Lab 11 – Coleções

Neste laboratório você exercitará o uso de **Collections** desenvolvendo aplicações.

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

Duração prevista: 190 minutos

Exercícios

Exercício 1: UML e Coleções (60 minutos)

Exercício 2: Usando a coleção Set (30 minutos)

Exercício 3: Usando a coleção List (30 minutos)

Exercício 4: Usando a coleção Map (20 minutos)

Exercício 5: Ordenação, Busca e Manipulação de Dados (20 minutos)

Exercício 1 – UML e Coleções

1. A **Figura-10.1** é a representação parcial em UML do digrama de classes da aplicação Banco que você está escrevendo ao realizar os laboratórios. Crie e/ou modifique as classes conforme mostrado às listagens abaixo.

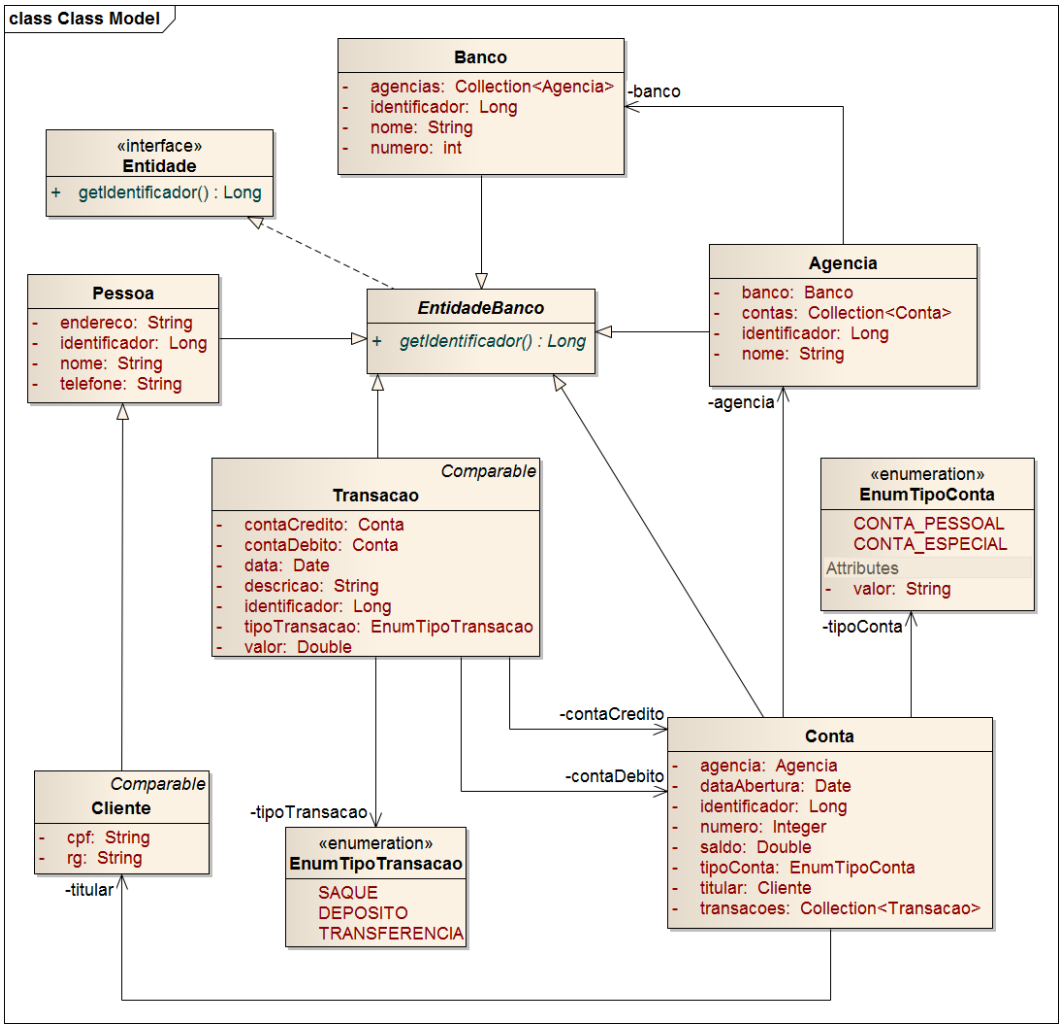

Figura 10.1 – Diagrama UML para aplicação Banco

O diagrama te apresenta as relações entre as classes, essas relações são relações do tipo **has-one**. Logo, o diagrama nos informa que a classe **Banco.java** possui nenhuma ou várias referências de objetos da classe **Agencia.java**, que **Agencia.java** tem zero ou mais referências de objetos da classe **Conta.java**, que a classe **Conta.java** tem zero ou mais referências de objetos da classe **Cliente.java**. Essas relações são denominadas de **agregação em UML**.

Abaixo é mostrado um possível código para esta representação gráfica das relações entre as classes. Observe que a cardinalidade, relação numérica entre instâncias de objetos das classes, está sendo implementada para variáveis do tipo **java.util.Collection**. Isto satisfaz o conceito da cardinalidade representada no **Diagrama UML.** Pense o que seria necessário fazer se você tivesse que usar um **Array**! A classe **Collection** é uma das classes que faz parte do **Framework Java Collections**, essas classes implementam algoritmos eficientes de busca, inserção e remoção de objetos dentro de estruturas que implementam conceitos de **Lista, Fila, Mapas e Conjuntos**, facilitando o trabalho do programador, que só tem a responsabilidade de escolher a implementação mais adequada para seu programa.

```
import java.util.Collection;

public class Banco extends EntidadeBanco {

	private Long identificador;

