

Lab 6 – Criando suas próprias classes

Neste laboratório você irá aprender a criar e usar suas próprias classes. A partir deste laboratório os exercícios dos próximos laboratórios possuem dependências, então deve ser feito todos os exercícios, não pule nenhum exercício porque irá precisar das classes que serão criadas e das alterações dos métodos solicitadas. 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 ou empresa que trabalha com dinheiro e várias contas para armazenar os saldos. 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 de a ferramenta sugerir o que deve ser feito para resolver alguns erros de compilação, se você não sabe ou não entende 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.

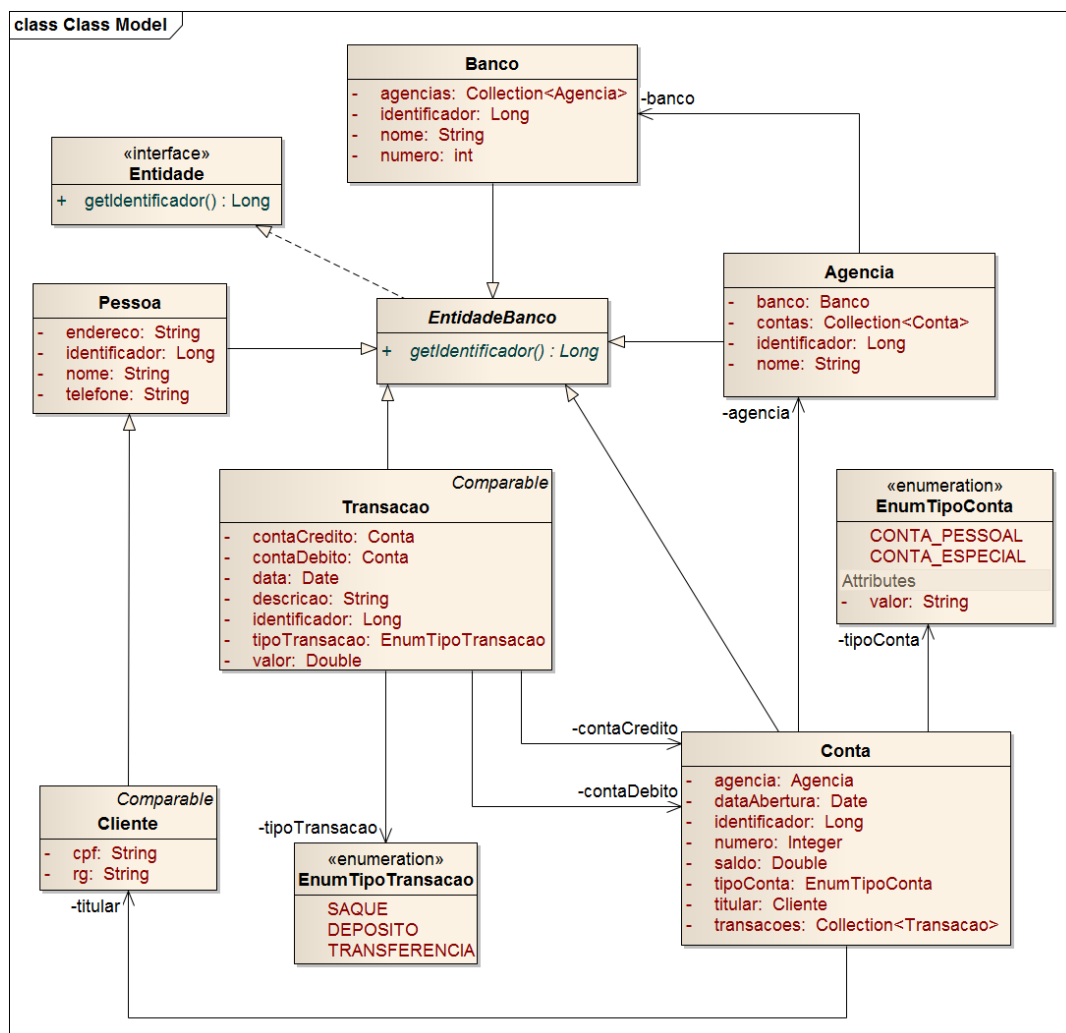


Figura 5.1 – Diagrama UML para aplicação Banco

Sugerimos que estes exemplos sejam feitos com uso da IDE Eclipse.

