Universidade Federal de São Carlos

Programa de Pós Graduação em Ciência da Computação - PPGCC

Paradigmas de Linguagem de Programação

Relatório do Trabalho
Desenvolvido na Linguagem Java

Alunos:

Bruno César Sales Alves

Patrícia Deud Guimarães

Docente Responsável: Heloísa Arruda Camargo

São Carlos - SP

05/07/2019

1. Classe Conta

Figura 1 - Código da classe Conta.

```
public class Conta
{
    private String nome = new String();
    private double saldo;

   public Conta(String nome, double saldo)
   {
        this.nome = nome;
        this.saldo = saldo;
   }

   public double getSaldo(){
       return this.saldo;
   }

   public String getNome(){
       return this.nome;
    }

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

Fonte: Repositório no GitHub.

A primeira classe desenvolvida neste trabalho foi a classe Conta. Ela foi desenvolvida para que fosse possível criar um *array* de contas, como será mostrado na próxima seção. Ela possui dois atributos: o **nome** do titular da conta, uma *String*, e seu **saldo** bancário, um *double*. Possui um método construtor e os *getters* e *setters*, que serão chamados por outras classes.

2. Classe Banco

Figura 2 - Código da classe Banco.

```
import java.util.ArrayList;

public class Banco
{
    private double val_inicio =10000;
    private ArrayList<Conta> contas = new ArrayList<Conta>();

    Banco(){
        contas.add(new Conta("Bruno",val_inicio));
        contas.add(new Conta("Kalyl",val_inicio));
        contas.add(new Conta("Marcus",val_inicio));
        contas.add(new Conta("Luciana",val_inicio));
        contas.add(new Conta("Patricia",val_inicio));
    }

    public synchronized void transferencia(Conta de,Conta para,double valor)
    {
        while(de.getSaldo() < valor){
            try{
                wait();
            }catch (InterruptedException e){

            }
        }

        de.setSaldo(de.getSaldo() - valor);
        para.setSaldo(para.getSaldo()+ valor);

        notify();
    }

    public Conta getConta(int index){
        return (Conta) getContas().get(index);
    }

    public ArrayList getContas(){
        return contas;
    }

}
```

Fonte: Repositório no GitHub.

A classe Banco é responsável por definir e inicializar cada conta do banco. Um *array* de contas do tipo Conta é inicializado e define-se a variável *double* **val_inicio**, que assume o valor 10000.

No construtor da classe, são adicionadas cinco contas ao *array* de contas, cada uma contendo uma *string* com o nome do responsável pela conta e o valor inicial mencionado anteriormente.

A classe possui um método chamado **transferencia**, responsável por transferir valores entre as contas bancárias. Tem como parâmetros duas contas do tipo Conta – uma que transfere um valor e outra que o recebe – e o valor a ser enviado, que é do tipo *double*. A declaração *synchronized* no método garante que, ao chamá-lo, as *threads* que serão executadas com exclusão mútua. Isto é, verifica-se se alguma outra *thread* já está de posse do bloco de operações daquele objeto antes de permitir o acesso. Caso isso aconteça, a segunda *thread* deve esperar a primeira sair da região crítica para que possa executar sua operação.

Enquanto o valor enviado pela conta de origem for menos que seu saldo, o processo aguarda a ocorrência de outras transferências até ter o saldo maior ou igual ao valor. Isso ocorre por meio da chamada de **wait()**, método da classe Object. Ele bloqueia a *thread* em questão, colocando-a em espera, e libera o bloco de operações para que outra *thread* do conjunto de entrada seja selecionada para executar. Caso o saldo da conta a realizar a transferência seja maior que o valor da mesma, retira-se este valor do saldo e acrescenta-o à conta de destino da transferência. Por fim, chama-se o método **notify()**, também da classe Object, responsável por avisar a uma *thread* que está em espera que o bloco de operações está liberado.

O método **getConta** (int **index**) retorna a conta do *index* passado por parâmetro após buscá-la no *array* de contas. O método **getContas()** retorna um *array* de contas. Esses métodos são utilizados na classe TransferThread.

3. Classe TransferThread

Figura 3 - Código da classe TransferThread.

```
import java.lang.Math;

class TransferThread extends Thread
{
  private Banco banco;
  private Conta contaorigem;
  private int de;

  public TransferThread (Banco b, int de)
  {
    banco = b;
    contaorigem = (Conta) b.getContas().get(de);
  }

  public void run ()
  {
    try
    {
      for (int i=0; i <10; i++)
      {

        do{
            de = (int)((Math.random()*100)%5);
        } while(de < 0 | de > 4 | banco.getConta(de).getNome().equals(contaorigem.getNome()));
        //validação para verificar se a conta de destino não é a mesma que a conta origem
        double valor = ((Math.random()*100));

