# LAB 6

Neste laboratório você irá aprender a criar e usar suas próprias classes.

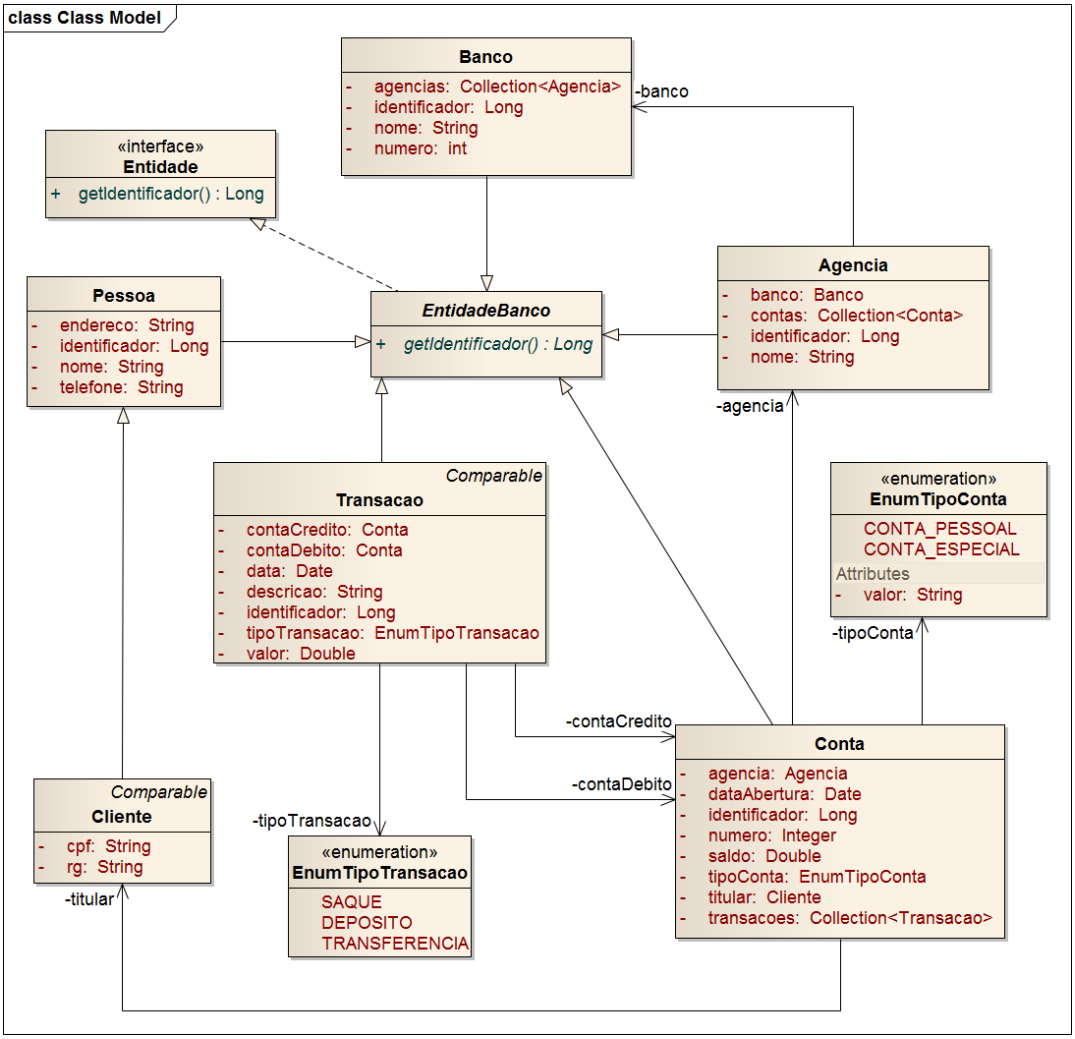
A patir deste laboratório os exercícios dos próximos laboratórios possuem dependências, todos os exercícios devem ser realizados. Leia atentamente os comentários nos códigos citados, eles são importantes para você entender o que está fazendo ou para qual utilidade do código.