Duração prevista: 80 minutos

Exercícios

Exercício 1: Definindo e usando classes (20 minutos)

Exercício 2: Membros estáticos (10 minutos)

Exercício 3: Sobrecarga (20 minutos)

Exercício 4: Construtores (10 minutos)

Exercício 5: referencia “this” (20 minutos)

Exercício 1: Definindo e usando classes

1. Criando uma classe **Conta.java**, que representa uma conta de banco.

1.1 – Definindo a nova classe Conta.java

1. Crie um novo projeto com o nome **Banco** e crie sua classe para que fique conforme a especificação UML abaixo.

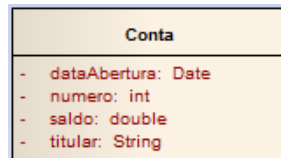


Figura 5.1 – Diagrama UML classe Conta

```
import java.util.Date;

public class Conta {

    private int numero;

    private String titular;

    private double saldo;

    private Date dataAbertura;

}
```

Listagem 5.1 – Conta.java

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

```
public int getNumero() {

    return numero;

}

public void setNumero(int numero) {

    this.numero = numero;

}
```

3. Crie a classe **ContaService.java** com os métodos **depositar()**, **sacar()** e **transferir()** conforme o código na **Listagem-5.2**.

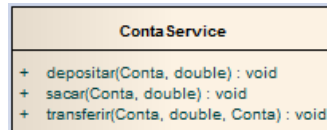


Figura 5.2 – Diagrama UML classe ContaService

```
public class ContaService {

    public void depositar(Conta contaDestino, double valor) {

        // credita na conta e debita no caixa
        contaDestino.setSaldo(contaDestino.getSaldo() + valor);
    }

    public void sacar(Conta contaSaque, double valor) {

        // debita na conta e credita no caixa
        contaSaque.setSaldo(contaSaque.getSaldo() - valor);
    }

    public void transferir(Conta contaSaque, double valor, Conta contaDestino) {

        // transfere valor da conta para conta destino
        this.sacar(contaSaque, valor);
        this.depositar(contaDestino, valor);
    }

}
```

Listagem 5.2 – Classe ContaService.java

4. Agora crie uma nova classe **TestaConta.java** para fazer uso da classe **Conta.java** e testarmos os seus métodos conforme abaixo:

```
import java.util.Scanner;

public class TestaConta {

    public static void main(String[] argv) {

        // objeto para ler dados da console, captura o que for digitado
        Scanner c = new Scanner(System.in);

        // declara e inicializa saldo com valor digitado pelo usuário
        System.out.println("Digite o saldo inicial da conta");
        double saldoConta = c.nextDouble();

        // declara e inicializa o numero da conta digitado pelo usuário
        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 valor dos atributos da instância da classe Conta
```

```
conta1.setNumero(numeroConta);
conta1.setSaldo(saldoConta);

// Nova instância de Conta para transferência
Conta conta2 = new Conta();

// Nova instância de Conta para transferência
Conta conta3 = new Conta();

// Mostra dados da instância da classe Conta
System.out.println("O numero da Conta1 :" + conta1.getNumero());
System.out.println("O saldo da Conta1 :" + conta1.getSaldo());

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

// chamada ao 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 Conta1 :" + conta1.getSaldo());

System.out.println("Saldo da Conta 1 :" + conta1.getSaldo());
System.out.println("Saldo da Conta 2 :" + conta2.getSaldo());
System.out.println("Saldo da Conta 3 :" + conta3.getSaldo());

// chamada ao método transferir para retirar o valor de conta1 para conta2
System.out.println("Transferir 50.00 de conta 1 para conta2 ");
operacoesConta.transferir(conta1, 50.00, conta2);

System.out.println("Saldo da Conta 1 :" + conta1.getSaldo());
System.out.println("Saldo da Conta 2 :" + conta2.getSaldo());
System.out.println("Saldo da Conta 3 :" + conta3.getSaldo());

}

}
```

