INSTITUTO FEDERAL
ESPÍRITO SANTO

Campus Santa Teresa

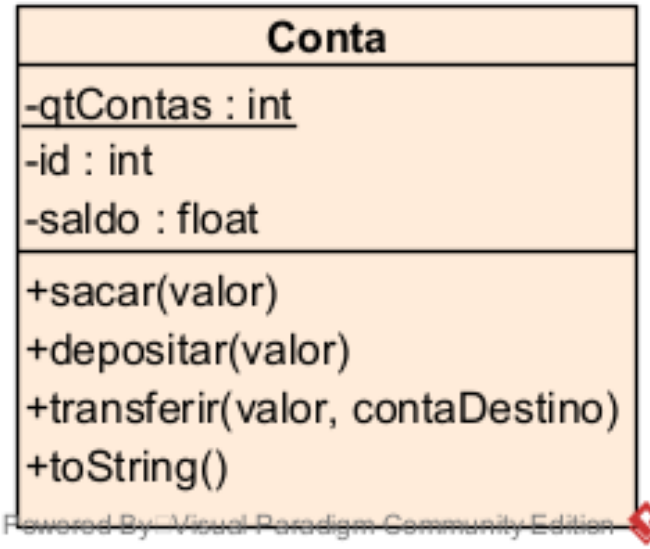
OO EM JAVASCRIPT: ATRITUTOS ESTÁTICOS (STATIC) ASSOCIAÇÃO ENTRE CLASSES HERANÇA

Profs. Archimedes Detoni

- Conteúdos
  - Atributos Estáticos (*static*)
  - Associação entre Classes
  - Herança: Generalização e Especialização
    - Sobrescrita de métodos

Atributos Estáticos

- Retomando o exemplo da classe Conta, imagine que precisamos controlar a quantidade de contas abertas no nosso sistema
- Dessa forma, cada vez que for criada uma nova conta, uma variável contadora deve ser incrementada
- Parece razoável que, para fazer isso, podemos criar um atributo ***qtContas*** que se iniciada com 0 e incrementada toda vez que instanciarmos um objeto da classe Conta



Atributos Estáticos

```
export class Conta {
    #qtContas = 0;
    #id;
    #saldo;
    constructor(saldo = 0.0){
        this.#saldo = saldo < 0.0 ? 0.0 : saldo;
        this.#qtContas++;
        this.#id = ""+ new Date().getFullYear() + this.#qtContas;
    }

    get qtContas(){
         return this.#qtContas;
    }
//... O restante do código não muda ...
```

Atributos Estáticos

- O problema dessa implementação é que cada objeto da classe ***Conta*** terá o atributo ***#qtContas*** com o valor 1, independente da quantidade de instâncias criadas da classe ***Conta***
  - ***#qtContas*** seria o atributo de um objeto

- Vamos executar as instruções a seguir para verificar:

```
const conta1 = new Conta(1000.00);
const conta2 = new Conta();

console.log ("Quantidade Contas do Banco = " + conta1.qtContas);
//Quantidade Contas do Banco = 1

console.log ("Quantidade Contas do Banco = " + conta2.qtContas);
//Quantidade Contas do Banco = 1
```

Atributos Estáticos

- Precisamos que o atributo ***#qtContas*** seja inerente à classe ***Conta*** como um todo, e não às instâncias individualmente
  - Não podemos declará-lo da mesma forma que os outros atributos
  - Precisamos que ele seja um **atributo estático**, do tipo que reflete as alterações em todos os objetos da classe. Para isso usaremos a palavra reservada ***static***.