O projeto destes laboratórios a partir deste é chegar ao final do curso com uma aplicação completa representando o sistema de um banco. É muito importante que seja feito os exercícios corretamente para que a aplicação não tenha comportamentos estranhos.

Em caso de dúvida para resolver um exercício consulte o instrutor para a melhor solução ou para descobrir o erro.

Apesar da ferramenta sugerir o que deve ser feito para resolver alguns erros de compilação, se você não souber ou não entender a solução proposta, então não a faça, e tome cuidado com a importação de classes de pacotes diferente do solicitado.

O diagrama abaixo mostra como ficará a parte de modelo do projeto, onde um cliente terá UMA ou várias CONTAS, uma CONTA é de apenas uma AGÊNCIA e uma AGÊNCIA é de um banco, agora lendo o contrário, um BANCO não tem nenhuma ou tem várias AGÊNCIAS, uma AGÊNCIA não tem nenhuma ou tem várias CONTAS, uma CONTA não tem nenhum ou tem vários clientes.




**Duração prevista: 60 minutos**

## Exercícios

**Exercício 1**: Definindo e usando classes

**Exercício 2**: Membros estáticos

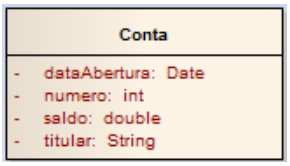
**Exercício 3**: Sobrecarga

**Exercício 4**: Construtores

**Exercício 5**: referencia “this”

# Definindo e usando classes

1 - Crie um novo projeto Java no eclipse com o nome **Banco.**
Após criar o projeto, crie a classe Conta conforme a especificação UML abaixo.




```
import java.util.Date;

public class Conta {

	private int numero;

	private String titular;

	private double saldo;

	private Date dataAbertura;

}
```

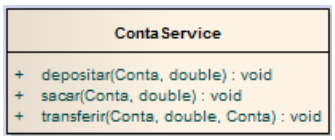
2 - Crie os métodos **getter's e setter's** para todos os atributos da classe **Conta** conforme exemplo abaixo:

```
public int getNumero() {

	return numero;
}
public void setNumero(int numero) {

	this.numero = numero;
}
```

3 - Crie a classe **ContaService** com os métodos **depositar(), sacar() e transferir()** conforme a especificação UML abaixo.




```
public class ContaService {

	public void depositar(Conta contaDestino, double valor) {
		contaDestino.setSaldo(contaDestino.getSaldo() + valor);
	}

	public void sacar(Conta contaSaque, double valor) {
		contaSaque.setSaldo(contaSaque.getSaldo() - valor);
	}

	public void transferir(Conta contaSaque, double valor, Conta contaDestino) {
		this.sacar(contaSaque, valor);
		this.depositar(contaDestino, valor);
	}
}
```

