



# Programação Orientada a Objetos

Herança, polimorfismo e sobrescrita

Ely – [ely.miranda@ifpi.edu.br](mailto:ely.miranda@ifpi.edu.br)

1

## Herança na vida real

- Herança (direito);
- Transmissão de bens, direitos e obrigações de uma pessoa falecida a seus sucessores legais.

[ely.miranda@ifpi.edu.br](mailto:ely.miranda@ifpi.edu.br)

2

2

## Herança na vida real

- Herança genética;
- Célula ou organismo adquirir características semelhantes a de um que o gerou.

*ely.miranda@ifpi.edu.br*

3

3

## Herança em P.O.O.

- Compartilhar atributos e métodos com o objetivo de reaproveitar código e comportamento;
- Por definição, existe uma superclasse e suas subclasses;
- Uma subclasse herda de uma superclasse.

*ely.miranda@ifpi.edu.br*

4

4

## Herança em P.O.O.

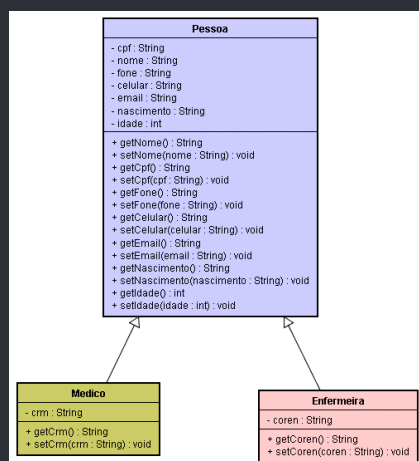
- Uma classe que herda de outra possui os atributos e métodos da superclasse;
- Com isso, não é necessário criar os mesmos atributos e métodos para classes semelhantes.
- Consegue-se uma relativa “economia” de código.

