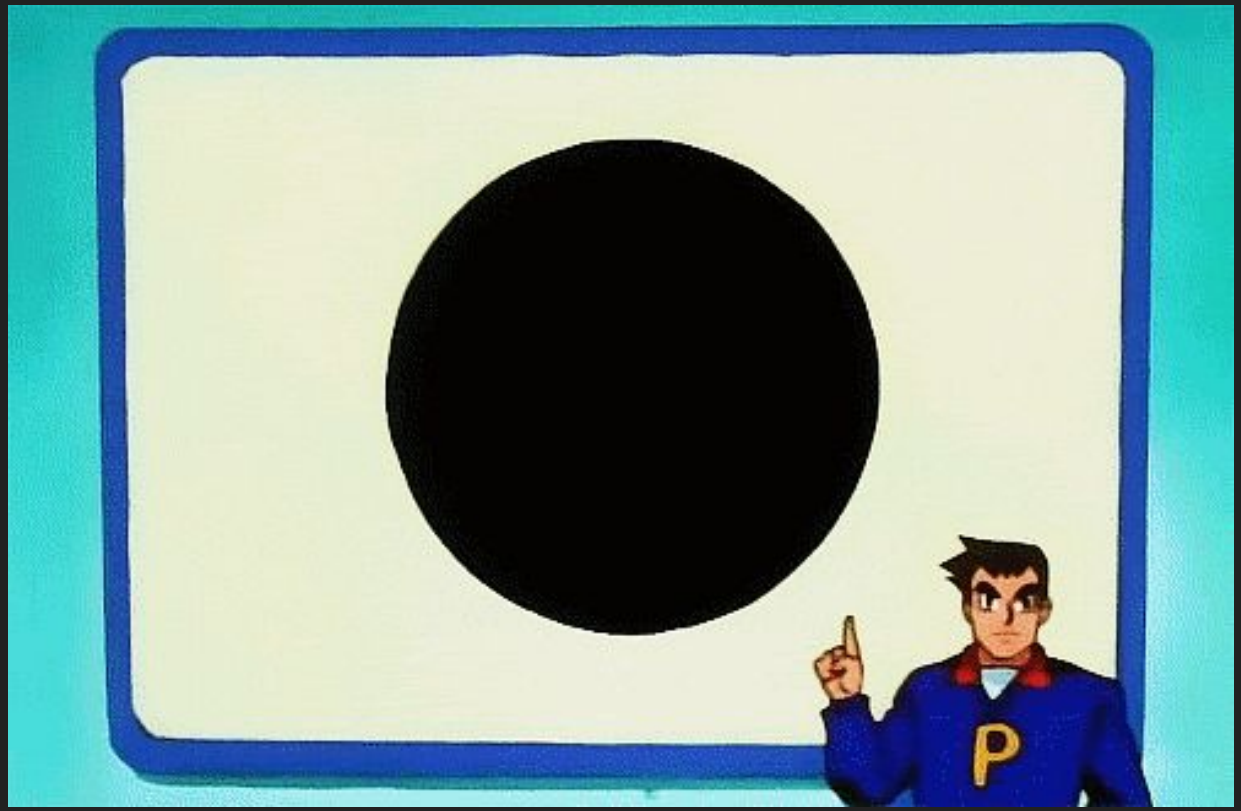
```
private class Charizard: extends Pokemon implements FirePokemon,
FlyingPokemon{
    Métodos e Atributos
    PRECISA IMPLEMENTAR MÉTODOS DAS INTERFACES!
    public int FirePokemon.super.FlameWings(Pokémon target){
        private int power = 120;
        private int acc = 100;
        if(AccuracyCheck(acc)){
            target.ApplyDamage(power, type)
        }
    }
}
```

super

- A palavra-chave *super* permite que você acesse métodos da classe base. Normalmente é usado para o caso de herança múltipla de implementação (o caso que acabamos de ver) ou quando um método é sobrescrito na classe filha, mas deseja-se usar o método da classe base (veremos na próxima aula!)
- Também pode ser usado para acessar os construtores das classes base (também veremos na próxima aula!)

