INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA BAHIA Campus Irecê	**Instituto Federal de Educação, Ciência e Tecnologia da Bahia – *Campus* Irecê** **Disciplina: Linguagem Técnica II** **Prof° Jonatas Bastos**

Nome: _____

LISTA DE EXERCÍCIO 1 – Collections Framework

1. Criem um projeto que contenha uma calsse Conta de maneira que ela possua no mínimo um atributo *numero;*

   private int numero;

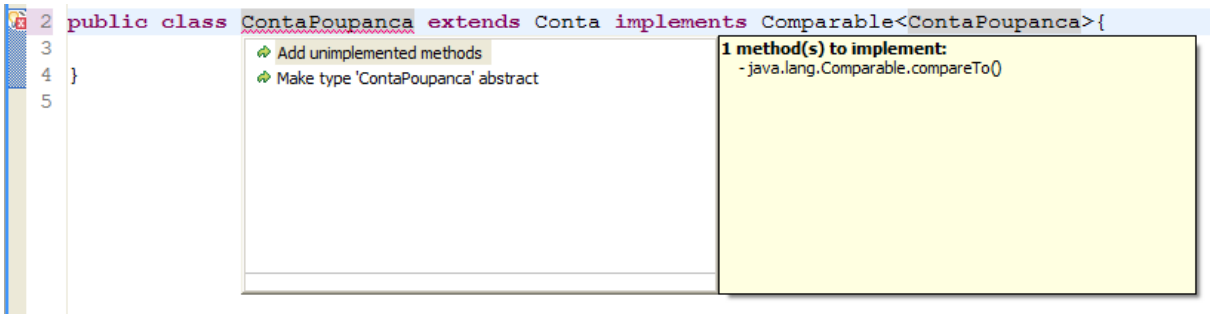
Além da classe **Conta** devem ser criadas duas classes **ContaCorrente** e **ContaPoupança** com os respectivos atributos que acharem necessário (obs: criar os metodos get e set das classes), que herdam da classe conta.

2. Faça sua classe ContaPoupanca implementar a interface Comparable<ContaPoupanca>. Utilize o critério de ordenar pelo número da conta ou pelo seu saldo

```
public class ContaPoupanca extends Conta implements Comparable<ContaPoupanca>{

    ...
}
```

Repare que o Eclipse prontamente lhe oferecerá um quickfix, oferecendo a criação do esqueleto dos métodos definidos na interface `Comparable`:



Deixe o seu método compareTo parecido com este:

```
public class ContaPoupanca extends Conta
```

```
            implements Comparable<ContaPoupanca> {

    // ... todo o codigo anterior fica aqui

    public int compareTo(ContaPoupanca o) {
        if (this.getNumero() < o.getNumero()) {
            return -1;
        }
        if (this.getNumero() > o.getNumero()) {
            return 1;
        }
        return 0;
    }
}
```

3. Crie uma classe TestaOrdenacao, onde você vai instanciar diversas contas e adicioná-las a uma List<ContaPoupanca>. Use o Collections.sort() nessa lista:

```
public class TestaOrdenacao {


    public static void main(String[] args) {

        List<ContaPoupanca> contas = new ArrayList<ContaPoupanca>();

        ContaPoupanca c1 = new ContaPoupanca();
        c1.setNumero(1973);
        contas.add(c1);

        ContaPoupanca c2 = new ContaPoupanca();
        c2.setNumero(1462);
        contas.add(c2);

        ContaPoupanca c3 = new ContaPoupanca();
        c3.setNumero(1854);
        contas.add(c3);

        Collections.sort(contas);
    }
}
```

Faça um laço para imprimir todos os números das contas na lista já ordenada:

```
for (int i = 0; i < contas.size(); i++) {

    Conta atual = contas.get(i);
```

```
    System.out.println("numero: " + atual.getNumero());
}
```

Atenção especial: repare que escrevemos um método compareTo em nossa classe e nosso código **nunca** o invoca!! Isto é muito comum. Reescrevemos (ou implementamos) um método e quem o invocará será um outro conjunto de classes (nesse caso, quem está chamando o compareTo é o Collections.sort, que o usa como base para o algoritmo de ordenação). Isso cria um sistema extremamente coeso e, ao mesmo tempo, com baixo acoplamento: a classe Collections nunca imaginou que ordenaria objetos do tipo ContaPoupanca, mas já que eles são Comparable, o seu método sort está satisfeito.

4. O que teria acontecido se a classe ContaPoupanca não implementasse Comparable<ContaPoupanca> mas tivesse o método compareTo? Faça um teste: remova temporariamente a sentença implements Comparable<ContaPoupanca>, não remova o método compareTo e veja o que acontece. Basta ter o método, sem assinar a interface?

5. Utilize uma LinkedList em vez de ArrayList:List<ContaPoupanca> contas = new LinkedList<ContaPoupanca>(); Precisamos alterar mais algum código para que essa substituição funcione? Rode o programa. Alguma diferença?