Listagem 5.3 – Classe TestaConta.java

4. Modifique sua classe para criar uma terceira instância da classe **Conta.java** em sua classe **TestaConta.java** como o nome da variável **conta3** e transfira 100,23 da variável **conta2** para **conta3**. Mostre o saldo de cada conta antes e depois da transferência.

5. Configure o campo **dataAbertura** de **Conta.java** para armazenar a data de criação da instância, ou seja, quando for instanciado o objeto **Conta.java** deve ser atribuído a data do sistema operacional para a variável e mostre o valor do atributo na tela após a ser instanciado o objeto para cada variável na classe **TestaConta.java**.

Dica: Deve ser alterado o construtor da classe para possuir este comportamento evitando que seja criada uma instância e não seja atribuído um valor para o atributo **dataAbertura**.

Exercício 2: Membros estáticos

1. Usando variáveis estáticas
2. Usando métodos estáticos

2.1 Criando aplicativos que usam variáveis estáticas

1. Crie uma classe de **UtilData.java**, 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. Crie a classe para que fique conforme a **Listagem-**

5.4, UtilData.java.

```
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 = " + agora(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
    // Formata uma data no formato dd/mm/aaaa hh:mm
    String DDMMAAAHHMM(Date data) {
```

```
        return new java.text.SimpleDateFormat("dd/MM/yyyy HH:mm").format(data);
    }
}
```

Listagem 5.4 – Classe Data.java, membros estáticos

2. Agora crie outra classe **ExemploVariavelEstatica.java** conforme **Listagem-5.5**. Observe o uso das variáveis estáticas, veja os comentários.

```
public class ExemploVariavelEstatica {

    public static void main(String[] args) {

        // Acessando variáveis estáticas 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);

        // Acesso a variável de instância data 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);

        // A variável estática pode ser acessada por variável de instância de objeto
        System.out.println("Dia da Semana " + data.DiaDaSemana);
        data.DiaDaSemana = 3;
        System.out.println("Mudou Dia da Semana " + data.DiaDaSemana);

        // Variáveis estáticas podem ser acessada de múltiplas instâncias
        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);
    }

}
```

Listagem 5.5 ExemploVariávelEstática.java

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

4. Modifique a **Listagem-5.5** para imprimir os valores dos novos membros estáticos que você modificou no exercício anterior.

2.2. Criando aplicativos que usam métodos estáticos

1. Observe a **Listagem-5.6**, 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 estatico da classe UtilData, 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.DDMMAAAHHMM(data));  
  
        // Metodos de instancia nao podem ser invocados diretamente ocorre erro de compilacao  
        UtilData.DDMMAAAHHMM(data);  
    }  
}
```

Listagem 5.6 ExemploDeMetodosEstaticos.java

2. Modifique a listagem anterior de modo a corrigir o erro de compilação e poder executar o programa. Você terá que alterar o modificador de acesso de alguns métodos da classe **UtilData.java** e alterar a referência a chamada de um método na classe **ExemploDeMetodoEstaticos.java**.

Exercício 3: Sobrecarga

1. Métodos sobrecarregados

3.1 Adicionando métodos sobrecarregados

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

```
public class ContaService {  
  
    public void depositar(Conta contaDestino, double valor) {  
  
        // credita na conta e debita no caixa  
        contaDestino.setSaldo(contaDestino.getSaldo() + valor);  
    }  
  
    public void sacar(Conta contaSaque, double valor) {  
  
        // debita na conta e credita no caixa  
        contaSaque.setSaldo(contaSaque.getSaldo() - valor);  
    }  
  