	private int numero;

	private String nome;
	private Collection<Agencia> agencias;

	//get e set

}
```

Listagem 10.1 – Codificação simplificada da classe Banco

```
import java.util.ArrayList;
import java.util.Collection;

public class Agencia extends EntidadeBanco {

	private Long identificador;

	private String nome;

	private Banco banco;

	private Collection<Conta> contas;

	public Agencia( String nome ) {

		this.nome = nome;

		contas = new ArrayList<Conta>();
	}

}
```

Listagem 10.2 – Codificação simplificada da classe Agencia

```
public class Pessoa extends EntidadeBanco {

	private Long identificador;

	private String nome;

	private String telefone;

	private String endereco;

}
public class Cliente extends Pessoa {

	private String cpf;

	private String rg;

}
```

Listagem 10.3 – Codificação simplificada da classe Cliente

```
public class Conta extends EntidadeBanco {

	private Long identificador;

	private int numero;

	private double saldo;

	private Date dataAbertura;

	private Collection<Transacao> transacoes;

	private Cliente titular;

	private Agencia agencia;

}
```

Listagem 10.4 – Codificação simplificada da classe Conta

2. Modifique a classe **Conta.java**, para que o campo titular seja do tipo da classe **Cliente.java**. Modifique também os construtores e métodos da classe que fazem referência ao campo titular como **String** alterando para que seja do tipo **Cliente.java**.

Isto irá gerar erros em outras classes que estava sendo passado o titular como **String** e agora deve ser um objeto do tipo **Cliente.java**, mas não se preocupe por enquanto, logo iremos definir uma solução pratica para resolver esses erros de compilação.

3. Modifique a classe **Cliente.java** adicionando um construtor com parâmetro String para inicializar o atributo nome do tipo String.

Agora para resolver os erros de compilação quando utilizado o construtor de **Conta("titular", 1)** substitua por **Conta(new Cliente("titular"), 1)**, com esta alteração você obedece a regra do construtor passando um cliente como parâmetro e ao mesmo tempo criando o objeto **Cliente** passando para ele o nome que é obrigatório para o construtor da classe **Cliente.java**.