ely.miranda@ifpi.edu.br

5

5

## Exemplos de herança

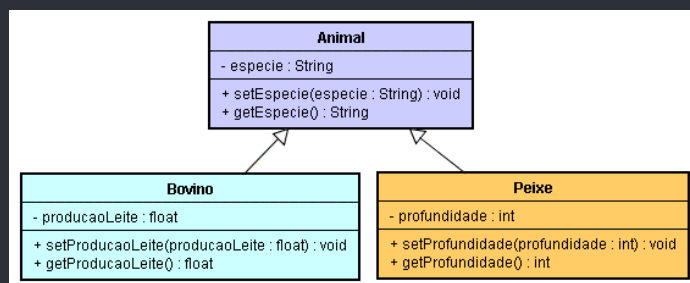


ely.miranda@ifpi.edu.br

6

6

## Exemplos de herança

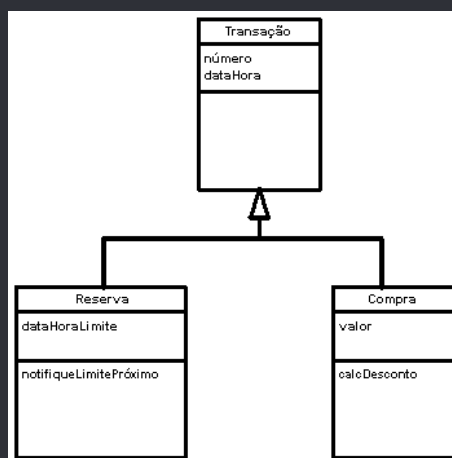


ely.miranda@ifpi.edu.br

7

7

## Exemplos de herança

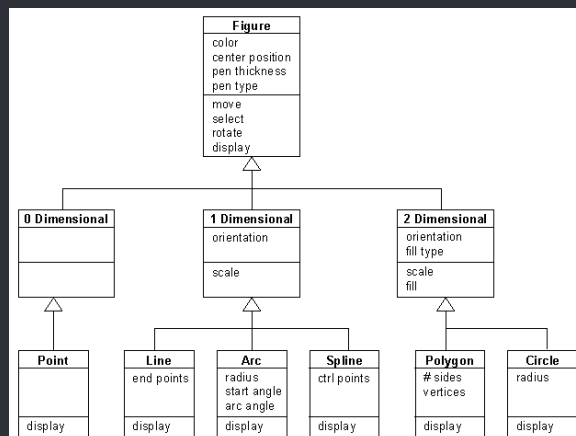


ely.miranda@ifpi.edu.br

8

8

## Exemplos de herança

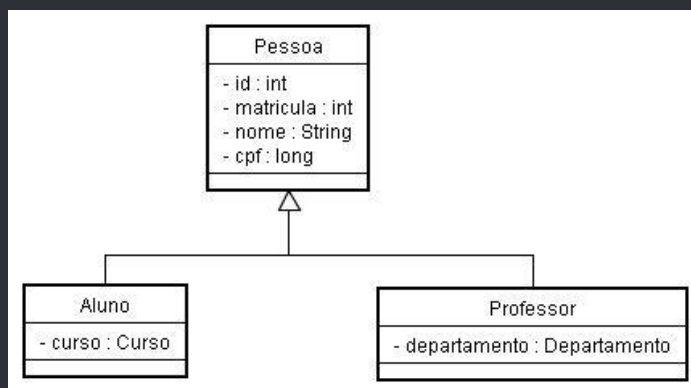


ely.miranda@ifpi.edu.br

9

9

## Exemplos de herança

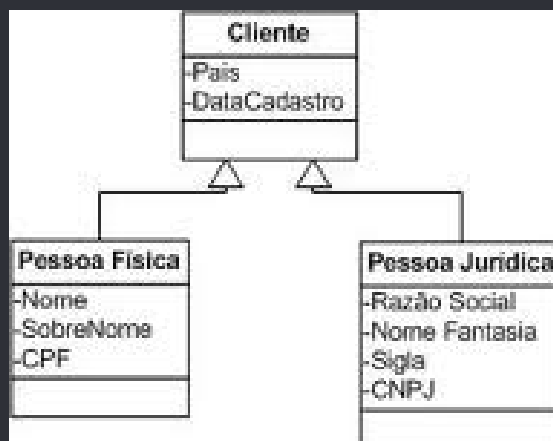


ely.miranda@ifpi.edu.br

10

10

## Exemplos de herança

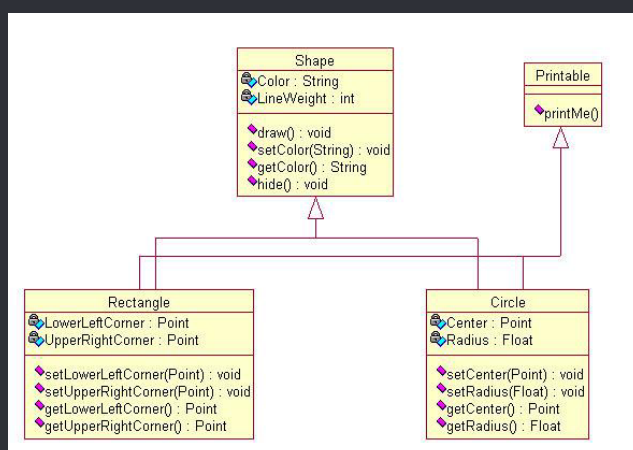


ely.miranda@ifpi.edu.br

11

11

## Exemplos de herança



ely.miranda@ifpi.edu.br

12

12

## Funcionários e Gerentes

- Uma empresa possui funcionários:

```
class Funcionario {  
    private _nome: string;  
    private _cpf: string;  
    private _salario: number;  
    //construtores e métodos get e set necessários  
}
```

*ely.miranda@ifpi.edu.br*

13

13

## Funcionários e Gerentes

- E além disso, funcionários que ocupam cargos de gerente;
- Gerentes possuem as mesmas características de funcionários e “algo mais”.

*ely.miranda@ifpi.edu.br*

14

14

## Funcionários e Gerentes

- Gerentes possuem as mesmas características de funcionários e:
  - Possuem login e senha;
  - Possuem um comportamento de autenticação em um sistema interno.

*ely.miranda@ifpi.edu.br*

15

15

## Funcionários e Gerentes

```
class Gerente {
    private _nome: string;
    private _cpf: string;
    private _salario: number;
    private _login: string;
    private _senha: string;

    public autentica(login: string, senha: string): boolean {
        return (this._login == login &&
            this._senha == senha) ;
    }
    //construtores e métodos get e set necessários
}
```

*ely.miranda@ifpi.edu.br*

16

16



## Problemas

- Duplicação de código;
- Duas classes com atributos repetidos;
- Métodos get, set e construtores também se repetem.

*ely.miranda@ifpi.edu.br*

17

17

## Problemas

- Mudanças em um atributo ou método (nome, tipo ou validação) geram alterações em ambas as classes;
- Havendo outro funcionário com outras características e comportamentos, o código seria triplicado;
- ... e assim por diante.

*ely.miranda@ifpi.edu.br*

18

18

## Possível solução

- Deixar a classe funcionário mais genérica, com atributos e métodos da classe Gerente;
- Criar um campo lógico em funcionário indicando se o mesmo é um gerente;
- Caso não seja um gerente, deixar os campos em branco (opcionais) e jamais chamar o método autenticar.

