

Orientação a Objetos

Herança,
Sobrescrita, Sobrecarga e
Classes Abstratas

Herança

Relacionamento do tipo “é um”

Novos Requisitos

- ▷ Em nosso exemplo de um sistema bancário, estamos implementando uma classe simples para representar uma conta.
- ▷ Neste momento, surge a necessidade de trabalhar, além da nossa conta que criamos anteriormente, com um tipo específico de conta, chamado Conta Corrente.
 - Este tipo de conta se diferencia da conta anterior por existir um limite em reais e por permitir que aconteça saque sem a existência de saldo até o limite da conta.



Como implementar isso?

Implementando Novos Requisitos

- ▷ Você pode:
 - Criar um atributo que indique qual o tipo da conta
 - Criar um atributo limite para controlar qual o limite da conta
 - Modificar o método Sacar para passar a permitir o saque considerando o limite da conta.



Problemas gerados: conta que não seja conta corrente não possui um limite, logo deve-se criar uma série de condicionais (if's) para controlar a atribuição de limite de forma indevida e para controlar o saque em acordo com o tipo de conta.



Solução:

A herança permite a uma classe herdar os atributos e operações de outra classe, podendo sobrepor operações, se desejado.

Herança

- ▷ Herança então implica em uma hierarquia de generalização/especialização, onde uma subclasse especializa a estrutura ou comportamento mais geral de sua superclasse.
 - A classe mais geral é conhecida como Classe Pai ou Superclasse.
 - A classe mais específica é conhecida como Classe Filho ou Subclasse.

Herança

- ▷ **Cuidado:**
 - Teste para herança:
 - Se B não é uma espécie de A, então B não pode herdar de A.
- ▷ A linguagem Java permite somente Herança Simples, onde uma subclasse pode herdar de somente uma superclasse.

Herança

- ▷ Nova classe:
 - ContaCorrente

```
public class ContaCorrente extends Conta {  
  
}
```

- Em Java, a herança é feita utilizando, após o nome da subclasse, a palavra reservada `extends` seguida do nome da superclasse.
- Como consequência, todos os atributos e operações da superclasse também existirão em objetos instanciados a partir da subclasse.

Herança

- ▷ Nova classe:
 - ContaCorrente

```
public class ContaCorrente extends Conta {  
  
}
```

- ▷ No Main:

```
Conta c1 = new Conta();
```

c1.

getCorrentista()	Pessoa
setCorrentista(Pessoa correntista)	void
setNumero(int numero)	void
getNumero()	int
depositar(double quantia)	void
getSaldo()	double
sacar(double quantia)	void

Herança

- ▷ Nova classe:
 - ContaCorrente

```
public class ContaCorrente extends Conta {  
  
}
```

- ▷ No Main:

```
ContaCorrente c2 = new ContaCorrente();
```

```
c2.
```

getCorrentista()	Pessoa
depositar(double quantia)	void
setNumero(int numero)	void
setCorrentista(Pessoa correntista)	void
getNumero()	int
getSaldo()	double
sacar(double quantia)	void

Herança

- ▶ Além dos atributos e operações da superclasse, a subclasse pode possuir atributos e operações próprias.

```
public class ContaCorrente extends Conta {  
  
    public double getLimite() {  
        return limite;  
    }  
  
    public void setLimite(double limite) {  
        this.limite = limite;  
    }  
  
    private double limite;  
}
```

Exercícios

1) Conforme visto, no exemplo do nosso sistema bancário, uma Pessoa possui nome e email. Precisamos em nosso sistema representar dois tipos de pessoas: Pessoa Física (ser humano enquanto indivíduo) e Pessoa Jurídica (empresa, sociedade ou organização). Uma pessoa física possui nome, e-mail e data de nascimento. Uma pessoa jurídica possui nome, e-mail e contato, sendo contato o nome da pessoa que deve ser contactada se enviado um e-mail. Implemente isso em nosso sistema.

Sobrescrita

Sobreposição ou sobrescrita



Como implementar o saque da conta corrente, que deve considerar o limite?

Sobrescrita / Sobreposição

- ▷ O método sacar pode ser implementado na classe ContaCorrente.

```
public class ContaCorrente extends Conta {  
    @Override  
    public void sacar(double quantia) throws Exception {  
        if ((saldo + limite) < quantia)  
            throw new Exception("Saldo insuficiente!");  
  
        saldo -= quantia;  
    }  
}
```

Sobrescrita / Sobreposição

- ▷ ● O annotation `@Override` é opcional e indica que o método está sobrepondo o método sacar da classe Conta.
 - Apesar de opcional, é uma boa prática a sua utilização.

```
public class ContaCorrente extends Conta {  
    @Override  
    public void sacar(double quantia) throws Exception {  
        if ((saldo + limite) < quantia)  
            throw new Exception("Saldo insuficiente!");  
  
        saldo -= quantia;  
    }  
}
```

Sobrescrita / Sobreposição

- ▷ O método sacar pode ser implementado na classe ContaCorrente.