Exercício 2 – Usando coleção Set

A **Interface Set** herda da **Interface Collection** e, por definição, proíbe elementos duplicados dentro da coleção. Todos os métodos da super-classe estão presentes e nem um novo método foi introduzido. As classes concretas que implementam **Set** dependem do método **equals()** do objeto adicionado para checar a igualdade dos objetos adicionados.

O **Framework Collection Java** fornece duas implementações de propósito geral para interface **Set: HasSet e TreeSet.** Com freqüência você irá usar um **HashSet** para manter coleções sem elementos duplicados. Por razões de eficiência, objetos adicionados numa **HashSet** necessitam da implementação do método **hashCode()** de forma que o ***código de hash*** seja distribuído adequadamente. Para todo sistema em que você criar suas próprias classes e desejar armazená-los em um **HashSet**, lembre-se que você deverá sobrescrever o método **hashCode() de Object**.

A implementação **TreeSet** é útil quando você precisa extrair elementos de uma coleção de uma maneira **ordenada**. Para que o **TreeSet** trabalhe corretamente os objetos adicionados devem ser classificáveis. O **Framework Collection** adiciona suporte para elementos que implementam a **Interface Comparable**, que veremos mais adiante.

2.1 – Descartando elementos duplicados com HashSet

1. Sobreponha **(override)** o método **toString()**, retornando o campo nome, na classe **Cliente.java**.

2. Lembrando que dois objetos são iguais se **ob1.equals(ob2) = true** então **ob1.hashCode() = ob2.hashCode()**, assim os dois métodos **equals()** e **hashCode()** sempre devem ser modificados juntos. Modifique a classe **Conta.java**, sobrepondo o método **equals()**, **hashCode()** e **toString()** de **Object** como na listagem abaixo.

```
/**
 * Testa a igualdade de um objeto com este
 */
@Override
public boolean equals(Object objeto) {

	boolean resultado = false;
	if (( objeto != null ) && ( objeto instanceof Conta )) {
		Conta c = (Conta) objeto;
		if (getNumero() == c.getNumero()) {
			resultado = true;
		}
	}
	return resultado;
}

/**
 * Gera código hash para tabelas de epalhamento
 */
@Override
public int hashCode() {

	return getNumero();
}

@Override
public String toString() {

	return getNumero() + "-" + getTitular().getNome();

}
```

Listagem 10.5 – Override do método equals() e hashCode()

3. Crie, compile e execute o código abaixo para comprovar que a lista não aceitará registros duplicados somente quando for implementado o **equals** e **hashCode,** o override no método **toString** é simplesmente para simplificar o valor a ser mostrado na console. Se quiser fazer um teste não implementado o **toString** irá notar que a regra de não duplicidade será mantida mas o valor impresso na console será desconhecido, irá mostrar o nome da classe concatenado com @ e um número.

```
import java.util.HashSet;

public class TesteHashSet {

	public static void main(String[] args) {

		HashSet<Cliente> clientes = new HashSet<Cliente>();
		clientes.add(new Cliente("Jesus"));
		// duplicado
		clientes.add(new Cliente("Jesus"));
		clientes.add(new Cliente("Mateus"));
		clientes.add(new Cliente("Maria"));
		// duplicado
		clientes.add(new Cliente("Maria"));
		clientes.add(new Cliente("Paulo"));
		clientes.add(new Cliente("João"));

		// imprimir com toString() de Cliente sem override de equals() e hashCode
		System.out.println(clientes);

		HashSet<Conta> contas = new HashSet<Conta>();
		contas.add(new Conta(new Cliente("Ze"), 5));
		contas.add(new Conta(new Cliente("Lucas"), 2));

		contas.add(new Conta(new Cliente("Pedro"), 1));
		// duplicado o numero da conta
		contas.add(new Conta(new Cliente("Maria"), 1));

		contas.add(new Conta(new Cliente("Joao"), 0));
		contas.add(new Conta(new Cliente("Ana"), 4));

		// imprimir com override de equals() e hashCode()
		System.out.println(contas);
	}
}
```

Listagem 10.6 – Testando HashSet com elementos duplicados

Com a execução do exemplo acima irá perceber que os primeiros registros é que será mantido, os registros que tentou inserir mas que o valor retornou o mesmo de um existente na lista foi ignorado e não adicionados.