6. Como posso inverter a ordem de uma lista? Como posso embaralhar todos os elementos de uma lista? Como posso rotacionar os elementos de uma lista? Investigue a documentação da classe Collections dentro do pacote java.util.

7. Mude o critério de comparação da sua ContaPoupanca. Adicione um atributo nomeDoCliente na sua classe (caso ainda não exista algo semelhante) e tente mudar o compareTo para que uma lista de ContaPoupanca seja ordenada alfabeticamente pelo atributo nomeDoCliente.

8. Crie um código que insira 30 mil números numa ArrayList e pesquise-os. Vamos usar um método de System para cronometrar o tempo gasto:

```
public class TestaPerformance {


    public static void main(String[] args) {
        System.out.println("Iniciando...");
        long inicio = System.currentTimeMillis();
        Collection<Integer> teste = new ArrayList<Integer>();

        int total = 30000;

        for (int i = 0; i < total; i++) {
            teste.add(i);
        }

        for (int i = 0; i < total; i++) {
            teste.contains(i);
        }

        long fim = System.currentTimeMillis();
        long tempo = fim - inicio;
        System.out.println("Tempo gasto: " + tempo);
    }
}
```

Troque a ArrayList por um HashSet e verifique o tempo que vai demorar:

Collection<Integer> teste = new HashSet<Integer>();

O que é lento? A inserção de 30 mil elementos ou as 30 mil buscas? Descubra computando o tempo gasto em cada for separadamente.

9. Repare que, se você declarar a coleção e der new assim:

```
Collection<Integer> teste = new ArrayList<Integer>();
```

em vez de

```
ArrayList<Integer> teste = new ArrayList<Integer>();
```

É garantido que vai ter de alterar só essa linha para substituir a implementação por HashSet. Estamos aqui usando o polimorfismo para nos proteger que mudanças de implementação venham nos obrigar a alterar muito código. Mais uma vez: *programe voltado a interface, e não à implementação!* Esse é um **excelente** exemplo de bom uso de interfaces, afinal, de que importa como a coleção funciona? O que queremos é uma coleção qualquer, isso é suficiente para os nossos propósitos! Nosso código está com **baixo acoplamento** em relação a estrutura de dados utilizada: podemos trocá-la facilmente.

Esse é um código extremamente elegante e flexível. Com o tempo você vai reparar que as pessoas tentam programar sempre se referindo a essas interfaces menos específicas, na medida do possível: os métodos costumam receber e devolver Collections, Lists e Sets em vez de referenciar diretamente uma implementação.

10. Faça testes com o Map, como visto nesse capítulo:

```
public class TestaMapa {


    public static void main(String[] args) {
        Conta c1 = new ContaCorrente();
        c1.deposita(10000);

        Conta c2 = new ContaCorrente();
        c2.deposita(3000);

        // cria o mapa
        Map mapaDeContas = new HashMap();

        // adiciona duas chaves e seus valores
        mapaDeContas.put("diretor", c1);
        mapaDeContas.put("gerente", c2);

        // qual a conta do diretor?
        Conta contaDoDiretor = (Conta) mapaDeContas.get("diretor");
        System.out.println(contaDoDiretor.getSaldo());
    }
}
```

Depois, altere o código para usar o generics, e não haver a necessidade do casting, além da garantia de que nosso mapa estará seguro em relação a tipagem usada.

11. Crie uma classe Banco que possui uma List de Conta chamada contas. Repare que numa lista de Conta, você pode colocar tanto ContaCorrente quanto contaPoupanca por causa do polimorfismo. Crie um método void adiciona(Conta c), um método Conta pega(int x) e outro int pegaTotalDeContas. Basta usar a sua lista e delegar essas chamadas para os métodos e coleções que estudamos.

12. No Banco, crie um método Conta buscaPorNome(String nome) que procura por uma Conta cujo nome seja equals ao nome dado. Você pode implementar esse método com um for na sua lista de Conta, porém não tem uma performance eficiente. Adicionando um atributo privado do tipo Map<String, Conta> terá um impacto significativo. Toda vez que o método adiciona(Conta c) for invocado, você deve invocar .put(c.getNome(), c) no seu mapa. Dessa maneira, quando alguém invocar o método Conta buscaPorNome(String nome), basta você fazer o get no seu mapa, passando nome como argumento!

13. Crie o método hashCode para a sua conta, de forma que ele respeite o equals de que duas contas são equals quando tem o mesmo número. Verifique se sua classe funciona corretamente num HashSet. Remova o método hashCode. Continua funcionando? Dominar o uso e o funcionamento do hashCode é fundamental para o bom programador.

14. Gere todos os números entre 1 e 1000 e ordene em ordem decrescente utilizando um TreeSet.

15. Gere todos os números entre 1 e 1000 e ordene em ordem decrescente utilizando um ArrayList.