MC302EF – Lista de Exercícios

Considere a classe abaixo, usada para representar árvores binárias:

```
public class Node {

       Node left;
       Node right;
       String info;

       public Node(String info, Node left, Node right){
              this.left = left;
              this.right = right;
              this.info = info;
       }
}
```

1 - Inclua nessa classe o método **`altura()`** que calcula a *altura* a altura da árvore que tem por raiz o nó ao qual o método é aplicado. A *altura* de uma árvore é definida como zero se a árvore for vazia ou 1 + altura da maior sub-árvore.

```
public class Node {

      ...

   public int altura(){
        int h1 = 0, h2 = 0;
        if(left != null) h1 = left.altura();
        if(right != null) h2 = right.altura();
        if(h1 > h2) return 1+h1;
        return 1+h2;
        // ou apenas return (h1 > h2) ? return 1 + h1 : return 1 + h2;
   }
```

2 - Inclua nessa classe o método **`static boolean isomorfas(Node n1, Node n2)`** que verifica se duas árvores passadas como parâmetro são isomorfas. Duas árvores são ditas isomorfas se ambas são vazias ou se os conteúdos das raízes são iguais e as sub-árvores esquerdas e direitas são isomorfas.

```
public class Node { ...
       static boolean isomorfas(Node n1, Node n2){
              if(n1 == null) return n2 == null;
              if(n2 == null) return false;
              // n1 != null && n2 != null
              if(n1.info == null) {
                     if(n2.info != null) return false;
              } else if(! n1.info.equals(n2.info)) return false;
              return (isomorfas(n1.left,n2.left) && isomorfas(n1.right,n2.right));
       }
}
```

3 - Crie a classe Node2, derivada de Node, que inclui o campo *father*, usado para referenciar o 'nó pai' de um nó. O construtor da nova classe deve usar o construtor da classe mãe e preencher o campo **`father`** de forma adequada. A classe Node2 deve também implementar a interface **`Hierarquia`**, mostrada a seguir.

```
public interface Hierarquia{

  // Verifica se este nó descende do nó passado como parâmetro
  public boolean descende(Node n);

  // Verifica se o nó passado como parâmetro pertence a esta árvore.
  public Boolean pertence(Node n);

}
```

```
public class Node2 extends Node implements Hierarquia {

        Node2 father;

        public Node2(String info, Node2 left, Node2 right){
                super(info,left,right);
                if(left != null)  left.father = this;
                if(right != null) right.father = this;
        }

        @Override
        public boolean descende(Node n) {
                if(this.father == null) return false;
                if(this.father == n) return true;
                return this.father.descende(n);
        }

        @Override
        public boolean pertence(Node n) {
                if(n == null) return false;
                return ((Node2)n).descende(this);
        }

        /**
         * Versão 2 - busca por n através do percurso em pré-ordem.
         */
        public boolean pertence_2(Node n){
                if(n == null) return false;
                if(n == this) return true;
                if((left != null) &&  ((Node2)left).pertence_2(n))  return true;
                if((right != null) && ((Node2)right).pertence_2(n)) return true;
                return false;
        }
}
```

4. O método abaixo foi escrito na tentativa de trocar dois objetos:

```
public static void troca(Node n1, Node n2){
      Node t = n1;
      n1 = n2;
      n2 = t;
}
```

explique porque esse método não faz o que é esperado.

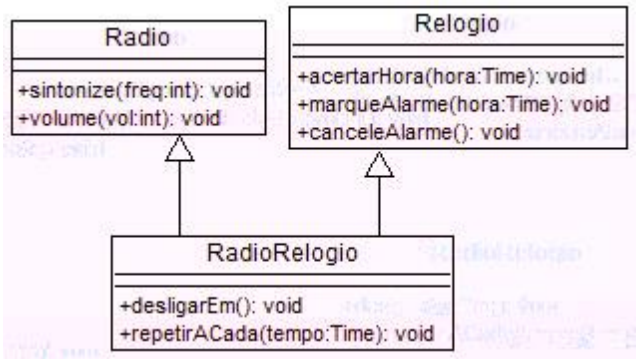
R. : O esperado é que após a troca, n1 se comporte exatamente como o 'antigo n2´ e n2 se comporte exatamente como o 'antigo n1'. Como em Java a única forma de passagem de parâmetros é por valor

(como em C, antes da chamada ao método o valor dos *parâmetros da chamada* é atribuído aos respectivos *parâmetros da definição* da função), qualquer alteração no valor dos *parâmetros da definição* da função é uma alteração apenas local.

5. Proponha um método que use uma forma alternativa de se obter o efeito desejado.

```
public static void troca(Node n1, Node n2){
     // supondo que n1 e n2 são diferentes de null
     String tInfo = n1.info;
     Node tLeft  = n1.left;
     Node tRight = n2.right;
     n1.info  = n2.info;  n2.info = tInfo;
     n1.left  = n2.left;  n2.left = tLeft;
     n1.Right = n2.right; n2.right = tRight;
}
```

6 - Considere o diagrama de classes abaixo.



Escreva o código Java correspondente a essas classes (não se preocupe com o 'corpo' dos métodos).

7- Considere um grafo representado por uma lista de objetos Node, conforme mostrado a seguir:

```
class Node{
        String name;
        List<Node> adjacents;
        public Node(String n) { name = n; adjacents = new ArrayList<Node>(); }
        public void addAdj(Node n) { adjacents.add(n); }
      }

// exempo de criação de um grafo
List<Node> graph = new ArrayList<Node>();
Node n1 = new Node("n1");
Node n2 = new Node("n2");
Node n3 = new Node("n3");
n1.addAdj(n2); n1.addAdj(n3);
n2.addAdj(n1); n2.addAdj(n3);
n3.addAdj(n1);
graph.add(n1); graph.add(n2); graph.add(n3);
```

Escreva o método List<Node> clone() que retorna o clone de um grafo passado como parâmetro. (2.0)

Observações:

-- a estrutura de objetos retornada por clone() deve ser formada por clones, ou seja, cópias, dos objetos que constituem o grafo original.

-- cada objeto da estrutura original deve ser clonado uma única vez.

```
	/**
	 * Método auxiliar: retorna o clone de um objeto Node (uso interno).
	 * Usa um mapa (clones) para garantir que cada nó é clonado uma única
vez.
	 * @param n       nó a ser clonado
	 * @param clones  mapa com os nós já clonados
	 * @return o clone do objeto Node passado como parâmetro
	 */
	private static Node klone(Node n, Map<Node,Node> clones){
		if(clones.containsKey(n)) return clones.get(n);
		else {
			Node res = new Node(n.name);
			clones.put(n, res);
			return res;
		}
	}

	/**
	 * Faz o clone de um grafo
	 * @param graph Lista de objetos Node representando o grafo
	 * @return clone do grafo
	 */
	static List<Node> clone(List<Node> graph){
		Map<Node,Node> clones = new HashMap<Node, Node>();
		List<Node> res = new ArrayList<Node>();
		for(Node n: graph){
			Node k = klone(n,clones);
			res.add(k);
			for(Node a: n.adjacents)
				k.addAdj(klone(a,clones));
		}
		return res;
	}

	/**
	 * Verifica se dois grafos são disjuntos.
	 * @param g1
	 * @param g2
	 */
	static void check(List<Node> g1, List<Node> g2){
		Set<Node> s1 = new HashSet<Node>(g1);
		boolean r = true;
		for(Node n: g2) if(s1.contains(n)) r = false;
		System.out.println("check:"+r);
	}
```