4 - Crie a classe **TestaConta,** através desta classe iremos instanciar objetos da classe Conta e testar os métodos da classe ContaService.

```
import java.util.Scanner;

public class TestaConta {

	public static void main(String[] argv) {

	//Objeto para ler dados via console
	Scanner c = new Scanner(System.in);

	//Declara e inicializa a variavel saldoConta
	System.out.println("Digite o saldo inicial da conta");
	double saldoConta = c.nextDouble();

	//Declara e inicializa o numero da conta
	System.out.println("Digite o numero da conta");
	int numeroConta = c.nextInt();

	//Cria uma instância de ContaService onde está presente as operações para Objeto Conta
	ContaService operacoesConta = new ContaService();

	//Cria uma instância da classe Conta
	Conta conta1 = new Conta();

	//Altera o valor dos atributos da instância conta criada
	conta1.setNumero(numeroConta);
	conta1.setSaldo(saldoConta);

	//Cria uma nova instância da classe Conta
	Conta conta2 = new Conta();

	//Imprime os dados do objetos conta1
```

```
        System.out.println("O numero da Conta1 :" + conta1.getNumero());
        System.out.println("O saldo da Conta1 :" + conta1.getSaldo());

        //Chama o método depositar para adicionar saldo na conta
        System.out.println("Será creditado 100 reais na conta ");
        operacoesConta.depositar(conta1, 100.00);
        System.out.println("Saldo da Conta1  :" + conta1.getSaldo());

        //Chama o método sacar para debitar no saldo da conta
        System.out.println("Será debitado 56.43 reais na conta ");
        operacoesConta.sacar(conta1, 56.43);
        System.out.println("Saldo da Conta :" + conta1.getSaldo());

        //Imprime o saldo das contas 1 e 2.
        System.out.println("Saldo da Conta 1 :" + conta1.getSaldo());
        System.out.println("Saldo da Conta 2 :" + conta2.getSaldo());


        //Chama o método tranferir para debitar na conta 1 e creditar na conta 2
        System.out.println("Transferir 50.00 de conta 1 para conta2 ");
        operacoesConta.transferir(conta1, 50.00, conta2);

        //Imprime o saldo das contas 1 e 2.
        System.out.println("Saldo da Conta 1 :" + conta1.getSaldo());
        System.out.println("Saldo da Conta 2 :" + conta2.getSaldo());

    }
}
```

## Desafio para o aluno

1 - Na classe **TestaConta,** crie uma terceira instância da classe **Conta** com o nome da variável de referência **conta3.**

As seguintes operações devem ser realizadas:

1.1 - Transferir a metade do valor da conta2 para conta3.

1.2- Após realizar a transferência, mostrar o saldo de cada conta.

2 – Faça a seguinte alteração:

2.1 - Sempre que um objeto da classe Conta for criado, o atributo de instância **dataAbertura** deverá receber a data atual da criação automaticamente.

**Dica:** Esta alteração deve ser realizada no construtor da classe Conta.

# Membros estáticos

## Criando aplicativos que usam variáveis estáticas

1 - Crie a seguinte classe **UtilData**. Esta classe será uma classe utilitária para podermos manipular datas no projeto, ela define **variáveis estáticas** que representam os nomes dos dias da semana em Português.

```
import java.util.Calendar;
import java.util.Date;

public class UtilData {

	//DiaDaSemana que representa Domingo
	static int DOMINGO = Calendar.SUNDAY;

	//DiaDaSemana que representa Segunda-Feira
	static int SEGUNDA = Calendar.MONDAY;

	//DiaDaSemana que representa Terça-Feira
	static int TERÇA = Calendar.TUESDAY;

	//DiaDaSemana que representa Quarta-Feira
	static int QUARTA = Calendar.WEDNESDAY;

	//DiaDaSemana que representa Quinta-Feira
	static int QUINTA = Calendar.THURSDAY;

	//DiaDaSemana que representa Sexta-Feira
	static int SEXTA = Calendar.FRIDAY;

	//DiaDaSemana que representa Sábado
	static int SÁBADO = Calendar.SATURDAY;

	// MesDoAno que representa Janeiro
	int JANEIRO = Calendar.JANUARY;

	// MesDoAno que representa Fevereiro
	int FEVEREIRO = Calendar.FEBRUARY;

	// MesDoAno que representa Março
	int MARÇO = Calendar.MARCH;

	// MesDoAno que representa Abril
	int ABRIL = Calendar.APRIL;

	// Dia do Mês
	static int DiaDoMes = Calendar.DAY_OF_MONTH;

	// Dia da semana
	static int DiaDaSemana = Calendar.DAY_OF_WEEK;

	// Método estático anônimo. As instruções dentro deste bloco
	// estático são executadas quando a classe é carregada,
	// ou seja, somente uma vez.
	static {
		System.out.println("Entrando no bloco estático.");
		Date data = Calendar.getInstance().getTime();
		System.out.println("Saindo do método estático data = " + dateToStringEstatico(data));
	}
```

```
        // método estático que retorna o valor da data formatado como String
        static String agora(Date data) {
                return new java.text.SimpleDateFormat("dd/MM/yyyy  HH:mm").format(data);
        }

    // método de instância que retorna o valor da data formatado como String
        String DDMMAAAAHHMM(Date data) {
                return new java.text.SimpleDateFormat("dd/MM/yyyy HH:mm").format(data);
        }
```