Atributos Estáticos

```
export class Conta {
    static #qtContas = 0;  //atributo estático
    #id;
    #saldo;
    constructor(saldo = 0.0){
        this.#saldo = saldo < 0.0 ? 0.0 : saldo;

        Conta.#qtContas++; //ATENÇÃO: não usa o this, mas Conta

        this.#id = ""+ new Date().getFullYear() + Conta.#qtContas;
    }

    static get qtContas(){
         return Conta.#qtContas; //ATENÇÃO: não usa o this, mas Conta
    }


//... O restante do código não muda ...
```

Atributos Estáticos

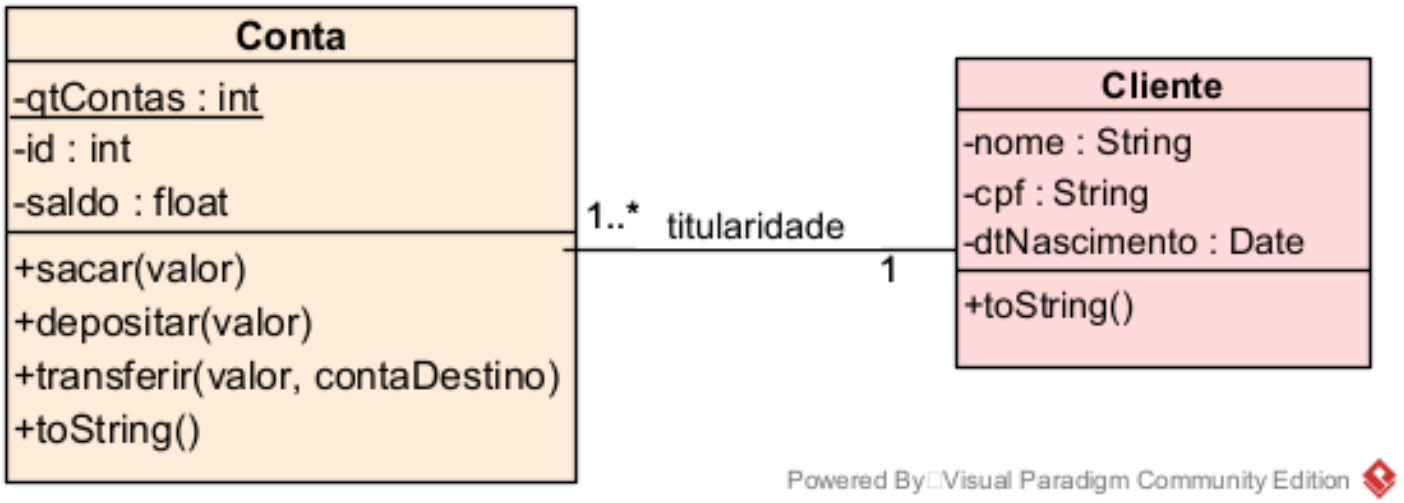
- Com isso, toda vez que o construtor de ***Conta*** for chamado, somaremos 1 ao atributo ***#qtContas*** da classe ***Conta*** como um todo, e não de um objeto específico.

- Vamos executar as instruções a seguir para verificar:

```
console.log ("Quantidade Contas do Banco = " + Conta.qtContas);
//Quantidade Contas do Banco = 0

const conta1 = new Conta(1000.00);
const conta2 = new Conta();


console.log ("Quantidade Contas do Banco = " + Conta.qtContas);
//Quantidade Contas do Banco = 2
```