Classe Abstrata

Fonte:
https://www.facebook.com/Pokemon/photos/a.242079005914051/901131740008771/?type=3&comment_id=901152336673378



Fonte:
https://66.media.tumblr.com/tumblr_m0os8iUt8T1r1n5pqo1_500.gif

Classe Abstrata

- *"...há casos em que é útil definir as classes a partir das quais o programador nunca pretenderá instanciar qualquer objeto. Essas classes são chamadas classes abstratas."*
  - DEITEL, H. M.; DEITEL, P. J. C++ Como Programar: 5 ed. São Paulo: Bookman, 2006. 1208 p.
- Um Pokémon por si só, sem os métodos de seu tipo (mesmo que do tipo normal) e sua classe específica não faz sentido ser instanciado.

Classe Abstrata

- Normalmente as classes abstratas são a base da herança
  - **Classes básicas abstratas.**
- São incompletas e, portanto, não podem instanciar objetos.
- Fornece uma base apropriada para as outras herdarem.
- As classes “completas”, que podem instanciar objetos são chamadas de **classes concretas**.
- Uma classe abstrata pode implementar interfaces

Classe Abstrata

➔ **Java**

```
abstract class Pokémon{
    Métodos e Atributos
    abstract int Tackle();}
```

➔ **C++**

```
class Pokémon{
    Métodos e Atributos
    public:
        virtual int Tackle();}
```

Classe Abstrata

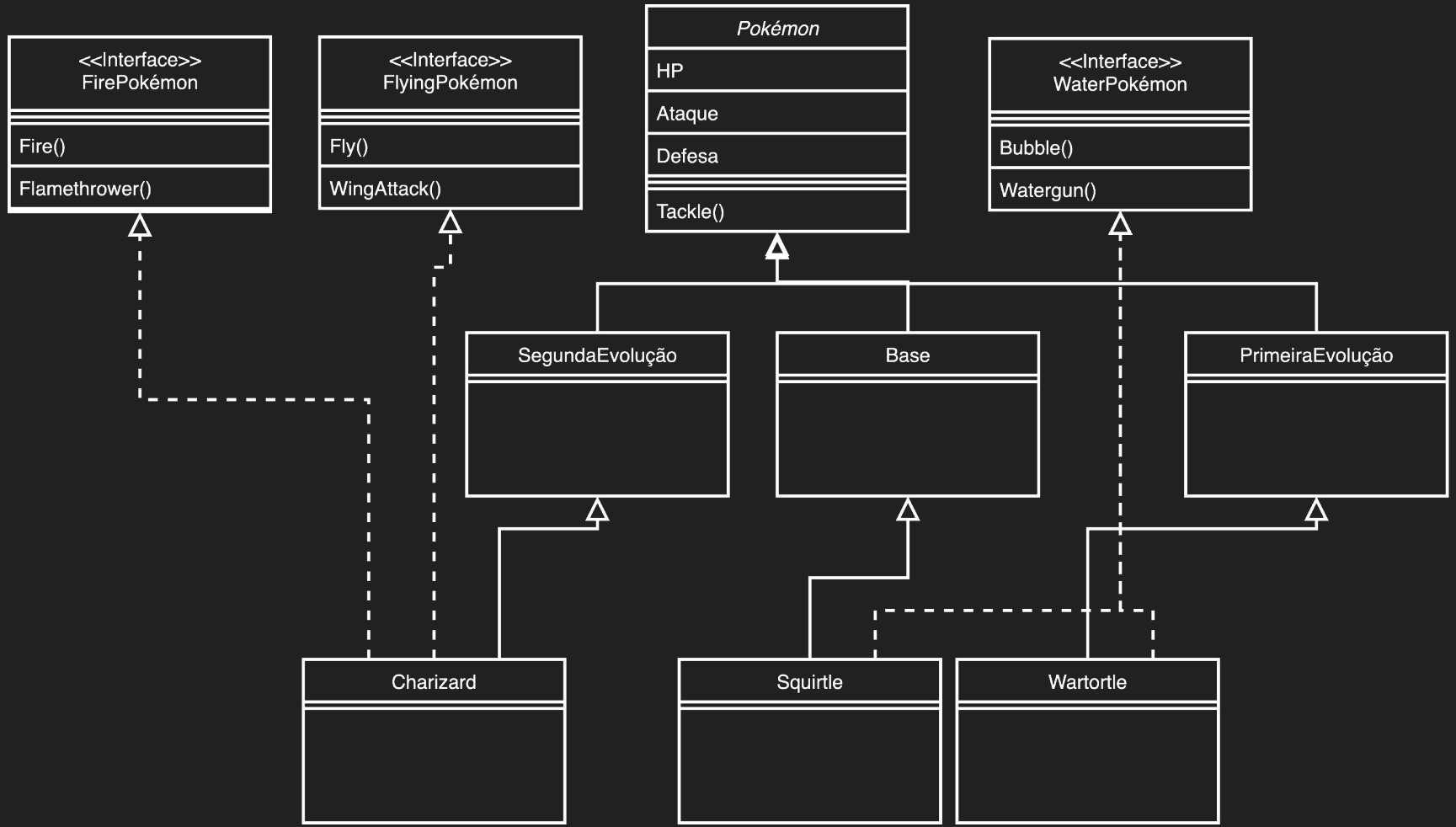
- **C#**

```
abstract class Pokemon{
    Métodos e Atributos
    public abstract int Tackle();}
```