*ely.miranda@ifpi.edu.br*

19

19

## Outro problema

- Havendo muitos atributos opcionais, o programador deveria lembrar quais são necessários;
- Ninguém garante que o método autenticar não seria chamado.

*ely.miranda@ifpi.edu.br*

20

20

## Outro problema

- Havendo outro funcionário com outras características e comportamentos:
  - A variável de tipo de funcionário não seria mais lógica e sim outro tipo;
  - O controle de atributos opcionais aumentaria;
  - O controle de métodos que não podem ser chamados aumentaria;
  - ... e assim por diante.

*ely.miranda@ifpi.edu.br*

21

21

## Conta x Poupança

- Em um banco, além da conta comum, temos uma conta poupança;
- Além de número e saldo, tem também uma taxa de juros;
- Pode-se gerar um rendimento devido ao saldo acumulado.

*ely.miranda@ifpi.edu.br*

22

22

## Conta x Poupança

- A classe Poupança, possui poucas diferenças para a classe Conta, apesar de ser um tipo distinto:

```
class Poupanca {
    private _numero: string;
    private _saldo: number;
    private _taxaJuros: number;

    public renderJuros(): void {
        this.depositar(this._saldo * this._taxaJuros/100);
    }
    //a partir daqui, tudo igual à classe conta
    public depositar(valor: number): void {
        this._saldo += valor;
    }
    //...
}
```

ely.miranda@ifpi.edu.br

23

23

## Conta x Poupança

```
class Poupanca {
    private _numero: string;
    private _saldo: number;
    private _taxaJuros: number;

    public renderJuros(): void {
        this.depositar(this._saldo * this._taxaJuros/100);
    }
    //a partir daqui, tudo igual à classe conta
    public depositar(valor: number): void {
        this._saldo += valor;
    }
    //...
}
```

Únicas diferenças

Mesmos problemas que há com as classes Funcionario e Gerente

ely.miranda@ifpi.edu.br

24

24

## Conta x Poupança: Piorando a situação

- Como ficaria a nossa aplicação do banco?
  - Uma possível solução é ter 2 arrays, um para cada tipo de conta;
  - Duplicar cada método (creditar, incluir, alterar...) para cada tipo de conta.

*ely.miranda@ifpi.edu.br*

25

25

## Conta x Poupança: Piorando a situação

```
class Banco {
  private _contas: Conta[] = [];
  private _poupancas: Poupanca[] = [];

  inserir(conta: Conta): void {
    let contaConsultada = this.consultar(conta.numero);
    if (contaConsultada == null) {
      this._contas.push(conta);
    }
  }
  inserirPoupanca(poupanca: Poupanca): void {
    let poupancaConsultada = this.consultarPoupanca(poupanca.numero);
    if (poupancaConsultada == null) {
      this._poupancas.push(poupanca);
    }
  }
  //...
}
```

*ely.miranda@ifpi.edu.br*

26

26

## Conta x Poupança: Piorando a situação

- Novamente: ... e caso apareçam mais N tipos de conta?  
Conta Imposto, Conta Salário, Conta Especial...???

*ely.miranda@ifpi.edu.br*

27

27

## Solução

- As classes citadas estão em um mesmo contexto:
  - Gerente é um funcionário;
  - Uma poupança é uma conta.

*ely.miranda@ifpi.edu.br*

28

28

## Solução

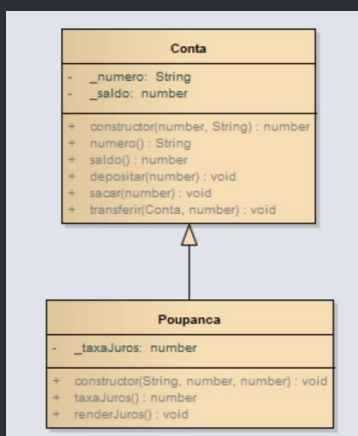
- Poupança e Gerente são simples extensões das definições de Conta e Funcionário;
- A partir dessa semelhança podemos usar “herança” e simplificar as implementações.