        Conta contadestino = (Conta) banco.getContas().get(de);
        System.out.println(contaorigem.getNome() + " transferiu R$"+ String.format("%.2f", valor)
        +" para a conta de " + contadestino.getNome());
          banco.transferencia(contaorigem, contadestino, valor);
        // define o valor a ser transferido aleatoriamente
        // faz a operação de transferência
        sleep(1000);
      }
    } catch (InterruptedException e) {}
  }
}
```

Fonte: Repositório no GitHub.

A classe TransferThread estende da superclasse Thread. Ela possui dois atributos: **banco** do tipo Banco e **contaorigem** do tipo Conta. Possui um construtor com parâmetros **banco** e **de**, que servirão para criar uma TransferThread recuperando uma conta **de** buscando por seu *index* em um a*rray* de contas de um Banco **b** qualquer.

Possui um método **run()**, responsável por deixar a *thread* pronta para ser executada. Neste método, utiliza-se a biblioteca matemática *Math* para gerar valores randômicos tanto para definir os valores a serem transferidos entre as contas (variável **valor**), mas também o *index* da conta de origem, que assume um valor entre 0 e 4

(variável **de**). Além disso, é realizada uma validação para verificar se as contas de origem e destino são iguais, para que não haja transferências para a mesma pessoa.

Define-se a conta de destino e, depois, mostra-se ao usuário uma mensagem com o valor da transferência, o nome do titular da conta de origem e de destino da transferência. A chamada do método **banco.transferencia** (**contaorigem**, **contadestino**, **valor**) é responsável por definir e executar a transferência desses valores aleatórios.

Finalmente, o método de ***sleep***(1000) é chamado e realiza uma requisição para bloquear uma *thread* por 1000 milissegundos. Depois disso, a *thread* é colocada na fila de tarefas prontas.

4. Classe Main

Figura 4 - Código da classe Main.

```
class Main {
    public static void main(String args[]){
        Banco banco = new Banco();
        TransferThread transfer1 = new TransferThread(banco, 0);
        TransferThread transfer2 = new TransferThread(banco, 1);
        TransferThread transfer3 = new TransferThread(banco, 2);
        TransferThread transfer4 = new TransferThread(banco, 3);
        TransferThread transfer5 = new TransferThread(banco, 4);

        transfer1.start();
        transfer2.start();
        transfer3.start();
        transfer4.start();
        transfer5.start();

        try{
            transfer1.join();
            transfer2.join();
            transfer3.join();
            transfer4.join();
            transfer5.join();
        } catch (InterruptedException e){
        }


    }


}
```

Fonte: Repositório no GitHub.

A classe *Main* é responsável por instanciar e executar todas as *threads* da aplicação. Nela, um banco é criado e inicializado e cinco *threads* do tipo *TransferThreads* são declaradas e construídas com os argumentos **banco** e **de**. O **de** corresponde ao *index* atribuído a cada conta criada.

Todas as threads são iniciadas através do método **start()**, da superclasse *Thread*, responsável por iniciar o objeto com uma unidade concorrente. Para isso, chama o método **run()**.

Todas as *threads* chamam seus métodos **join()**. Ele permite que um encadeamento aguarde até que outro encadeamento conclua sua execução. Ou

seja, se **transfer1** é um objeto cuja *thread* está atualmente em execução, então transfer1.join() irá certificar-se de a *thread* transfer1 seja terminada antes que a próxima instrução seja executada pelo programa.

5. Exemplos de execução

Figura 5 - Primeiro exemplo de compilação e execução do programa.

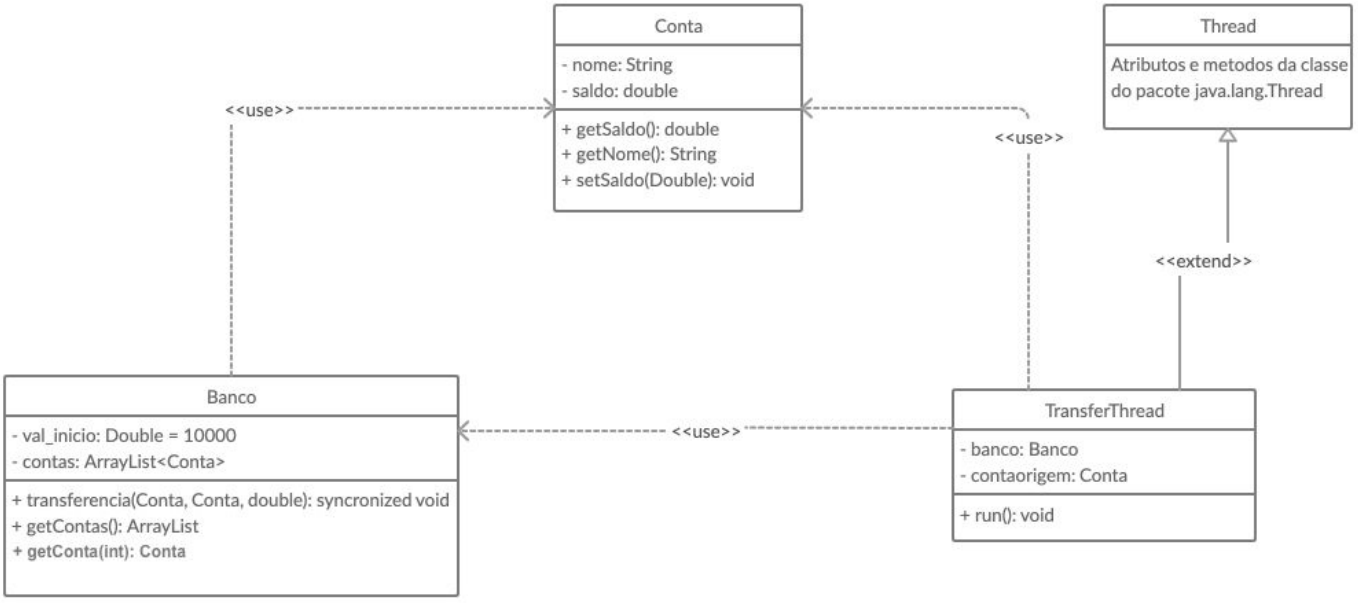
Fonte: Terminal.

Figura 6 - Segundo exemplo de compilação e execução do programa.

Fonte: Terminal.

6. Diagrama de classes

Figura 7 - Diagrama de classes.



Fonte: Desenvolvido pelos autores.

O diagrama acima resume o relacionamento entre as classes criadas neste trabalho, bem como os atributos e métodos que estas possuem.