2 - Agora crie outra classe **ExemploVariavelEstatica**. Observe o uso das variáveis estáticas, veja os coméntários.

```
public class ExemploVariavelEstatica {

        public static void main(String[] args) {

                // Acessando atributos estaticos da classe UtilData
                // veja que você não precisou criar uma instância da classe UtilData
                System.out.println("Dia da semana " + UtilData.DOMINGO);
                System.out.println("Dia da semana " + UtilData.SEGUNDA);
                System.out.println("Dia da semana " + UtilData.QUARTA);
                System.out.println("Dia da semana " + UtilData.SÁBADO);

                // Acessando atributos de instância da classe UtilData
                // Você tem que criar uma instância da classe antes de você poder acessar seu valor.
                UtilData data = new UtilData();
                System.out.println("Mes do ano " + data.JANEIRO);
                System.out.println("Mes do ano " + data.FEVEREIRO);
                System.out.println("Mes do ano " + data.ABRIL);
                System.out.println("Mes do ano " + data.MARÇO);

                // O atributo estático pode ser acessado por uma variável de instância
                System.out.println("Dia da Semana " + data.DiaDaSemana);
                data.DiaDaSemana = 3;
                System.out.println("Mudou Dia da Semana " + data.DiaDaSemana);


                UtilData data2 = new UtilData();
                System.out.println("instancia 1 Dia do Mes " + UtilData.DiaDoMes);
                System.out.println("instancia 2 Dia do Mes " + data2.DiaDoMes);
                data2.DiaDoMes = 20;
                System.out.println("instancia 1 Mudou Dia do Mes " + UtilData.DiaDoMes);
                System.out.println("instancia 2 Mudou Dia do mês " + data2.DiaDoMes);
        }

}
```

3 - Modifique a classe **UtilData.java** para que os membros não estáticos (**Janeiro,Fevereiro, etc**) se tornem estáticos.

4 - Modifique a classe ExemploVariavelEstatica para imprimir os valores dos novos membros estáticos que você modificou no exercício anterior.

## Criando aplicativos que usam métodos estáticos

1 - Observe a classe abaixo, ela apresenta o uso de métodos estáticos, compile o programa e veja as notas da compilação.

```
import java.util.Date;

public class ExemploMetodoEstatico {

        public static void main(String[] args) {

          Date data = new Date();

          //Invocando metodo estático, nao é preciso instanciar a classe  UtilData
          System.out.println(UtilData.agora(data));

          //Metodo estático pode ser invocado por uma instancia da classe UtilData
          UtilData idata = new UtilData();
          System.out.println(idata.agora(data));

          //Metodo de instancia so pode ser invocado por uma instancia
          System.out.println(idata.DDMMAAAAHHMM(data));

          //Metodos de instancia nao podem ser invocados diretamente ocorre erro de compilacao
          UtilData.DDMMAAAAHHMM(data);
        }
}
```

2 - Modifique a listagem anterior de modo a corrigir o erro de compilação e poder executar o programa.