*ely.miranda@ifpi.edu.br*

29

29

## Conta e Poupança em UML



*ely.miranda@ifpi.edu.br*

30

30

## Herança com TypeScript

- Uma classe pode ser derivada de outra e herdar seu estado (atributos) e seu comportamento (métodos);
- Caso um objeto faz o mesmo que outro objeto e “mais alguma coisa”;
- Reutilizamos o código, definindo uma subclasse apenas com diferenças.

ely.miranda@ifpi.edu.br

31

31

## Herança com TypeScript

- A *extends* é utilizada para indicar a herança

```
class Gerente extends Funcionario { ... }
```

```
class Poupanca extends Conta { ... }
```

ely.miranda@ifpi.edu.br

32

32



## Herança

```
class Gerente extends Funcionario {
    private _login: string;
    private _senha: string;

    public autentica(login: string, senha: string): boolean {
        return (this._login == login && this._senha == senha) ;
    }

    //construtores e métodos get e set necessários
}
```

*Os demais atributos e métodos não precisam ser reescritos, pois são herdados*

ely.miranda@ifpi.edu.br

33

33

## Herança

```
class Poupanca extends Conta {
    private _taxaJuros: number;
    public renderJuros(): void {
        this.depositar(this.saldo * this._taxaJuros/100));
    }

    get taxaJuros(): number {
        return this._taxaJuros
    }
}
```

*Os demais atributos e métodos não precisam ser reescritos, pois são herdados*

ely.miranda@ifpi.edu.br

34

34

## Herança - construtor

```
class Poupanca extends Conta {  
    //...  
    constructor(numero: string, saldo: number,  
                                     taxaJuros: number ) {  
        super(numero, saldo);  
        this._taxaJuros = taxaJuros;  
    }  
}
```

*ely.miranda@ifpi.edu.br*

35

35

## Super

- A palavra reservada super dá acesso aos métodos e construtor da superclasse;
- Veremos mais adiante mais detalhes do seu uso...

*ely.miranda@ifpi.edu.br*

36

36

## Hierarquia de classes

- A mais acima na hierarquia é chamada de super classe ou classe mãe;
- As que herdam são chamadas de subclasses ou classes filhas.

*ely.miranda@ifpi.edu.br*

37

37

## Hierarquia de classes

- Uma subclasse não tem acesso direto aos membros privados de sua superclasse;
- Todo objeto de uma subclasse também é um objeto de sua superclasse.

*ely.miranda@ifpi.edu.br*

38

38

## Restrições

- Atributos e métodos privados são herdados, mas são acessíveis apenas por métodos get/set públicos;
- Modificador protected: visibilidade restrita a classe e subclasses;
- Construtor padrão só é disponível se também for disponível na superclasse.

*ely.miranda@ifpi.edu.br*

39

39

## Usando a classe Poupanca

- Apesar de não definidos, os métodos de crédito e de débito são herdados visíveis por serem públicos:

```
let p: Poupanca = new Poupanca("2", 100, 0.5);  
p.depositar(100);  
p.sacar(50);  
p.renderJuros();  
console.log(p.saldo); //150.75
```

*ely.miranda@ifpi.edu.br*

40

40

## Polimorfismo

- É a capacidade de um objeto poder ser referenciado de várias formas;
- Onde temos uma subclasse, também podemos usar uma superclasse.

*ely.miranda@ifpi.edu.br*

41

41

## Polimorfismo

```
let conta: Conta;  
conta = new Poupanca("2", 100, 0.5)  
conta.depositar(100);  
  
// ou no caso funcionário/gerente:  
let gerente: Gerente = new Gerente();  
let funcionario: Funcionario = gerente;
```

*ely.miranda@ifpi.edu.br*

42

42

## Polimorfismo

- Dizemos que uma Poupanca cabe em uma Conta, mas não o contrário:

```
let conta: Conta;
conta = new Poupanca("2", 100, 0.5);
conta.depositar(100);
console.log(p.saldo); //200
```

ely.miranda@ifpi.edu.br

43

43

## Casts

- Declarando como uma superclasse, devem-se usar casts para acessar elementos específicos da subclasse:

```
let conta: Conta;
conta = new Poupanca("2", 100, 0.05)
conta.depositar(100);
let poupanca : Poupanca = <Poupanca> conta;
poupanca.renderJuros();
// ou (<Poupanca> conta).renderJuros()
console.log(poupanca.saldo); //201
```

ely.miranda@ifpi.edu.br

44

44

## Verificação de tipos

- Usa-se o operador `instanceof` para saber se um objeto é de determinado tipo;
- Uso indicado para evitar casts que gerem erros.

