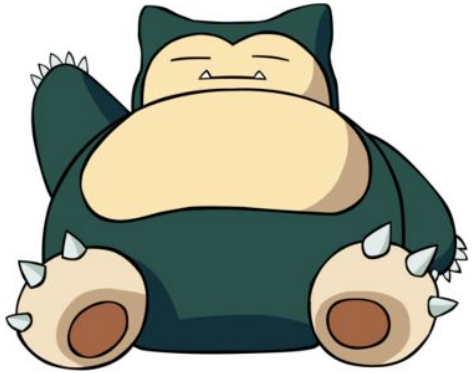
Paradigma de **O**rientação a **O**bjetos

com exemplos de POkémOn

Slides por:
Leonardo Tórtoro Pereira
Marina Coimbra
Mateus Malvessi



Definição

Objetos

Troca de Mensagens

Classes

Herança

Linguagens

Java

Ruby

Python

Smalltalk

C++/C#

PHP

Javascript

Conceitos

1. Abstração
2. Objeto
3. Classe
4. Herança
5. Polimorfismo
6. Sobrecarga de Métodos / Operadores
7. Encapsulamento
8. Tipagem dinâmica

1. Abstração

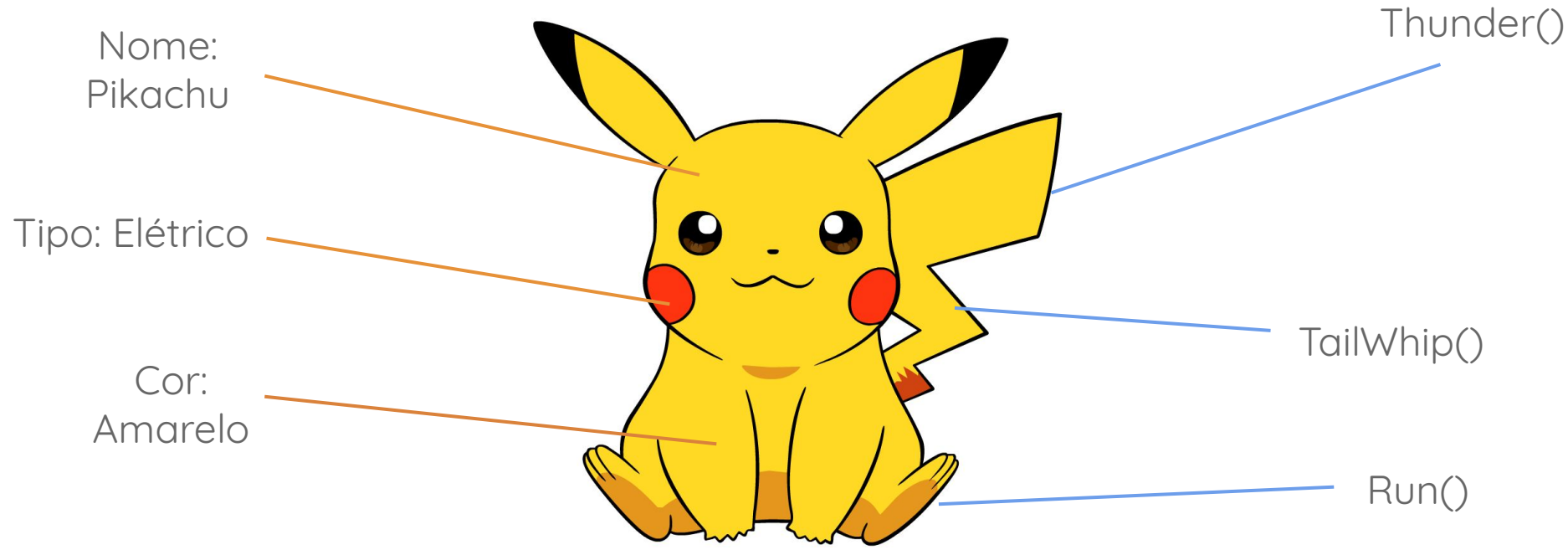
Em POO, os códigos são modelados baseados em conceitos de **Classe** e **Objeto**.