## Sobrecarga

## Adicionando métodos sobrecarregados

1 - Modifique a classe **ContaService** conforme abaixo, observe as duas versões do método **transferir()** apresentadas abaixo.

```
public class ContaService {

        public void depositar(Conta contaDestino, double valor) {
                contaDestino.setSaldo(contaDestino.getSaldo() + valor);
        }

        public void sacar(Conta contaSaque, double valor) {
                contaSaque.setSaldo(contaSaque.getSaldo() - valor);
        }

        // Para não implementar a mesma regra duplicando este método, chama o segundo transferir
        // e informa o limite com valor zero para representar que não possui saldo.
        public void transferir(Conta contaSaque, double valor, Conta contaDestino) {
                // transfere valor da conta para a conta destino
                transferir(contaSaque, valor, contaDestino, 0);
        }
```

```
	// Sobrecarga do método transferir. Quando for invocado este método
	// deverá ser informado um valor para limite (cheque especial) que será adicionado ao
	// saldo da conta para verificar se pode ocorrer a transferência.
	public void transferir(Conta contaSaque, double valor, Conta contaDestino, double limite) {

		if (( contaSaque.getSaldo() + limite ) < valor) {
			System.out.print("Saldo insuficiente para esta operação");
			return;
		}
		// transfere valor da conta para conta destino
		this.sacar(contaSaque, valor);
		this.depositar(contaDestino, valor);
	}

}
```

2 - Execute a classe **TestaConta** para testar se irá executar corretamente.

3 - Para testar a execução dos dois métodos **transferir** da classe **ContaService.java**, crie a classe **SobrecargaTransferir.java** confome abaixo:

```
public class SobrecargaTransferir {

	public static void main(String[] argv) {

		// Cria uma instância de ContaService onde está presente as operações para Objeto Conta
		ContaService operacoesConta = new ContaService();

		// cria uma instância da classe Conta
		Conta conta1 = new Conta();

		// configura instância da classe Conta
		conta1.setNumero(1234567890);
		conta1.setSaldo(500.00);

		// cria nova instancia de Conta para tranferencia
		Conta conta2 = new Conta();
		conta2.setSaldo(50.00);

		System.out.println("Transferir 400.00 de conta 1 para conta2 ");

		// tranferindo valor de conta1 para conta2 utilizando transferencia sem limite
		operacoesConta.transferir(conta1, 400.00, conta2);
		System.out.println("Saldo da Conta 1:" + conta1.getSaldo());
		System.out.println("Saldo da Conta 2:" + conta2.getSaldo());

		// tranferindo valor de conta1 para conta2 utilizando transferencia com limite
		operacoesConta.transferir(conta1, 200.00, conta2, 300);
		System.out.println("Saldo da Conta 1:" + conta1.getSaldo());
		System.out.println("Saldo da Conta 2:" + conta2.getSaldo());

	}

}
```

4 - Crie uma terceira instância da classe **Conta** com nome da variável **conta3** e transfira R$ 100.00 com e sem limite de conta2 para conta3. Mostre o saldo de cada conta antes e depois de cada transferência, usando os métodos sobrecarregados.