- **Python**

```
from abc import ABC, abstractmethod
class Pokemon(ABC):
    Métodos e Atributos
    @abstractmethod
    def Tackle():
        pass
```

E se o estágio evolutivo do Pokémon tivesse informações muito importantes e que se repetissem para cada estágio?


<<Interface>>
FirePokémon
Fire()
Flamethrower()
<<Interface>>
FlyingPokémon
Fly()
WingAttack()
Pokémon
HP
Ataque
Defesa
Tackle()
<<Interface>>
WaterPokémon
Bubble()
Watergun()
SegundaEvolução
Base
PrimeiraEvolução
Charizard
Squirtle
Wartortle

Um exemplo mais próximo da realidade

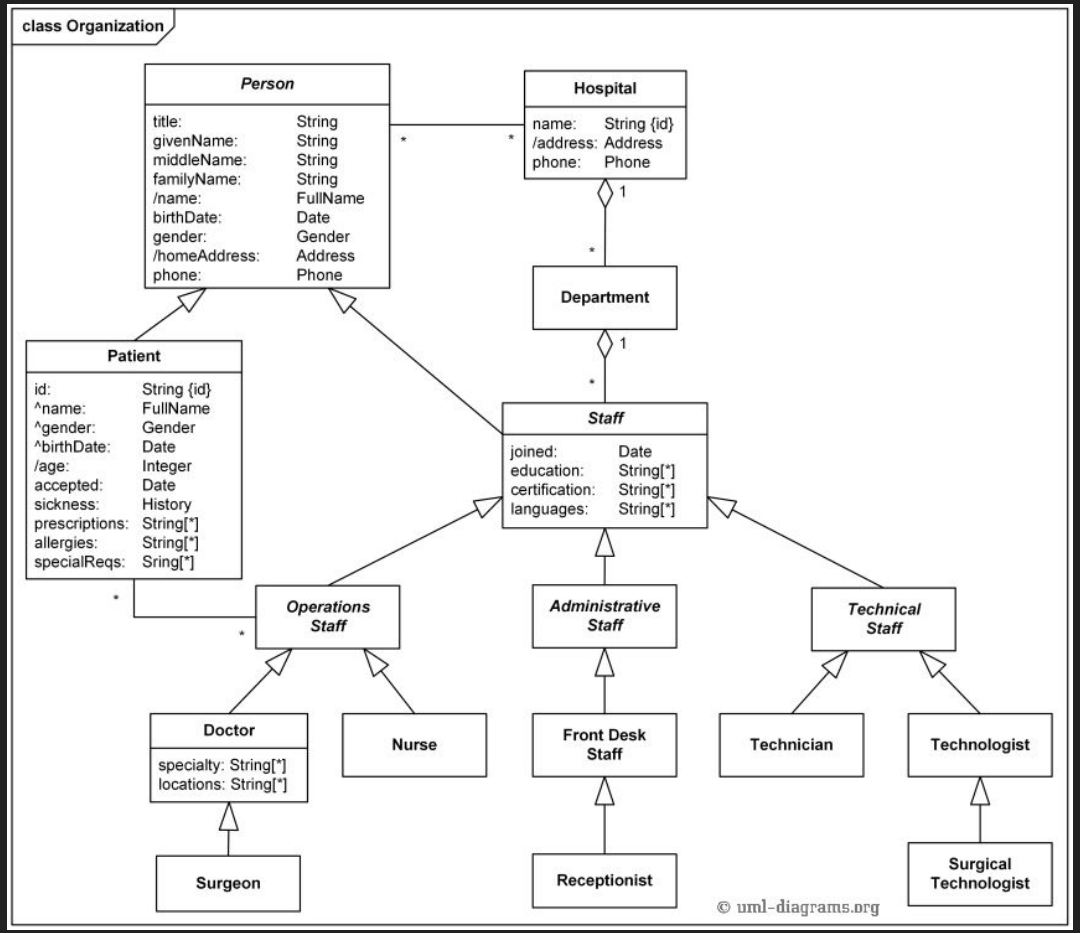


Diagrama de classes de um hospital

Fonte: https://www.uml-diagrams.org/examples/hospital-domain-diagram.html?context=cls-examples

Classes Internas

Classes Internas

- Podemos declarar classes como um membro de outra classe
- Existem 4 tipos de classes internas em java
  - Classe Aninhada Interna (Nested Inner Class)
  - Classe Internar de Método Local (Method Local Inner Class)
  - Classe Interna Anônima (Anonymous Inner Class)
  - Classe Aninhada Estática (Static Nested Class)

Classe Aninhada Interna

- Pode acessar qualquer variável de instância privada da classe externa
- Interfaces também podem ser aninhadas e ter acesso aos modificadores. Tem usos interessantes!
- Não é possível ter um método estático numa classe aninhada interna
  - Ela é implicitamente associada com um objeto da sua classe externa

Classe Aninhada Interna [1]

```
class Outer {
   private int outerVariable = 10;
   class Inner {
      public void show() {
           System.out.println("In a nested class method");
           System.out.println("Variable = "+outerVariable);
      }
   }
}

class Main {