Classes: características e atributos que definem um molde para a criação de objetos.
Objetos: instâncias da classe.
Ex: Existem vários Pikachus, cada um pode possuir habilidades diferentes, mas possuem o mesmo conjunto de habilidades que podem aprender.

Abstração é o conceito de que cada **objeto** é responsável por realizar suas **próprias operações** sem que aquele que lhe passou a mensagem precise se preocupar com como ele faz isso.
Ex: um treinador pede a um Pokemon para utilizar uma habilidade, não precisa saber como o Pokemon a usará.

2. Objeto

Objeto = Características + Comportamento



2. 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, paralizado.

2. 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!

3. 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.

3. 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;
    }
  }
}
```

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

4. Herança

As classes também podem apresentar **semelhanças** entre si

Por exemplo:

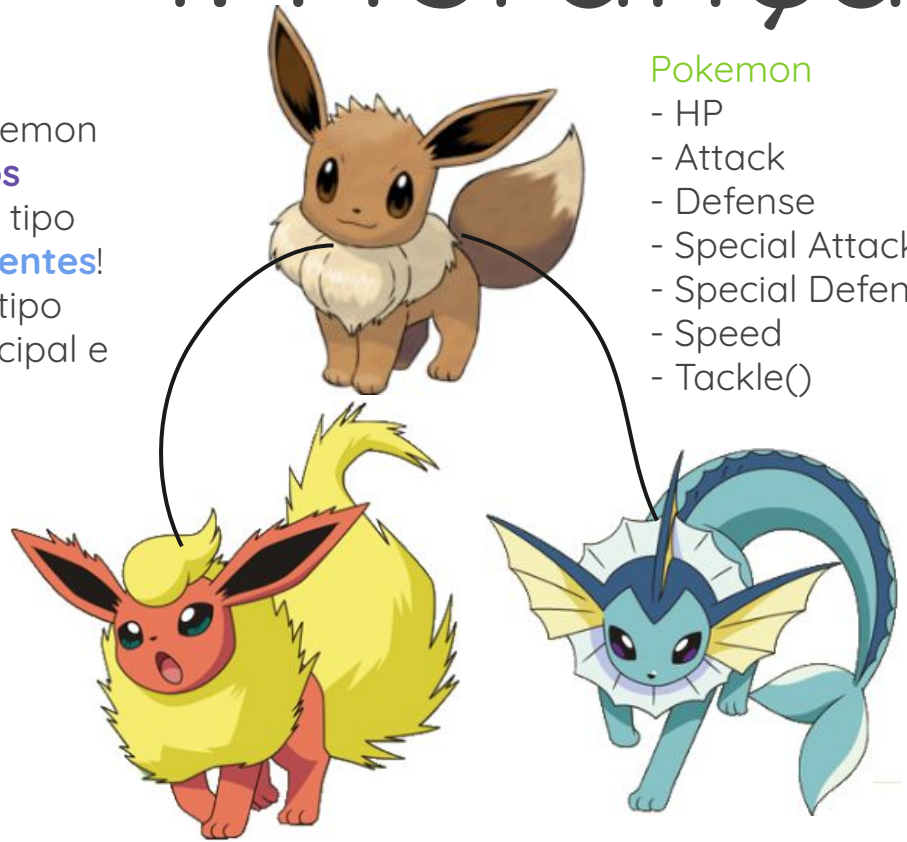
Flareon : FirePokemon
- HP
- Attack
- Defense
- Special Attack
- Special Defense
- Speed

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

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

4. Herança

Nesse caso, cada pokemon 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()