2.2 – Classificando os elementos com TreeSet

1. Crie a classe **TesteTreeSet.java** conforme abaixo na **Listagem-10.7**, tente compilar e executar.

```
import java.util.TreeSet;

public class TesteTreeSet {

	public static void main(String[] args) {

		TreeSet<Cliente> clientes = new TreeSet<Cliente>();
		clientes.add(new Cliente("Jesus"));
		// duplicado
		clientes.add(new Cliente("Maria"));
		clientes.add(new Cliente("Mateus"));
```

```
                clientes.add(new Cliente("Maria"));
                // duplicado
                clientes.add(new Cliente("Maria"));
                clientes.add(new Cliente("Paulo"));
                clientes.add(new Cliente("JoÃ£o"));
                // imprimir com toString() de Cliente
                // sem implementar Comparable
                System.out.println(clientes);
        }

}
```

Listagem 10.7 – Usando TreeSet

2. Ao executar você deverá ser notificado de que ocorre uma exceção do tipo **ClassCastException**, isso deve-se ao fato de **TreeSet** tentar comparar os objetos que estão sendo adicionados através do método **add()**. O método lança essa exceção toda vez que não consegue comparar um objeto inserido na coleção. Para resolver este problema há duas soluções: implementar a **interface Comparable** ou usar um objeto comparador que implemente **Comparator** através do construtor **TreeSet(Comparator<? super E> comparator)**.

3. Modifique a classe **Cliente.java** para implementar a **interface Comparable** como segue abaixo e compile e execute novamente a classe **TesteTreeSet.java**.

```
import java.util.Vector;

public class Cliente implements Comparable<Cliente> {

        /**
         * Permite comparações, usando para classificar
         */
        public int compareTo(Cliente o) {
                // comparando somente campo nome
                return getNome().compareTo(o.getNome());
        }

}
```

Listagem 10.8 – Implementando Comparable

Exercício 3 – Usando coleção List

A **interface List** herda a **interface Collection** para definir uma coleção ordenada, permitindo elementos duplicados. A interface possui operações orientadas pela posição, habilidade de trabalhar com somente partes da lista.

Há duas implementações de propósito geral na **Framework Collection: ArrayList e LinkedList**. Qual das duas implementações utilizar vai depender de suas necessidades especificas. Se você precisar de suporte para acesso aleatório, com inserção ou remoção de elementos em qualquer parte da lista além do final desta, então **ArrayList** oferece uma implementação mais otimizada. Se com freqüência você precisar adicionar e remover elementos do meio da lista e só acessar elementos seqüencialmente, então a **LinkedList** oferece o melhor algoritmo.

1. Crie a classe **TesteArrayList.java** compile e execute a listagem abaixo:

```
import java.util.ArrayList;
import java.util.List;

public class TesteArrayList {
```

```
        public static void main(String[] args) {

                List<String> al = new ArrayList<String>(2);

                System.out.println(al + ", size = " + al.size());

                // Adiciona itens ao ArrayList
                al.add("Robaum");
                al.add("Usurpa");
                al.add("Omitte");

                System.out.println(al + ", size = " + al.size());

                // Remove item, usa equals e hashCode
                al.remove("Usurpa");
                System.out.println(al + ", size = " + al.size());

                // Checa se a lista contem o elemento especificado
                Boolean b = al.contains("Omitte");
                System.out.println("A lista contem Omitte = " + b);
                b = al.contains("Paumdu");
                System.out.println("A lista contem Paumdu = " + b);
        }

}
```