    // Para não implementar a mesma regra duplicado este método chama o segundo transferir  
    // e informa o limite com valor zero para representando 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 com argumentos diferentes, 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);
}

}
```

Listagem 5.7 – Métodos sobrecarregado

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

3. Para testar a execução dos dois métodos **transferir** da classe **ContaService.java** que foram criados crie a classe **SobrecargaTransferir.java** conforme 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 instância de Conta para transferencia
        Conta conta2 = new Conta();
        conta2.setSaldo(50.00);

        // cria nova instância de Conta para transferencia
        Conta conta3 = new Conta();

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

        // transferindo 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());
        System.out.println("Saldo da Conta 3:" + conta3.getSaldo());

        // transferindo 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());
        System.out.println("Saldo da Conta 3:" + conta3.getSaldo());

    }

}
```


Listagem 5.8 – Usando métodos sobrecarregados

4. Crie uma terceira instância da classe **Conta.java** com nome da variável **conta3** na **Listagem-5.8** e transfira 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.java** 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 nos 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 DDMMAAAHHMM(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);
}
}
```

Listagem 5.9 – UtilData.java com sobrecarga de construtores

Exercício 4: Construtores

1. Usando múltiplos construtores
2. Encadeando construtores com o método **this()**

4.1 Definindo múltiplos construtores numa classe

1. Crie um construtor na classe **Conta.java** que receba como argumento o nome do titular e o número da conta e armazene esses valores nas suas respectivas variáveis e armazene o valor 0 (zero) no atributo saldo conforme abaixo:

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

    numero = nconta;
    titular = nome;
    saldo = 0.0; // Conta em reais e zerada
}
```

2. Tínhamos definido que todo objeto da classe **Conta.java** ao ser criado deve ser atribuído a data de abertura conforme construtor padrão criado. Para que não seja necessário reescrevermos esta regra de negócios vamos fazer com que ao ser chamado o construtor com nome e numero de conta, este construtor automaticamente invoque o construtor que atribui a data de criação com **this()** conforme abaixo.

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

3. Crie a classe **TestaConstrutor.java** para criar duas variáveis de **Conta.java**, cada uma utilizando um construtor que foi criado e imprima a data de abertura.

Exercício 5 : Referência “this”

1. Invocando métodos com “this”
2. “this” como parâmetro

5.1 Invoque métodos com “this”

1. Nossa classe **ContaService.java** 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.java** e a classe **Transacao.java** como abaixo **Listagem-5.10**: Arrume os importes para as classes dos pacotes corretos.

```
public enum EnumTipoTransacao {

    SAQUE, DEPOSITO, TRANSFERENCIA;
}
```

```
import java.util.Calendar;

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;
    }

    //get e set

    public String toString() {

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

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

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

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

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

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

        return "Nenhum tipo de transação";
    }

}
```

Listagem 5.10 – Classe Transação.java

2. Modificaremos nossa classe **Conta.java** afim 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 transacao
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

}
```

Listagem 5.11 – Classe Conta

3.Modifique a classe **ContaService.java** inserindo os métodos para manter o histórico de transações conforme abaixo **Listagem-5.12**.

```
public class ContaService {

    public void depositar(Conta contaDestino, double valor) {

        // credita na conta e debita no caixa
        contaDestino.setSaldo(contaDestino.getSaldo() + valor);

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

    public void sacar(Conta contaSaque, double valor) {

        // debita na conta e credita no caixa
        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 " +
        contaDestino.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);
}
}
```

} Listagem 5.12 – Classe Conta

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 operações que podem ser realizadas por **ContaService.java** 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("Hinfel 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());
}
}
```

Listagem 5.13 – MovimentoContaCaixa.java