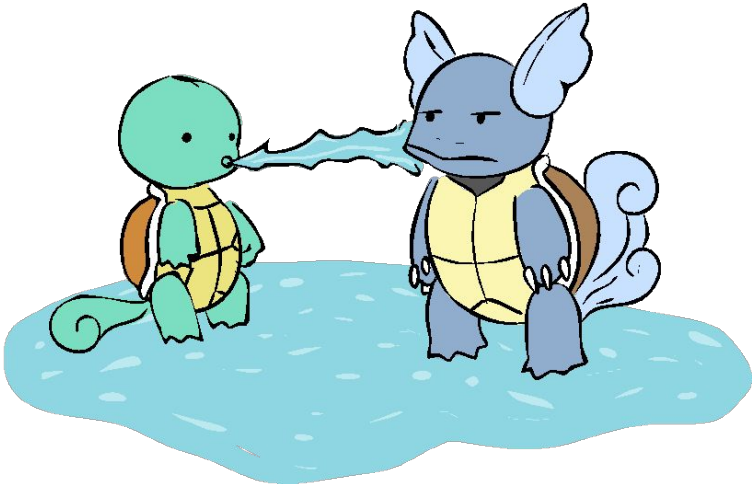
Quando ocorre a herança, as classes filhas como FirePokemon e WaterPokemon são **capazes de utilizar os atributos e métodos da classe mãe** Pokemon

FirePokemon
- Fire()
- Flamethrower()

WaterPokemon
- Bubble()
- WaterGun()

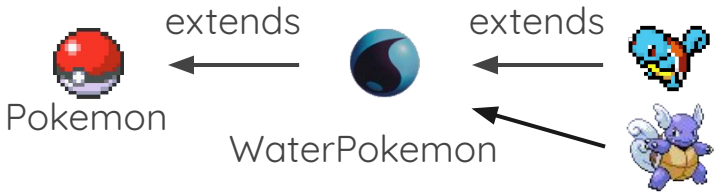
4. Herança

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

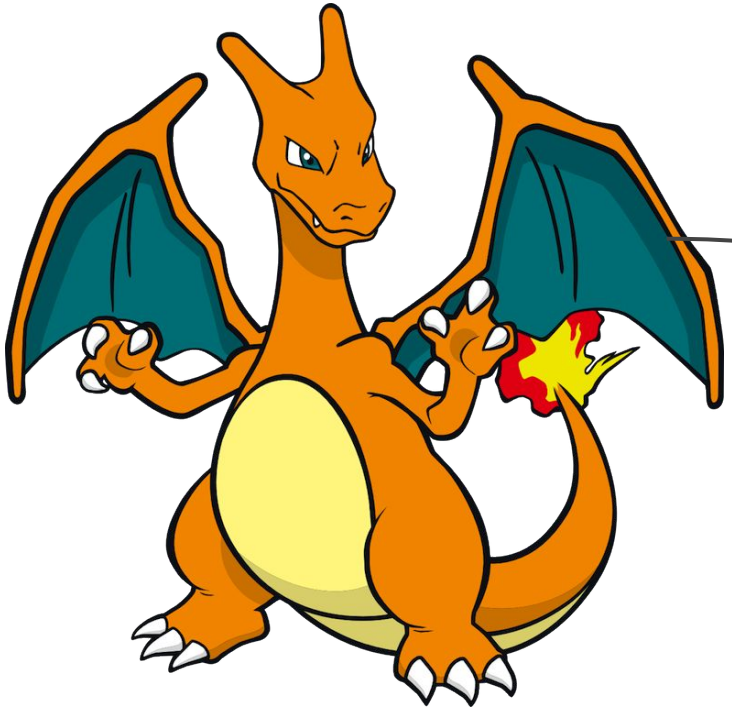


Cada pokemon possui **caracteristicas únicas** que diferem de seu tipo como uma **imagem** diferente e **quais habilidades** podem aprender e em qual nível.

a palavra-chave **extends** é utilizada para definir herança.



4. Herança

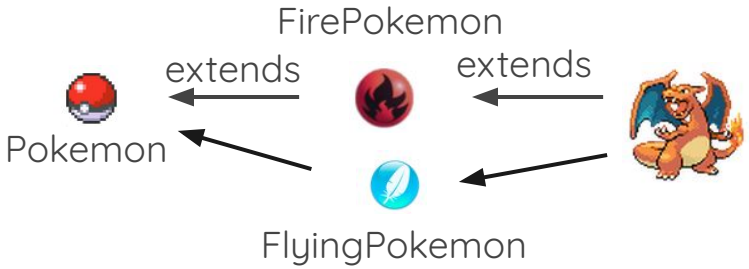


Python e C++ possuem suporte para **Herança**

Múltipla

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

Portanto, Charizard pode herdar das duas classes!



5. Polimorfismo

Polimorfismo é a capacidade de dois ou mais objetos responderem à **mesma mensagem** mas de **maneiras diferentes**.

Pokemons possuem um método que calcula o **dano recebido** por eles.

O dano recebido por cada um depende das fraquezas e resistências inerentes ao seu tipo: planta recebe 2x dano de fogo, já fogo recebe ½ dano de planta.

Seria muito trabalhoso programar se cada um dos métodos de calcular dano tivesse um **nome diferente**. São 720+ Pokemons!