Listagem 10.9 – TesteArrayList

2. Modifique sua classe **Agência.java**, **Listagem-10.10,** como segue o código abaixo adicionando os construtores:

```
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

public class Agencia {

        //mantenha os atributos

        public Agencia( String nome ) {

                this.nome = nome;
                contas = new HashSet<Conta>();
        }

        public Agencia( String nome, Banco banco ) {

                this(nome);
                this.banco = banco;
        }
```

Listagem 10.10 – Classe Agencia

3. Crie a classe **TesteLinkedList.java** conforme abaixo, para testar uso de **LinkedList**.

```
import java.util.LinkedList;

public class TesteLinkedList {

        public static void main(String[] args) {

                // Cria LinkedList e adiciona 2 contas.
                LinkedList<Conta> ll = new LinkedList<Conta>();
```

```
                ll.add(new Conta(new Cliente("Falula"), 1));
                ll.add(new Conta(new Cliente("Bolula"), 2));
                System.out.println(ll + ", tamanho = " + ll.size());

                // Insere novo cliente no inicio e fim da LinkedList.
                ll.addFirst(new Conta(new Cliente("Zelula"), 3));
                ll.addLast(new Conta(new Cliente("Solula"), 4));

                System.out.println(ll);
                System.out.println(ll.getFirst() + ", " + ll.getLast());
                System.out.println(ll.get(1) + ", " + ll.get(2));

                // Remove o primero e o ultimo
                ll.removeFirst();
                ll.removeLast();
                System.out.println(ll);

                // Adiciona outro cliente na posicao 2 da lista
                ll.add(1, new Conta(new Cliente("Zoiudinha"), 5));
                System.out.println(ll);

                // Substitui objeto no indice 2
                ll.set(2, new Conta(new Cliente("Zoiudinha"), 5));
                System.out.println(ll);
        }
}
```

Listagem 10.11 – TesteLinkedList

Exercício 4 – Usando coleção Map

A **interface Map** não é uma extensão da **interface Collection**. Ao contrário, a interface **Map** possui sua própria **hierarquia**, mantendo uma associação **chave-valor**. A interface descreve um mapa de chaves para valores, sem chaves duplicadas, por definição.

Os métodos da interface podem ser vistos como três conjuntos de operações: alteração, consultas e visões alternativas.

As operações de alteração permitem que você adicione e remova uma chave-valor de um mapa. Ambos, chave e valor podem ser **null**. Entretanto você não deve adicionar um Map nele mesmo como uma chave ou valor.

As operações de consulta permitem que você verifique o conteúdo do mapa.

Outras operações lhe permitem trabalhar com um grupo de chaves ou valores da coleção. Uma vez que as chaves no mapa devem ser únicas, você pode recuperar estas chaves como um **Set**. E como os valores não necessitam ser únicos, você pode tê-los de volta como um **Collection**.

Em especial, é possível retornar uma coleção **Set** de elementos que implementa a interface **Map** através do método **entrySet()**. Cada objeto na coleção é um par de chave-valor especifico no **Map** em questão. Através desta coleção, você pode retornar uma chave ou valor, bem como mudar o valor de cada entrada. Entretanto, o conjunto de entradas se tornará invalido, podendo deixar seu comportamento indefinido, caso o **Map** em questão, seja modificado sem o uso do método **setValue()** da interface **Map**.

O **Framework Collections** provê duas implementações de propósito geral para interface **Map: HashMap e TreeMap.** Assim como todas as implementações, você vai usar a mais adequada a sua necessidade especifica. Para inserir, remover e localizar elementos no **Map**, o **HashMap** é sua melhor escolha. Entretanto se você precisar percorrer as chaves numa ordem classificada, então **TreeMap** é sua melhor alternativa. Dependendo do aumento de tamanho da coleção, pode ser mais rápido inserir elementos a um **HashMap**, então converter o mapa para um **TreeMap** e desta forma percorrer a coleção ordenada de chaves. O uso de um **HashMap** requer que a classe dos objetos das chaves adicionadas tenham uma implementação bem definida de **hashCode().** Com um **TreeMap** os elementos adicionados devem ser classificáveis.