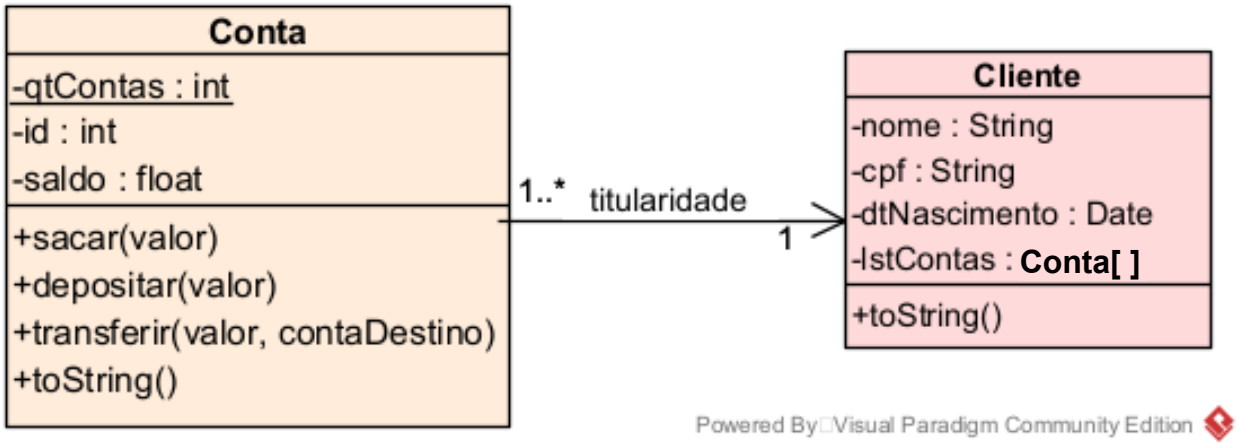
Associação entre Classes

- É o tipo mais comum de relacionamento entre classes
- Uma **associação** define que os objetos de uma classe são conectados a objetos de outra classe.
- Existe uma **associação** entre duas classes se uma instância de uma classe deve conhecer sobre a existência da outra de modo a realizar seu trabalho.



Associação entre Classes

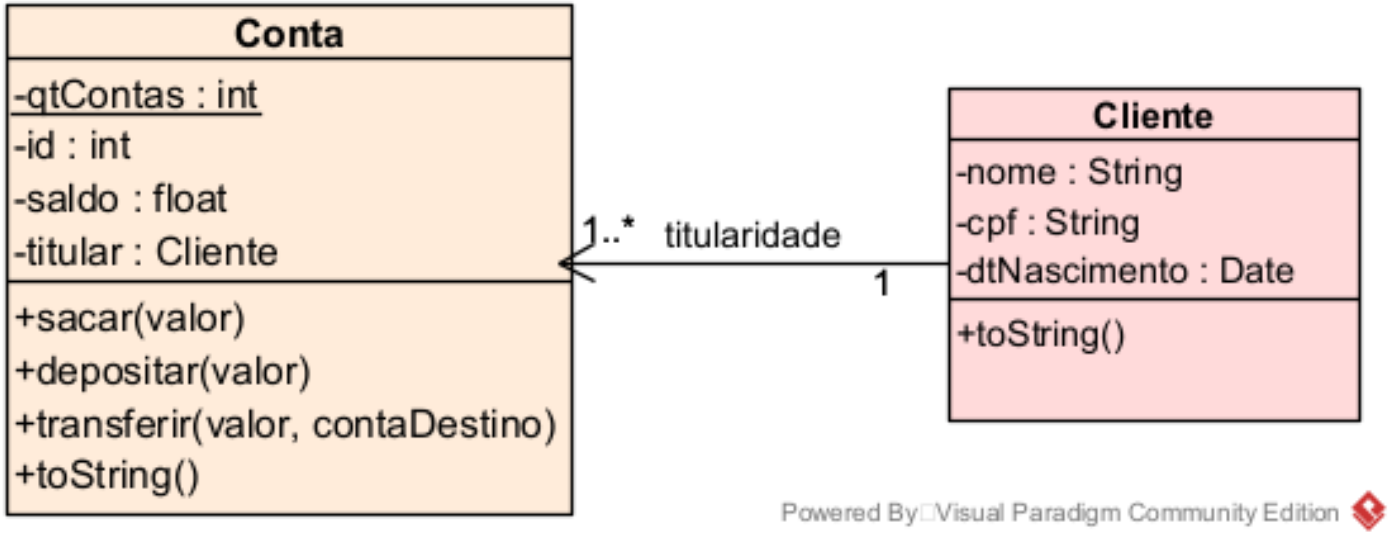
- De acordo com o diagrama, na associação entre as classes Conta e Cliente, um objeto Conta tem a titularidade de um Cliente, sendo que um objeto Cliente é titular de uma ou mais Contas.
- Para implementar essa associação o projetista poderia optar por inserir um atributo na classe Cliente, representando a lista das Contas sob a titularidade de um objeto Cliente.