   public static void main(String[] args) {
       Outer.Inner in = new Outer().new Inner();
       in.show();
   }
}
```

Classe Aninhada Interna [1]

```
class Outer {
    void outerMethod() {
        System.out.println("inside outerMethod");
    }
    class Inner {
        public static void main(String[] args){
            System.out.println("inside inner class Method");
        }
    }
}
```

ISSO VAI DAR ERRO POR CAUSA DO STATIC!!!

Classe Interna de Método Local

- ➔ Classes internas podem ser declaradas dentro de um método da classe externa
- ➔ Não podem usar uma variável local do método externo a não ser que essa variável não seja declarada como *final*

Classe Interna de Método Local [1]

```
class Outer {
    void outerMethod() {
        System.out.println("inside outerMethod");
        // Inner class is local to outerMethod()
        class Inner {
            void innerMethod() {
                System.out.println("inside innerMethod");
            }
        }
        Inner y = new Inner();
        y.innerMethod();
    }
}
class MethodDemo {
    public static void main(String[] args) {
        Outer x = new Outer();
        x.outerMethod();
    }
}
```

Classe Interna de Método Local [1]

```
class Outer {
    void outerMethod() {
        int x = 98;
        System.out.println("inside outerMethod");
        class Inner {
            void innerMethod() {
                System.out.println("x= "+x);
            }
        }
        Inner y = new Inner();
        y.innerMethod();
    }
}
class MethodLocalVariableDemo {
    public static void main(String[] args) {
        Outer x=new Outer();
        x.outerMethod();
    }
}
```

VAI DAR ERRO POIS VARIÁVEL NÃO É FINAL!