4.1 – Usando HashMap

1. Modifique a classe **Agencia.java** sobreponto os métodos **equals, hashCode e toString** da classe **Object** utilizando o seu atributo **nome**.

2. Crie, compile e execute listagem abaixo.

```
import java.util.Collection;
import java.util.HashMap;
import java.util.Set;

public class TesteHashMap {

	public static void main(String[] args) {

		HashMap<Agencia, Cliente[]> hm = new HashMap<Agencia, Cliente[]>();

		// adiciona chave - agencia e valor = array Cliente
		hm.put(new Agencia("ag01"), new Cliente[] { new Cliente("Enricando Cardoso"), new
Cliente("Inacio Estole"), new Cliente("Luiz Ladrum"), });

		hm.put(new Agencia("ag02"), new Cliente[] { new Cliente("Henri Cando"), new
Cliente("Stolin Lu La"), new Cliente("Lara Pio"), });

		hm.put(new Agencia("ag03"), new Cliente[] { new Cliente("Sony Gando"), new
Cliente("Leiro Pisto"), new Cliente("Waga Oubum Du"), });

		// imprime a colecao
		System.out.println(hm);
		// pega as chaves
		Set chaves = hm.keySet();

		// imprime as chaves
		System.out.println(chaves);

		// pega os valores
		Collection<Cliente[]> valores = hm.values();

		// imprime os valores
		for (Cliente[] cs : valores) {
			for (Cliente c : cs) {
				System.out.println(c);
			}
		}
	}

}
```

Listagem 10.12 – Agências e Clientes

4.2 – Usando TreeMap

1. Para usar um **TreeMap** é necessário satisfazer o requisito da chave, que deve se comparável. Como nossa chave no exemplo é a classe **Agencia.java** devemos garantir que ele é comparável. Modifique a classe **Agencia.java**, redefinindo-a, de forma a implementar a **interface Comparable** como você fez para classe **Cliente.java**.

2. Crie, compile e execute listagem abaixo.

```
import java.util.Collection;
import java.util.HashMap;
import java.util.Set;
import java.util.TreeMap;

public class TesteTreeMap {

	public static void main(String[] args) {

		HashMap<Agencia, Cliente[]> hm = new HashMap<Agencia, Cliente[]>();

		// adiciona chave - agência e valor = array Cliente
		hm.put(new Agencia("ag01"), new Cliente[] { new Cliente("Enricando Cardoso"), new
Cliente("Inacio Estole"), new Cliente("Luiz Ladrum"), });

		hm.put(new Agencia("ag02"), new Cliente[] { new Cliente("Henri Cando"), new
Cliente("Stolin Lu La"), new Cliente("Lara Pio"), });

		hm.put(new Agencia("ag03"), new Cliente[] { new Cliente("Sony Gando"), new
Cliente("Leiro Pisto"), new Cliente("Waga Oubum Du"), });

		// criando TreeMap
		TreeMap<Agencia, Cliente[]> tm = new TreeMap<Agencia, Cliente[]>(hm);

		// imprime a coleção
		System.out.println(tm);

		// pega as chaves
		Set chaves = tm.keySet();

		// imprime as chaves
		System.out.println(chaves);

		// pega os valores
		Collection<Cliente[]> valores = tm.values();

		// imprime os valores
		for (Cliente[] cs : valores) {
			for (Cliente c : cs) {
				System.out.println(c);
			}
		}
	}

}
```

Listagem 10.13 – Agências e Clientes

Exercício 5 – Ordenando e Comparando Objetos nas Coleções

Sabemos agora que a comparação, classificação e outras regras utilizadas nas coleções dependem de métodos a serem sobrescritos e interfaces a serem implementadas.

1. Implemente os métodos **equals(), hashCode()** e **toString()** para as classes **Banco.java** e **Pessoa.java**.

2. Implemente a interface **Comparable** na classe **Banco.java** e **Conta.java**.