```
class Pokemon{
  takeDamage(){...}
}
```

```
class Squirtle : WaterPokemon{
  takeDamageSquirtle(){...}
}
```

5. Polimorfismo

Solução - polimorfismo

```
class Pokemon{
  takeDamage(enemyType){...}
}

class Ghost : Pokemon {
  takeDamage(enemyType){...}
}

class Gengar : Ghost {
  takeDamage(enemyType){...}
}
```

A implementação de cada método é diferente, mas eles são chamados da mesma maneira. Se, por exemplo Gengar não implementar takeDamage(), o método é herdado de Ghost.

6a. Sobrecarga de Métodos

Arcanine possui uma habilidade **Fire Fang**.

Fire Fang tem 10% de chance de causar "burning", o que implica um dano verdadeiro de 1/16 de seu HP total que deve ser aplicado a cada turno.

Fire Fang também pode causar "flinch".

6a. Sobrecarga de Métodos

```
class FirePokemon : Pokemon{
  int fireFang(target){
    burnChance = rand()%100;
    flinchChance = rand()%100;
    if( burnChance < 10 && flinchChance < 10)
        return target.takeDamage(damage(),
            ["burn", "flinch"], 1/16, type);
    if ( burnChance < 10)
        return target.takeDamage(damage(),
            "burn", 1/16, type);
    if ( flinchChance < 10 )
        return target.takeDamage(damage(),
            "flinch", type);
    return target.takeDamage(damage(), type);
  }
}
```

6a. Sobrecarga de Métodos

Tropius receberá o dano de **Fire Fang.**

```
class GrassPokemon : Pokemon{
  void takeDamage(damage, statList, mod, type ){...}
  void takeDamage(damage, statList, type ){...}
  void takeDamage(damage, type ){...}
}
```

A **sobrecarga de métodos** permite que uma mesma mensagem seja transmitida com **parâmetos diferentes** e sua **interpretação correta** seja identificada.

6b. Sobrecarga de Operadores

Sem sobrecarga: 1 + 1 = 2

Pokemon + Pokemon = ?

A Sobrecarga de Operadores permite definir uma **função específica** para acontecer quando utilizamos os **operadores** (+, -, *, /, etc) sobre certa classe.

Podemos definir, por exemplo, que + significa reprodução de dois pokemons, **Pokemon + Pokemon = Ovo Pokemon**.

7. Encapsulamento

O Encapsulamento adiciona **segurança** à aplicação pois **esconde as propriedades,** criando uma espécie de **caixa preta**.

Métodos e atributos podem ser definidos como públicos (**public**), privados (**private**) ou protegidos (**protected**). Essa propriedade acrescenta **segurança e maior controle**, pois impede que objetos externos alterem valores de atributos do objeto.

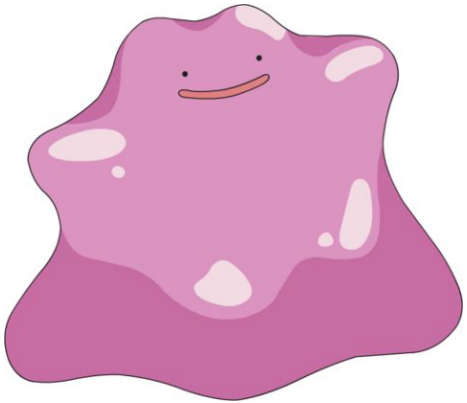
```
class Pokemon {
    protected hp;    //só pode ser acessado por subclasses e objeto
    private type;    //só pode ser acessado dentro do proprio objeto
    public getHP(); //pode ser chamada em qualquer classe
    protected setHP(); //só pode ser acessado por subclasses e objeto
    private damage(); //só pode ser acessada dentro do proprio objeto