Associação entre Classes

- Outra opção do projetista para implementar a associação seria inserir um atributo na classe Conta, fazendo referência ao objeto Cliente que é o titular da conta.

- A seguir é mostrada essa opção de implementação...



Associação entre Classes

```
export class Cliente {
   #nome;
   #cpf;
   #dtNascimento;

   constructor(name, cpf, birthDay) {
      this.#nome = name.toUpperCase();
      this.#cpf = cpf;
      if (birthDay != undefined){
         this.#dtNascimento = birthDay;
      } else {
         this.#dtNascimento = "2000/01/01";
      }
   }

   get nome() {
      return this.#nome;
   }

   set nome(newName) {
      if (newName == "") {
         return null;
      }
      this.#nome = newName.toUpperCase();
      return this.#nome;
   }

   get cpf() {
      return this.#cpf;
   }

   get dtNascimento() {
      return this.#dtNascimento;
   }

   set dtNascimento(newBirthDay) {
      if (newBirthDay == undefined ||
          newBirthDay == "" ||
          newBirthDay.length != 10)  {
         return null;
      }
      this.#dtNascimento = newBirthDay;
      return this.#dtNascimento;
   }

   toString(){
      return "Nome: " + this.#nome +
         "\nCPF: " + this.#cpf +
         "\nNascimento : " + this.#dtNascimento;
   }
}
```

Associação entre Classes

```
import { Cliente } from "./Cliente.js";

export class Conta {
   static #qtContas = 0; //atributo estático
   #id;
   #saldo;
   #titular;  //referência para instância de Cliente

   constructor(cliente, saldo=0.0) {
      this.titular = cliente; //chama o método set titular
      this.#saldo = saldo < 0.0 ? 0.0 : saldo;
      Conta.#qtContas++;
      this.#id = ""+ new Date().getFullYear()
                   + Conta.#qtContas;
   }

   static get qtContas() {
      return Conta.#qtContas;
   }

   get id() {
      return this.#id;
   }

   get titular() {
      return this.#titular;
   }

   set titular(cliente) {
      if (cliente != undefined && cliente instanceof Cliente) {
         this.#titular = cliente;
         return cliente;
      }
      return null;
   }

   get saldo() {
      return this.#saldo;
   }

   sacar(valor) {
      if (valor <= this.#saldo) {
         this.#saldo -= valor;
         return true;
      }
      return false;
   }

   //... Continua ...
```