5 - Modifique a classe **UtilData** como mostrado abaixo. Perceba que foi alterado o tipo das variáveis **data** para **Calendar** e definido métodos utilitários para retornar o valor respectivo de uma data e para retornar uma data baseado nor argumentos passados.

```
import java.util.Calendar;
import java.util.Date;

public class UtilData {

	// representa Domingo
	static final int DOMINGO = Calendar.SUNDAY;

	// Segunda-Feira
	static final int SEGUNDA = Calendar.MONDAY;

	// Terça-Feira
	static final int TERÇA = Calendar.TUESDAY;

	// Quarta-Feira
	static final int QUARTA = Calendar.WEDNESDAY;

	// Quinta-Feira
	static final int QUINTA = Calendar.THURSDAY;

	// Sexta-Feira
	static final int SEXTA = Calendar.FRIDAY;

	// Sábado
	static final int SÁBADO = Calendar.SATURDAY;

	// Constrói uma data representando agora
	public static Date data() {

		return Calendar.getInstance().getTime();
	}

	// Constrói uma data representando um dado dia.
	// Para efetuar comparações entre datas, hora será 00:00:00.0 (0 horas, 0 minutos, 0 segundos, 0 milissegundos)
	public static Calendar data(int dia, int mes, int ano) {

		return data(dia, mes, ano, 0, 0, 0);
	}

	// Constrói uma data representando um dado dia e hora.
	// Para permitir comparações de datas, os milissegundos da data são zerados.
	public static Calendar data(int dia, int mes, int ano, int hora, int min, int seg) {

		Calendar data = Calendar.getInstance();
		data.set(ano, --mes, dia, hora, min, seg);
		data.set(Calendar.MILLISECOND, 0);
		return data;
	}

	// Retorna uma data com dia, mes e ano passado como String e formato como argumento
	public static Calendar data(String data) {
		return data(Integer.valueOf(data.split("/")[0]), Integer.valueOf(data.split("/")[1]), Integer.valueOf(data.split("/")[2]));
	}

	public static Date getDate(Calendar data) {
		return data.getTime();
	}

	// Formata uma data no formato dd/mm/aaaa
```

```
	public static String DDMMAAAA(Date data) {
		return new java.text.SimpleDateFormat("dd/MM/yyyy").format(data);
	}

	// Formata uma data no formato dd/mm/aaaa hh:mm
	public static String DDMMAAAAHHMM(Date data) {
		return new java.text.SimpleDateFormat("dd/MM/yyyy HH:mm").format(data);
	}

	// método estático que retorna o valor da data formatado como String
	public static String agora(Date data) {
		return new java.text.SimpleDateFormat("dd/MM/yyyy  HH:mm").format(data);
	}

	// Retorna o Ano correspondente a esta data
	public int getAno(Date data) {

		final Calendar calendario = Calendar.getInstance();

		calendario.setTime(data);

		return calendario.get(Calendar.YEAR);
	}

	// Retorna o mês correspondendo a esta data.
	public int getMes(Date data) {

		final Calendar calendario = Calendar.getInstance();

		calendario.setTime(data);

		return calendario.get(Calendar.MONTH);
	}

	// Retorna o dia correspondendo a esta data.
	public int getDia(Date data) {

		final Calendar calendario = Calendar.getInstance();

		calendario.setTime(data);

		return calendario.get(Calendar.DAY_OF_MONTH);
	}

	// Acrescenta um número de dias à data.
	public void somarDia(Date data, int numDias) {

		final Calendar calendario = Calendar.getInstance();

		calendario.setTime(data);

		calendario.add(Calendar.DAY_OF_MONTH, numDias);
	}

}
```

## Construtores

## Definindo múltiplos construtores numa classe

1 - Crie um construtor sobrecarregado na classe **Conta** que tenha os seguintes parâmetros:
1.1 - Nome do titular e o número da conta.
O valor de cada parâmetro deve ser atribuído em suas respectivas variáveis de instância.

2 – Dentro do construtor, sete com o valor 0.0 a variavel de instância saldo.

```
//Construtor com dois parametros
public Conta( String nome, int nconta ) {
        this.titular = nome;
        this.numero = nconta;
        this.saldo = 0.0; // Conta em reais e zerada
}
```

3 - Definimos que todo objeto da classe **Conta** quando for criado deverá ter o atributo **dataAbertura** inicializado com a data corrente. Para que não seja necessário reescrevermos esta regra de negócio neste novo construtor, devemos através do método **this()** invocar o construtor que atribui a data de criação a variável de instância **dataAbertura**.

```
public Conta( String nome, int nconta ) {
        this();//Invoca o construtor sem parâmetros definido na classe
        numero = nconta;
        titular = nome;
        saldo = 0.0;
}
```

4 – Vamos testar o comportamento dos dois construtores que definimos em nossa classe **Conta.java.**

Crie a classe **TestaConstrutor.java.** Nesta classe você irá criar duas instâncias da classe Conta, e para cada instância você irá invocar um construtor. Após a criação, imprima a data de abertura de cada objeto conta.