Classe Interna de Método Local [1]

```
class Outer {
void outerMethod() {
    final int x=98;
    System.out.println("inside outerMethod");
    class Inner {
        void innerMethod() {
            System.out.println("x = "+x);
        }
    }
    Inner y = new Inner();
    y.innerMethod();
}
}
class MethodLocalVariableDemo {
    public static void main(String[] args){
    Outer x = new Outer();
    x.outerMethod();
    }
}
```

Classe Interna de Método Local

- A variável local fica na *stack* enquanto o método está na *stack*
- Mas o objeto da classe interna pode estar na *heap* em alguns casos
- Por isso a variável precisa ser *final*
- As classes internas de métodos locais não podem ser marcadas como private, protected, static ou *transient*
- Mas podem ser marcadas como *abstract* OU *final*

Classe Aninhada Estática

- ➔ Não são tecnicamente uma classe interna
- ➔ São um membro estático da classe externa

Classe Aninhada Estática [1]

```
class Outer {
    private static void outerMethod() {
        System.out.println("inside outerMethod");
    }
    // A static inner class
    static class Inner {
        public static void main(String[] args) {
            System.out.println("inside inner class Method");
            outerMethod();
        }
    }
}
```

Classe Interna Anônima

- São declaradas sem nome!
- Podem ser criadas de dois jeitos
  - Como uma subclasse do tipo específico
  - Como implementadora de uma interface especificada

Como uma subclasse do tipo específico [1]

```
class Demo {
    void show() {
        System.out.println("i am in show method of super class");
    }
}
class Flavor1Demo {

    // An anonymous class with Demo as base class
    static Demo d = new Demo() {
        void show() {
            super.show();
            System.out.println("i am in Flavor1Demo class");
        }
    };
    public static void main(String[] args){
        d.show();
    }
}
```

Como implementadora de uma interface especificada [1]

```
class Flavor2Demo {
    // An anonymous class that implements Hello interface
    static Hello h = new Hello() {
        public void show() {
            System.out.println("i am in anonymous class");
        }
    };
    public static void main(String[] args) {
        h.show();
    }
}

interface Hello {
    void show();
}
```

Referências

1. https://www.geeksforgeeks.org/inner-class-java/
2. DEITEL, H. M.; DEITEL, P. J. C++ Como Programar: 5 ed. São Paulo: Bookman, 2006. 1208 p.
3. https://www.python-course.eu/python3_course.php
4. https://docs.microsoft.com/pt-br/dotnet/csharp/tutorials/
5. https://docs.oracle.com/javase/tutorial/java/index.html
6. https://pt.wikibooks.org/wiki/Programar_em_C%2B%2B
7. https://www.devmedia.com.br/abstracao-encapsulamento-e-heranca-pilares-da-poo-em-java/26366
8. https://www.devmedia.com.br/os-4-pilares-da-programacao-orientada-a-objetos/9264

Referências

- https://www.inf.pucrs.br/~gustavo/disciplinas/pli/material/paradigmas-aula13.pdf
- http://www.ic.unicamp.br/~cmrubira/aacesta/java/javatut10.html
- https://www.uml-diagrams.org/index-examples.html
- https://www.w3schools.com/java/java_inner_classes.asp
- https://staff.fnwi.uva.nl/a.j.p.heck/Courses/JAVAcourse/ch3/s1.html
- https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html

Obrigado!