*ely.miranda@ifpi.edu.br*

45

45

## Verificação de tipos

```
let conta: Conta = new Poupanca("2", 100, 0.5);  
if (conta instanceof Poupanca) {  
    (<Poupanca> conta).renderJuros();  
}  
console.log(conta.saldo); //100.50
```

*ely.miranda@ifpi.edu.br*

46

46

## Aplicação Banco

- Como Poupança é do mesmo tipo de Conta, a classe Banco pouco deve ser alterada:

```
banco.inserir(new Conta("1", 100));  
banco.inserir(new Poupanca("2", 100, 0.5));  
banco.depositar("1",200);  
banco.transferir("1","2",50);  
console.log(banco.consultar("1").saldo); //250  
console.log(banco.consultar("2").saldo); //150
```

*ely.miranda@ifpi.edu.br*

47

47

## Sobrescrita (override)

- É a redefinição de métodos de uma superclasse em uma subclasse;
- A reescrita de um método sobrepõe a implementação original;
- O método deve possuir o mesmo nome, tipo de retorno, visibilidade e lista de parâmetros.

*ely.miranda@ifpi.edu.br*

48

48



## Sobrescrita (override)

- Utilizada quando o comportamento do método da superclasse não corresponde ao desejado para o método da subclasse;
- Pode-se chamar ainda a implementação original se necessário.

*ely.miranda@ifpi.edu.br*

49

49

## Sobrescrita (override)

- Supondo uma ContaImposto:
  - Herda de conta e possui um atributo que representa um % descontado a cada débito;
  - Exemplo: A antiga CPMF descontava 0,38% do valor de cada operação de débito.

*ely.miranda@ifpi.edu.br*

50

50

## Sobrescrita (override)

- Supondo uma Contalmposto:
  - Exemplo: A antiga CPMF descontava 0,38% do valor de cada operação de débito
    - Saldo: R\$ 100,00;
    - Saque: R\$ 10,00;
    - CPMF: R\$ 0,038;
    - Saldo final: R\$ 89,962.

*ely.miranda@ifpi.edu.br*

51

51

## Sobrescrita (override)

- O débito da Contalmposto apenas subtrai um valor do saldo 2x;
- Precisa ser sobrescrito na classe Contalmposto para ter o comportamento desejado.

*ely.miranda@ifpi.edu.br*

52

52

## Sobrescrita (override)

```
class ContaImposto extends Conta {
    private _taxaDesconto: number;

    constructor(numero: string, saldo: number,
                taxaDeDesconto: number) {
        super(numero, saldo);
        this._taxaDesconto = taxaDeDesconto;
    }
    //...
}
```

ely.miranda@ifpi.edu.br

53

53

## Sobrescrita (override)

```
class ContaImposto extends Conta {
    private _taxaDesconto: number;
    //...
    debitar(valor: number): void {
        this._saldo -= valor;
        this._saldo -= valor * getTaxaDesconto() / 100;
    }
}
```

← Sobrescrita sem acesso  
à implementação anterior  
(não compila, pois o  
saldo é privado)

ely.miranda@ifpi.edu.br

54

54

## Sobrescrita (override)

- A ideia do código anterior está parcialmente correta:
  - O atributo saldo é privado, então não compilará;
  - O método debitar da classe pai não foi considerado e nele há regras de negócio implementadas.

ely.miranda@ifpi.edu.br

55

55

## Sobrescrita (override)

```
class ContaImposto extends Conta {
  private _taxaDesconto: number;
  //...
  debitar(valor: number): void {
    let total = valor + valor * (this._taxaDesconto/100)
    super.sacar(total);
  }
}
```

*Sobrescrita com acesso à implementação anterior* ←

ely.miranda@ifpi.edu.br

56

56

## Sobrecarga (overload)

- Escrever um método com o mesmo nome, mas:
  - Argumentos diferentes;
  - Tipo do retorno pode ser igual ou diferente;
- Usada quando se tem a necessidade de diferentes formas de se chamar um método.

ely.miranda@ifpi.edu.br


57

57

## Sobrecarga (overload)

```
class Calculadora {  
    soma(op1: number, op2: number): number {  
        return op1 + op2;  
    }  
  
    soma(op1: string, op2: string): string {  
        let op1Int: number = parseInt(op1);  
        let op2Int: number = parseInt(op2);  
        let resultado: number = op1Int + op2Int;  
        return resultado.toString();  
    }  
}
```

*Não compila*



ely.miranda@ifpi.edu.br

58

58

## Sobrecarga (overload)

- Atualmente, em Typescript é suportado, mas desaconselhado:
  - <https://www.stevefenton.co.uk/2013/02/what-is-wrong-with-method-overloads-in-typescript/>
  - <https://stackoverflow.com/questions/12688275/how-to-do-method-overloading-in-typescript/>

*ely.miranda@ifpi.edu.br*

59

59



# Programação Orientada a Objetos

Herança, polimorfismo e sobrescrita

Ely – [ely.miranda@ifpi.edu.br](mailto:ely.miranda@ifpi.edu.br)

60