Associação entre Classes

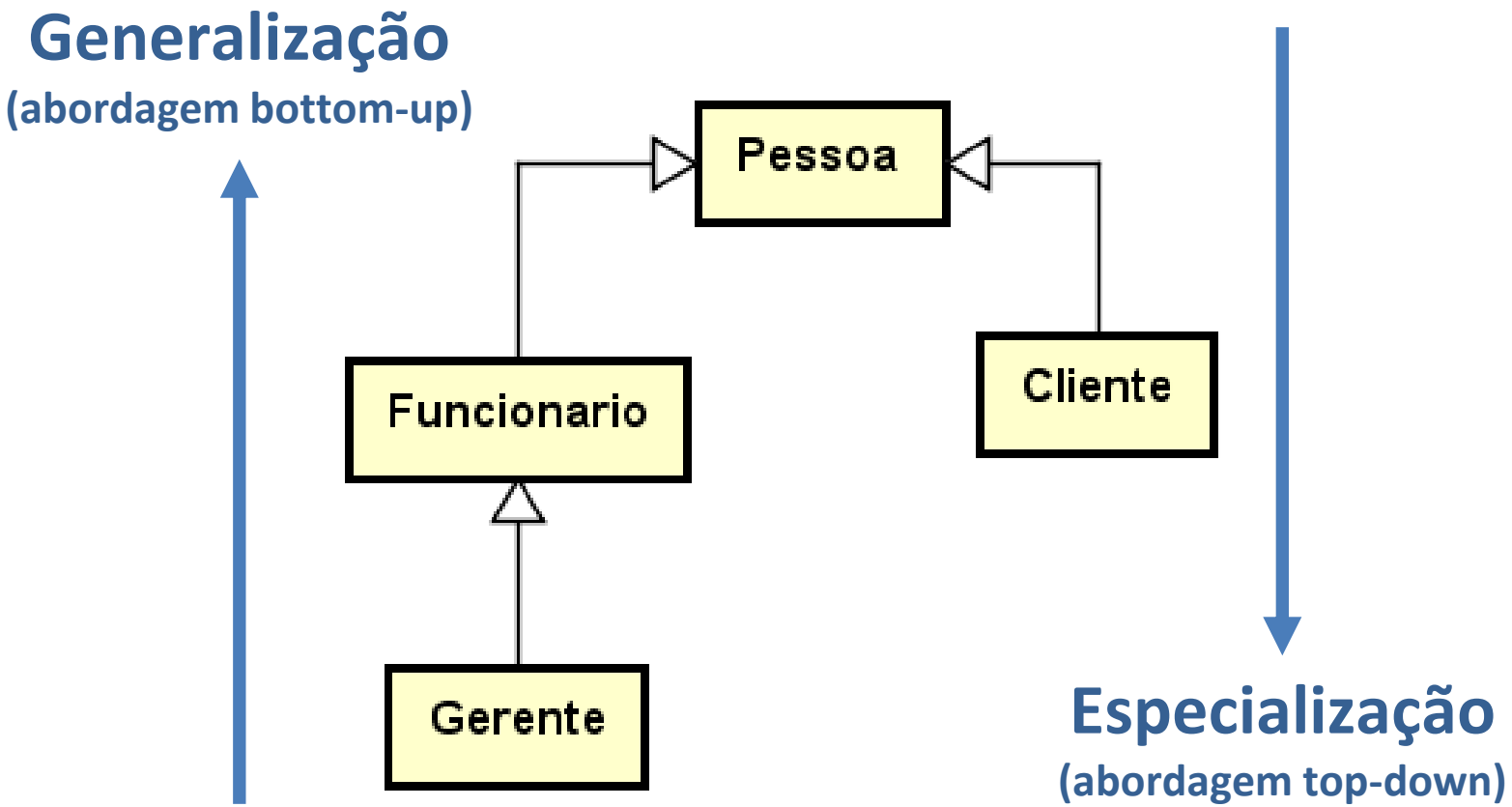
```
//... Continuação do código da classe Conta ...

  depositar(valor) {
     if (valor > 0.00) {
        this.#saldo += valor;
        return true;
     }
     return false;
  }

  transferir(valor, contaDestino) {
     if (contaDestino instanceof Conta &&
         this.sacar(valor)) {
        contaDestino.depositar(valor);
        return true;
     }
     return false;
  }

  toString() {
      return ("Nº Conta = " + this.#id +
                "\nTitular = " + this.#titular.toString() +
                "\nSaldo = R$" + this.#saldo.toFixed(2));
  }
}
```

Associação entre Classes

```
/*Vamos usar as instruções a seguir para testar os códigos das classes Cliente e Conta */
import {Conta} from "./Conta.js";
import {Cliente} from "./Cliente.js";

const cliente1 = new Cliente("Bruno“, "11122233344", "1997/09/03");
const cliente2 = new Cliente("Milton", "99988877766", "1990/05/20");

console.log ("Quantidade Contas do Banco = " + Conta.qtContas);

const conta1 = new Conta(cliente1, 1000.00);
const conta2 = new Conta(cliente2);

console.log ("\nQuantidade Contas do Banco = " + Conta.qtContas);

console.log ("\n\n" + conta1);
console.log ("\n" + conta2);

conta2.depositar(500.00);

let valorTransf = 250.00;
conta1.transferir(valorTransf, conta2);

console.log ("\n\n" + conta1.toString());
console.log ("\n" + conta2.toString());
```

Herança

- O conceito de **herança** está fundamentado nas relações de **Generalização** ou **Especialização** entre classes



Herança (abordagem top-down)