## Referência “this”

## Invoque métodos com “this”

1 - Nossa classe **ContaService** precisa registrar o histórico de transações (débito, crédito), para ficar mais dinâmico precisamos de uma classe que armazene as informação das transações. Então crie o enum **EnumTipoTransacao** e a classe **Transacao** conforme abaixo: Preste atenção com os importes para as classes dos pacotes corretos.

```
public enum EnumTipoTransacao {

        SAQUE, DEPOSITO, TRANSFERENCIA;
}
```

```
import java.util.Calendar;
import java.util.Date;

public class Transacao {

        private Date data;
        private Conta contaDebito;
        private Conta contaCredito;
```

```
	private double valor;
	private String descricao;


	private EnumTipoTransacao tipoTransacao;

	public Transacao( Date data, Conta contaDebito, Conta contaCredito, Double valor, String descricao,
EnumTipoTransacao tipoTransacao ) {

		this.data = data;
		this.contaDebito = contaDebito;
		this.contaCredito = contaCredito;
		this.valor = valor;
		this.descricao = descricao;
		this.tipoTransacao = tipoTransacao;
	}

	//Crie os get e set.

	public String toString() {

		if (EnumTipoTransacao.TRANSFERENCIA == getTipoTransacao()) {

			return "Transacao data " + UtilData.DDMMAAAAHHMM(getData()) + ", conta debito "
+ getContaDebito().getNumero() + ", conta credito " + getContaCredito().getNumero() + ", valor " +
getValor() + ", descricao -> " + getDescricao();

		} else if (EnumTipoTransacao.DEPOSITO == getTipoTransacao()) {

			return "Deposito data " + UtilData.DDMMAAAAHHMM(getData()) + ", conta " + get-
ContaCredito().getNumero() + ", valor " + getValor() + ", descricao -> " + getDescricao();

		} else if (EnumTipoTransacao.SAQUE == getTipoTransacao()) {

			return "Saque data " + UtilData.DDMMAAAAHHMM(getData()) + ", conta " + getConta-
Credito().getNumero() + ", valor " + getValor() + ", descricao -> " + getDescricao();
		}

		return "Nenhum tipo de transação";
		}
}
```

2 - Modificaremos nossa classe **Conta** a fim de mantermos o histórico de transações, será criado uma variável do tipo **ArrayList** para guardar uma lista de transações ocorridas na conta.

```
import java.util.ArrayList;
import java.util.Calendar;

public class Conta {

	private int numero;

	private String titular;

	private Date dataAbertura;

	private double saldo;

	private ArrayList movimento;

	// construtor padrão da classe Conta que define a data de criação da conta e inicializa o array de tran-
sacao
	public Conta() {
		dataAbertura = UtilData.data();
		movimento = new ArrayList();
```

```
	}

	// construtor com dois parametros
	public Conta( String nome, int nconta ) {

		this();
		numero = nconta;
		titular = nome;
		saldo = 0.0; // Conta em reais e zerada
	}

	// INSIRA OS MÉTODOS GETTERS E SETTERS PARA O ATRIBUTO MOVIMENTO

}
```