- Quanto o método sacar de um objeto do tipo ContaCorrente for chamado, esta é a implementação que será executada.

```
public void sacar(double quantia) throws Exception {  
    if (saldo + limite) < quantia  
        throw new Exception("Saldo insuficiente!");  
  
    saldo -= quantia;  
}  
}
```

Sobrescrita / Sobreposição

- ▶ ● Para que o saldo possa ser acessado apenas dentro da classe Conta e dentro de qualquer classe que herda de Conta, troque o modificador de acesso do atributo saldo da classe Conta para protected (protegido). Ficaré assim:
protected double saldo;

```
public void sacar(double quantia) throws Exception {  
    if (saldo + limite) < quantia)  
        throw new Exception("Saldo insuficiente!");  
  
    saldo -= quantia;  
}  
}
```

Sobrescrita / Sobreposição

▷ No Main:

```
Conta c1 = new Conta();  
c1.depositar(100);
```



```
try  
{  
    c1.sacar(100.01);  
    System.out.printf("Novo saldo: R$%.2f",  
        c1.getSaldo());  
}  
catch (Exception ex)  
{  
    System.out.println(ex.getMessage());  
}
```

Saldo insuficiente!

X

Sobrescrita / Sobreposição

▷ No Main:

```
ContaCorrente c2 = new ContaCorrente();  
c2.setLimite(50);  
c2.depositar(100);
```



```
try  
{  
    c2.sacar(100.01);  
    System.out.printf("Saldo + limite: R$%.2f",  
        c2.getSaldo() + c2.getLimite());  
}  
catch (Exception ex)  
{  
    System.out.println(ex.getMessage());  
}
```

Saldo + limite: R\$49,99

X

Exercícios

2) Continuando o nosso exemplo do sistema bancário, na classe `ContaCorrente`, sobrescreva o método `transferir` da classe `Conta`.

Sobrecarga

Sobrecarga de Métodos

Sobrecarga

- ▷ Veja novamente o método transferir da classe Conta:

```
public void transferir(double quantia, Conta destino) throws Exception {  
    if (this.saldo < quantia)  
        throw new Exception("Saldo insuficiente!");  
  
    this.sacar(quantia);  
    destino.depositar(quantia);  
}
```



Como implementar um segundo método transferir que transfere uma determinada quantia para um conjunto de contas?

Sobrecarga

- ▷ É possível fazer uma sobrecarga de método.
 - Temos sobrecarga de métodos quando dois ou mais métodos de uma classe possuem o mesmo nome, porém cada método possui uma assinatura diferente.
 - A assinatura de um método é a linha que contém o seu tipo de retorno, nome e lista de parâmetros.
- ▷ Enquanto a sobreposição ou sobrescrita é conhecida pelo termo `override`, a sobrecarga é conhecida pelo termo `overload`.

Sobrecarga

▷ Método transferir sobrecarregado:

```
public void transferir(double quantia, Conta destino) throws Exception {
    if (this.saldo < quantia)
        throw new Exception("Saldo insuficiente!");

    this.sacar(quantia);
    destino.depositar(quantia);
}

public void transferir(double quantia, Conta[] destinos) throws Exception {
    if (this.saldo < quantia * destinos.length)
        throw new Exception("Saldo insuficiente!");

    this.sacar(quantia * destinos.length);

    for (Conta destino: destinos) {
        destino.depositar(quantia);
    }
}
```

Classes Abstratas

Métodos e Classes Abstratas

Motivação

- ▷ Nosso sistema bancário, até o momento, possui dois tipos de contas: Conta e ContaCorrente.
- ▷ É necessária a criação de um outro tipo de conta:
 - Conta Poupança
 - O saldo é a soma dos depósitos e são guardados por data.
 - Saques são realizados nos depósitos mais antigos.

Motivação

- ▷ O nosso sistema passará a ter dois tipos de contas:
 - ContaCorrente
 - ContaPoupanca
- ▷ A classe Conta não pode virar ContaPoupanca, pois ContaCorrente herda dela e não possui as restrições de depósito e saque da ContaPoupanca.
- ▷ A classe ContaPoupanca não pode herdar da classe ContaCorrente, pois não possui limite.



Se ContaCorrente e
ContaPoupanca herdarem
de Conta, como faço para
que não possa ser
instanciada uma conta que
não seja ContaCorrente e
nem ContaPoupanca?

Classes Abstratas

- ▷ Uma classe abstrata é uma classe que **não** pode ser instanciada.
 - Se colocarmos a classe Conta como abstrata, não será possível instanciar objetos do tipo Conta.
 - Apenas será possível instanciar objetos do tipo ContaCorrente ou ContaPoupanca.

Classes Abstratas

- ▷ A classe Conta passa a ser abstrata:

```
public abstract class Conta {  
    ...  
}
```

Classes Abstratas

- ▷ A classe Conta passa a ser abstrata:

```
public abstract class Conta {  
    ...  
}
```

- ▷ Agora não é possível instanciar a classe Conta no Main:

```
Conta c1 = new Conta();
```

Motivação

- ▷ As operações sacar e depositar, agora não fazem mais sentido de existirem na classe Conta, uma vez que suas implementações são diferentes entre os dois tipos de contas que temos (ContaCorrente e ContaPoupanca)
- ▷ Já temos a previsão de criarmos em um futuro próximo mais um tipo de conta e, neste momento, a nossa preocupação se volta para uma questão:



Como padronizar os nomes dos métodos sacar e depositar dentre as subclasses existentes e as futuras?