- Na especialização, novas classes são definidas, HERDANDO os atributos e métodos de uma classe já existente, sendo aprimoradas (**especializadas**) com atributos e métodos novos ou modificados.
  - A classe usada como referencial pode ser chamada de **superclasse, classe-pai(ou mãe) ou classe-base**
  - As novas classes criadas a partir de outra passam a ser suas **subclasses, classes-filhas, classes derivadas ou classes especializadas**
  - Podemos redefinir métodos e criar novos atributos nas subclasses

Herança (abordagem top-down)

- Exemplo de especialização:
  - Admita que inicialmente já existisse a implementação da classe Pessoa, com seus atributos (nome, CPF, dtNascimento, endereço, telefone, email) e métodos;
  - Ao implementar as novas classes Funcionario e Cliente, percebe-se que suas instâncias devem ter as mesmas propriedades e métodos implementados na classe Pessoa (afinal, **um funcionário ou um cliente de uma empresa são pessoas**), além de possuírem outras propriedades e métodos específicos (por exemplo, um funcionário tem um salário e está lotado em um departamento da empresa).
  - Ou seja, uma instância de Funcionario (ou Cliente) é uma especialização de Pessoa (**herda todos os atributos e métodos de Pessoa, reaproveitando os códigos já implementados**) e possui alguns atributos e métodos específicos de Funcionário (ou Cliente).

Herança (abordagem bottom-up)

- Outra possibilidade de implementar herança é definir uma nova classe a partir da **generalização** de atributos e métodos comuns entre diferentes. Por exemplo:
  - Admita que inicialmente já existissem implementadas as classes Funcionario e Cliente e que elas possuíssem atributos e métodos semelhantes entre si (por exemplo, nome, CPF, dtNascimento, endereço, telefone, email);
  - Nesse caso, pode-se generalizar seus atributos e métodos comuns em uma superclasse Pessoa, a fim de simplificar a implementação (**reaproveitamento de código**) e posterior manutenção do código.

Herança

- Mecanismo que permite o compartilhamento de atributos e operações entre classes com base em um relacionamento hierárquico (**reutilização de códigos**)
  - A **instância da subclasse** também **é uma instância da superclasse**, pois possui tudo aquilo definido na superclasse, além de atributos e operações adicionados pela subclasse.

Herança em JavaScript

- Vamos ver como poderíamos implementar a herança entre uma superclasse ***Pessoa*** e uma subclasse ***Funcionario...***

```
export class Pessoa {
        #nome;
        #cpf;
        #dtNascimento;

        constructor (nome, cpf, dtNasc) {
                this.#nome = nome;
                this.#cpf = cpf;
                this.#dtNascimento = dtNasc;
        }
        /*... as implementações dos métodos get e set seguem
           os códigos mostrados anteriormente para a classe Cliente
        */
}
```

Herança em JavaScript

```
import { Pessoa } from "./Pessoa.js/";

export class Funcionario extends Pessoa{
        #matricula;
        #salario;

        constructor (matricula, nome, cpf , dtNasc, salario=0.0){
                super(nome, cpf, dtNasc);
                this.#matricula = matricula;
                this.#salario = salario;
        }
        toString() {
                return ("Matricula: " + this.#matricula +
                "\nNome: " + super.nome +
                "\nCPF: " + super.cpf +
                "\nNascimento: " + super.dtNascimento +
                "Salário: " + this.#salario);
        }
}
```

Herança em JavaScript

- Perceba que quando definimos o **construtor** de uma subclasse, devemos chamar o construtor da superclasse
  - Tal solicitação pode ser efetuada por meio do operador ***super***