3 - Modifique a classe **ContaService** inserindo os métodos para manter o histórico de transações conforme abaixo.

```
public class ContaService {

	public void depositar(Conta contaDestino, double valor) {
		contaDestino.setSaldo(contaDestino.getSaldo() + valor);

		this.historicoTransacao(null, contaDestino, valor, "deposito na conta " + contaDestino.getNu-
mero(), EnumTipoTransacao.DEPOSITO);
	}

	public void sacar(Conta contaSaque, double valor) {
		contaSaque.setSaldo(contaSaque.getSaldo() - valor);

		this.historicoTransacao(null, contaSaque, valor, "saque na conta " + contaSaque.getNumero(),
EnumTipoTransacao.DEPOSITO);
	}

	// método sobrecarregado, transfere dados desta conta (this) para outra
	public boolean transferir(Conta contaSaque, double valor, Conta contaDestino) {

		return transferir(contaSaque, valor, contaDestino, "transferencia para conta " + contaDesti-
no.getNumero());
	}

	// método sobrecarregado, transfere valor desta conta (this) para outra conta e registra a transação
	public boolean transferir(Conta contaSaque, double valor, Conta contaDestino, String descr) {

		if (contaSaque.getSaldo() - valor >= 0) {

			this.debito(contaSaque, valor);

			this.credito(contaDestino, valor);

			this.historicoTransacao(contaSaque, contaDestino, valor, descr, EnumTipoTransacao.-
TRANSFERENCIA);

			return true;

		} else {

			return false;
		}

	}

	// subtrai valor do saldo
	protected void debito(Conta contaOperacao, double valor) {
```

```
			contaOperacao.setSaldo(contaOperacao.getSaldo() - valor);

		}

		// adiciona valor ao saldo
		protected void credito(Conta contaOperacao, double valor) {
			contaOperacao.setSaldo(contaOperacao.getSaldo() + valor);

		}

		// cria um objeto transação e registra adicionando no movimento da conta
		protected void historicoTransacao(Conta contaDebito, Conta contaCredito, double valor, String descr, EnumTipoTransacao tipoTransacao) {

			Transacao transacao = new Transacao(UtilData.data(), contaDebito, contaCredito, valor, descr, tipoTransacao);

			if (contaDebito != null) {
				contaDebito.getMovimento().add(transacao);

			}

			contaCredito.getMovimento().add(transacao);
		}

	}
```

4 - Observe como fazemos uso da referência **this** no método **transferir(),** neste caso queremos evidenciar o uso dos métodos pelo próprio objeto. Como são objetos da mesma classe dizemos que há um auto relacionamento. Perceba que todas as operações que podem ser realizadas por **ContaService** agora estão sendo direcionadas internamente pelos métodos invocados para o **método transferir** que registra o histórico de **Transação** no atributo **movimento**.

5 - Crie a classe **MovimentoContaCaixa.java** como definida abaixo para testarmos se o histórico de transações esta sendo gravado corretamente.

```
public class MovimentoContaCaixa {

	public static void main(String[] args) {

		// Cria uma instância de ContaService onde está presente as operações para Objeto Conta
		ContaService operacoesConta = new ContaService();

		// cria conta caixa
		Conta caixa = new Conta("ContaCaixa", 0);
		caixa.setSaldo(100000);

		Conta correntista1 = new Conta("Hinfe Liz", 1001);

		// faz deposito
		operacoesConta.depositar(correntista1, 1000);

		Conta correntista2 = new Conta("ZILEF D'AVIDA", 1002);

		// faz deposito, transferir para conta caixa
		operacoesConta.depositar(correntista2, 2000);

		// Mostra saldo correntista 1
		System.out.println("correntista1 saldo =" + correntista1.getSaldo());
		// Mostra saldo correntista 2
		System.out.println("correntista2 saldo =" + correntista2.getSaldo());
```

```
                if (operacoesConta.transferir(correntista1, 100.00, correntista2)) {
                        System.out.println("transferencia ok");
                } else {
                        System.out.println("nao pode transferir !");
                }

                // Mostra saldo correntista 1
                System.out.println("correntista1 saldo =" + correntista1.getSaldo());

                // Mostra saldo correntista 2
                System.out.println("correntista2 saldo =" + correntista2.getSaldo());

                // faz saque
                operacoesConta.sacar(correntista2, 120.00);
                System.out.println("saque ok");

                // Mostra saldo correntista 2
                System.out.println("correntista2 saldo =" + correntista2.getSaldo());

                // mostra movimento correntista 1
                System.out.println(correntista1.getMovimento());
                // mostra movimento correntista 2
                System.out.println(correntista2.getMovimento());
        }
}
```