Motivação

- ▷ O método transferir parece ser diferente apenas na subclasse ContaCorrente, o que pode nos indicar que o mesmo pode ser implementado na classe Conta e sobrescrito apenas na classe ContaCorrente.
- ▷ Daí surge uma nova questão:



O método transferir depende dos métodos sacar e depositar, que não mais existem na classe Conta. Como solucionar isso?

Métodos Abstratos

- ▷ Podemos utilizar métodos abstratos!
 - Um método abstrato é um método sem implementação.
 - Uma classe não abstrata que herda de uma superclasse que possui métodos abstratos é obrigada a sobrescrever esses métodos.

Métodos Abstratos

- ▷ Uma classe que possui ao menos um método abstrato, precisa necessariamente ser abstrata.
 - Isso faz sentido, pois se esta obrigação não existisse, ao chamar o método, o mesmo não teria implementação, o que causaria um erro.

Métodos Abstratos

- ▷ O método transferir implementado na classe Conta poderá continuar chamando os métodos sacar e depositar se colocarmos esses métodos como abstratos na superclasse Conta.
 - Isso porque as subclasses concretas ContaCorrente e ContaPoupanca serão obrigadas a implementar esses métodos.
- ▷ Veja nos próximos slides como fica o nosso código:

Métodos Abstratos

▷ Classe Conta:

```
public abstract class Conta {  
    public abstract void depositar(double quantia);  
  
    public abstract void sacar(double quantia) throws Exception;  
  
    ...  
}
```

Métodos Abstratos

- ▷ Nova classe ContaPoupancaSaldo:

```
import java.time.LocalDate;

public class ContaPoupancaSaldo {
    private double saldo;
    private LocalDate data;

    public double getSaldo() {
        return saldo;
    }

    public void setSaldo(double saldo) {
        this.saldo = saldo;
    }
}
```

Métodos Abstratos

- ▷ Nova classe ContaPoupancaSaldo:

```
public LocalDate getData() {  
    return data;  
}  
  
public void setData(LocalDate data) {  
    this.data = data;  
}  
}
```

Métodos Abstratos

▷ Classe ContaCorrente:

```
public class ContaCorrente extends Conta {  
    @Override  
    public void depositar(double quantia)  
    {  
        this.saldo += quantia;  
    }  
  
    ...  
}
```

Métodos Abstratos

▷ Classe ContaPoupanca:

```
import java.util.ArrayList;
import java.time.LocalDate;

public class ContaPoupanca extends Conta {
    @Override
    public void depositar(double quantia) {
        var saldoDia = obterSaldoDia(LocalDate.now());
        saldoDia.setSaldo(saldoDia.getSaldo() + quantia);
    }
}
```

Métodos Abstratos

▷ Classe ContaPoupanca:

```
@Override
public void sacar(double quantia) throws Exception {
    if (getSaldo() < quantia) {
        throw new Exception("Saldo insuficiente!");
    }

    var primeiro = saldos.getFirst();

    while (primeiro.getSaldo() < quantia) {
        quantia -= primeiro.getSaldo();
        saldos.removeFirst();

        primeiro = saldos.getFirst();
    }

    primeiro.setSaldo(primeiro.getSaldo() - quantia);
}
```


Métodos Abstratos

▷ Classe ContaPoupanca:

```
private ContaPoupancaSaldo obterSaldoDia(LocalDate data) {  
    ContaPoupancaSaldo saldoDia = null;  
  
    if (!saldos.isEmpty() &&  
        saldos.getLast().getData().equals(data)) {  
        saldoDia = saldos.getLast();  
    }  
  
    if (saldoDia == null) {  
        saldoDia = new ContaPoupancaSaldo();  
        saldoDia.setData(LocalDate.now());  
        saldos.add(saldoDia);  
    }  
  
    return saldoDia;  
}
```

Métodos Abstratos

▷ Classe ContaPoupanca:

```
@Override
public double getSaldo() {
    double saldo = 0.0;

    for (var s: saldos) {
        saldo += s.getSaldo();
    }

    return saldo;
}

private ArrayList<ContaPoupancaSaldo> saldos =
    new ArrayList<ContaPoupancaSaldo>();
}
```

Exercícios

3) Em um determinado sistema, existem três tipos de funcionários:

- ▷ Funcionário Mensalista: salário líquido corresponde ao salário mensal subtraído do valor dos impostos.
- ▷ Funcionário Diarista: salário líquido corresponde ao valor por dia multiplicado pelo número de dias trabalhados e o resultado subtraído do valor dos impostos.
- ▷ Funcionário Horista: salário líquido corresponde ao valor por hora multiplicado pelo número de horas trabalhadas e o resultado subtraído do valor dos impostos.

Implemente as classes necessárias para controlar esses tipos de funcionários.

Fim de material

Dúvidas?