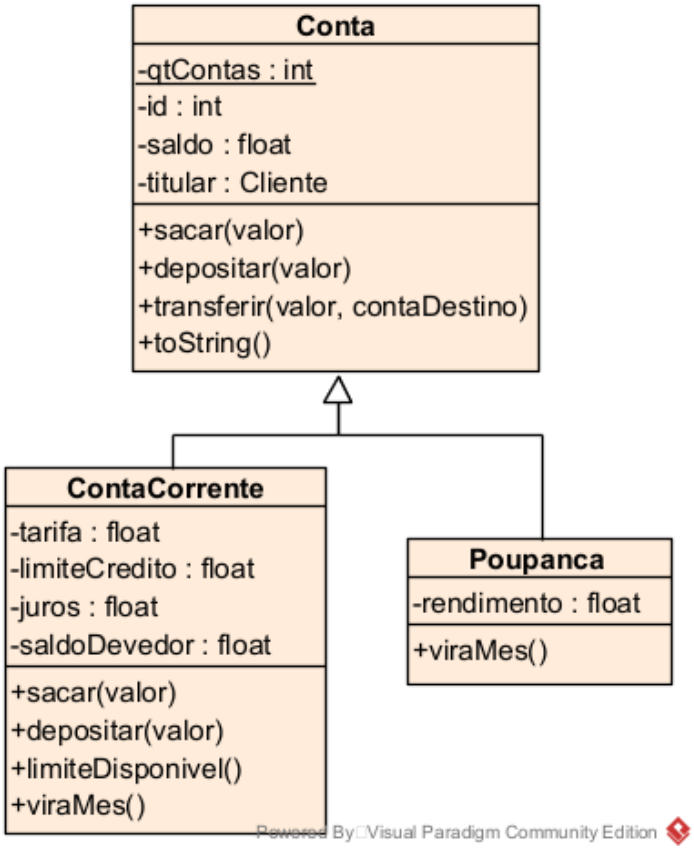
- Além disso, a subclasse não tem acesso aos **atributos *privados*** da superclasse
  - É necessário usar os métodos ***get*** e ***set*** definidos na superclasse para acessar os atributos dela

Exercitando Herança em JavaScript

- Para exercitar seus conhecimentos, use a implementação da classe Conta, com seus atributos e métodos.

- Crie duas subclasses que herdam da classe Conta: ContaCorrente e Poupanca.
  - **o método viraMes, deve debitar a tarifa de administração e os juros (caso o saldo esteja negativo) das ContaCorrente e acrescentar o rendimento nas Poupanca.**

- Faça uma aplicação que cria instâncias das subclasses e teste os resultados da execução dos métodos.



Herança e Sobreposição em JavaScript

- **Sobreposição** (ou sobrescrita) de métodos
  - Quando usamos herança, além de definir novos métodos e atributos na subclasse, podemos também **redefinir um método** na subclasse **com a mesma assinatura** do método da superclasse
  - Esse procedimento é conhecido por ***override***

- Por exemplo, o método ***sacar (valor)*** de ***Conta*** precisa ser sobrescrito na subclasse ***ContaCorrente***, uma vez que há um limite de crédito, permitindo sacar valor acima do saldo (*Obs: o método depositar também precisa ser sobrescrito*)

```
sacar(valor) {
   if (valor <= (super.saldo + this.#limiteCredito - this.#saldoDevedor) ){
      if (!super.sacar(valor)){ // se não conseguiu sacar
         valor -= super.saldo;
         super.sacar(super.saldo);
         this.#saldoDevedor += valor;
      }
      return true;
   }
   return false;
}
```

Herança e Sobreposição em JavaScript

- A técnica de **sobreposição** permite redefinir um método na subclasse que tenha mesma assinatura, mas com comportamento diferente do método na superclasse
  - Suponha que, no final do ano, os funcionários comuns do banco recebam uma bonificação de 10% sobre o valor dos seus salários, enquanto os gerentes recebem 15%

```
export class Funcionario extends Pessoa{
    #matricula;
    #salario;
    // ...
    bonificacao ( ) {
        return  this.#salario * 0.10;
    }
}
```

```
export class Gerente extends Funcionario {
    // ...
    bonificacao ( ) {
        return  super.salario * 0.15;
    }
}
```