}
```

8. Tipagem Dinâmica

Tipagem Dinâmica: o **tipo** de uma variável **não precisa ser declarado** e é **verificado em tempo de execução** em vez de ser verificado durante compilação.

Exemplos: Groovy, Javascript, Lisp, Objective-C, PHP, Prolog, Python, Ruby e Smalltalk



Pros: Flexibilidade e agilidade no desenvolvimento.

Exemplo:

C:
```
int a = 2 + "abc";
// erro detectado em fase de compilação
```
Python:
```
a = 2 + "abc";
//erro detectado somente durante a execução
```

Cons: podem acontecer erros difíceis de serem detectados em tempo de execução. Pode ser mais lenta para executar devido às checagens.

Solução: utilizar unit-testing durante o desenvolvimento para descobrir possíveis erros.

Conclusões

why is it Super effective?

1. Modularidade

"manutenção independente de cada objeto"

2. Ocultamento de Informações

"só há interação entre objetos pelos métodos"

3. Reuso de Código

"a não duplicação de código facilita a manutenção"

4. Conectividade e Facilidade de Depuração

"é facil remover somente o objeto com problemas"

Conclusões

why is it Super effective?

5. Código Enxuto

"os programas costumam ter menos linhas de código, em geral"

6. Rapidez no Desenvolvimento

"estrutura do código acelera o desenvolvimento"

7. Legibilidade do Código

"a modelagem em classes e objetos facilita o entendimento"

8. Segurança

"encapsulamento deixa o código mais seguro"

Referências

- Programação Orientada a Objetcos: uma introdução, Hardware.com.br. Encontrado em: <http://www.hardware.com.br/artigos/programacao-orientada-objetos/> (visitado por último em 20/05/15).
- Lesson: Object-Oriented Programming Concepts, Oracle Java Documentation. Encontrado em: <http://docs.oracle.com/javase/tutorial/java/concepts/> (visitado por último em 20/05/15).
- Os 4 Pilares da Porgramação Orientada a Objetos, Devmedia. Encontrado em: <http://www.devmedia.com.br/os-4-pilares-da-programacao-orientada-a-objetos/9264> (visitado por último em 20/05/15).
- Abstração, Encapsulamento e Herança: Pilares da POO em Java, Devmedia. Encontrado em: <http://www.devmedia.com.br/abstracao-encapsulamento-e-heranca-pilares-da-poo-em-java/26366> (visitado por último em 20/05/15).
- Kamienski, C. A. (1996). Introdução ao Paradigma de Orientação a Objetos. Centro Federal de Educação Tecnológica da Paraíba. Encontrado em: <http://www.cin.ufpe.br/~rcmg/cefet-al/proo/apostila-poo.pdf> (visitado por último em 20/05/15).
- Programar em C++/Herança, Wikilivros. Encontrado em: <http://pt.wikibooks.org/wiki/Programar_em_C%2B%2B/Heran%C3%A7a#Heran.C3.A7as_m.C3.BAltiplas> (visitado por último em 20/05/15).
- Pierce, Benjamin (2002). Types and Programming Languages. MIT Press. ISBN 0-262-16209-1.
- Holger Hans Peter Freyther GitHub, gnu-smalltalk/smalltalk/examples, encontrado em <https://github.com/gnu-smalltalk/smalltalk/tree/master/examples> (visitado por último em 20/05/15).

Obrigado!