Polimorfismo em JavaScript

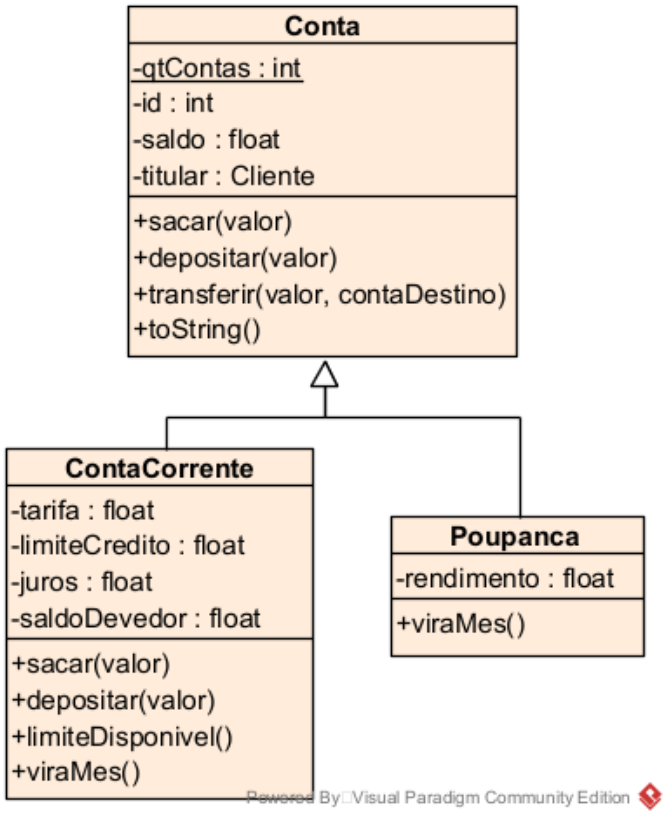
- O termo polimorfismo é originário do grego e significa "**muitas formas**"
- O polimorfismo permite escrever **programas que processam objetos que compartilham a mesma superclasse** (direta ou indiretamente) **como se todos fossem objetos da superclasse**, visando simplificar a programação.
- Para usarmos polimorfismo, precisamos que exista **herança** entre as classes e que haja **sobreposição** de algum método.
  - Em outras palavras, podemos ver o polimorfismo como a possibilidade de um mesmo método ser executado de formas diferentes de acordo com a classe do objeto que aciona o método.

Polimorfismo em JavaScript

- Polimorfismo é a capacidade de **um objeto poder ser referenciado de várias formas** (*cuidado, polimorfismo não quer dizer que o objeto fica se transformando, muito pelo contrário, um objeto é criado de uma classe e permanece daquela mesma classe, o que pode mudar é a maneira como ele é referenciado*)
  - Por exemplo, um **objeto** da subclasse ***Gerente***, também pode ser **referenciado como** um objeto da superclasse ***Funcionario***.
  - Dessa forma, podemos ver o polimorfismo como a possibilidade de "um mesmo método" ser executado de formas diferentes, de acordo com a classe do objeto que aciona o método (na verdade são métodos diferentes, com a mesma assinatura) .

Polimorfismo em JavaScript – Exercício 1

- Faça outra aplicação, agora para testar a hierarquia das classes Conta, ContaCorrente e Poupanca.
- Crie um vetor de Contas e insira nele algumas instâncias de ContaCorrente e Poupanca.
- A aplicação deve inicialmente mostrar a relação de contas usando o método toString().
- Em seguida, execute o método viraMes() de cada conta do vetor e torne a mostrar a relação de contas com seus saldos atualizados.
- Use a ferramenta de debug para perceber o polimorfismo dos métodos toString() e viraMes().



Polimorfismo em JavaScript – Exercício 2

- Considerando a hierarquia de classes mostrada na figura abaixo, crie uma aplicação para explorar os conceitos de herança e polimorfismo.
- Crie um vetor de Funcionários e insira nele algumas instâncias de Gerente e de Atendente.
- Mostre a relação de funcionários usando o método toString() e suas respectivas bonificações usando o método bonificacao().
- Execute a aplicação usando a ferramenta de debug para perceber o funcionamento do polimorfismo (o código de qual método é efetivamente executado)