MC336 – Paradigmas de Programação
Primeiro Semestre de 2008
Terceira Prova

Nome:	**RA:**

- A prova tem duração de 2 horas.
- Não é permitida consulta a qualquer material.
- Qualquer tentativa de cola será punida com *nota zero no semestre* para todos os envolvidos.
- Não solte o grampo que segura as páginas da prova.
- Todas as questões devem ser respondidas na mesma folha do enunciado (é permitido o uso do verso da folha).

Questão	Nota
1	
2	
3	
4	
5	
Total	

Boa sorte.

1 – Desenhe um *modelo UML* com 3 classes e use-o para explicar os conceitos de *polimorfismo de inclusão* e de *acoplamento dinâmico* (da forma como são implementados em Java). O modelo UML compatível com os conceitos valerá 30% nota, enquanto cada explicação sobre os conceitos valerá 35% da nota.

2 – Considere as implementações das classes `Class1` e `Class2` e das interfaces `Interface1` e `Interface2`:

```
interface Interface1 {
    public void method1();
    public void method2();
    public void method5();
}
```

```
interface Interface2 {
    public void method4();
    public void method7();
}
```

```
class Class1 implements Interface1 {
    public void method1() {
        // block1
    }
    public void method2() {
        // block2a
    }
    public void method3() {
        // block3a
    }
    public void method5() {
        // block5
    }
    public void method6() {
        // block6a
    }
}
```

```
class Class2 implements Interface1, Interface2 {
    public void method1() {
        // block1
    }
    public void method2() {
        // block2b
    }
    public void method3() throws Exception {
        // block3a
        // block3b (throw Exception)
    }
    public void method4() {
        // block4
    }
    public void method5() {
        // block5
    }
    public void method6() {
        // block6a
        // block6b
    }
    public void method7() {
        // block7
    }
}
```

Suponha que cada bloco (`block1`, `block2a`, `block2b`, etc) seja formado por aproximadamente uma centena de linhas de código e que cada bloco é independente entre si. Reimplemente as classes `Class1` e `Class2`, utilizando uma classe auxiliar `ClassX` para juntar, *o máximo possível*, os trechos de código que estão repetidos em ambas as classes. Objetos das novas classes `Class1` e `Class2` devem ter comportamentos idênticos aos das classes implementadas acima, ou seja, devem dar suporte à exatamente os mesmos métodos que existiam antes. Não deve ser possível instanciar objetos de `ClassX`. A nova classe `Class1` não deve implementar diretamente nenhuma interface, mas tanto a nova `Class1` quanto a nova `Class2`, devem implementar (direta ou indiretamente) as mesmas interfaces que implementavam antes.

```
abstract class ClassX implements Interface1 {

    public void method1() {
      // block1
    }

    abstract public void method2();

    public void method5() {
      // block5
    }

    public void method6() {
      // block6a
    }

    protected void method3a() {
      // block3a
    }
}
```

```
class Class1X extends ClassX {

    public void method2() {
      // block2a
    }

    public void method3() {
      super.method3a();
    }
}
```

```
class Class2X extends ClassX implements Interface2 {

    public void method2() {
      // block2b
    }

    public void method3() throws Exception {
      super.method3a();
      // block3b: throws Exception
    }

    public void method4() {
      // block4
    }

    public void method6() {
      super.method6();
      // block6b
    }

    public void method7() {
      // block7
    }
}
```

3 – Considere o modelo UML da classe `Conta`:

Implemente a classe `Conta` em Java.

O método `numeroContas` deve retornar o número total de objetos da classe `Conta` criados até o momento, assim como o método `depositosTotais` deve retornar o valor total depositado em todas os objetos da classe `Conta`.

Já os métodos `deposito`, `saque` e `saldo` devem, respectivamente, implementar o deposito, saque e saldo em um objeto da classe `Conta`. Note que o método `deposito` e o método `saque` devem retornar verdadeiro apenas se a operação for válida, ou seja, o valor depositado ou sacado deve ser positivo e a conta deve ter um saldo compatível no momento da operação (objetos da classe `Conta` não podem ficar com saldo negativo). A variável `numero` dos objetos da classe `Conta` devem possuir números seqüenciais (o primeiro objeto da classe deve ter `numero` = 1).

```
class Conta {
    private static int numeroContas = 0;
    private static float depositosTotais = 0;
    protected float saldo;
    protected int numero;

    public static int numeroContas() {
      return numeroContas;
    }

    public static float depositosTotais() {
      return numeroContas;
    }

    public Conta() {
      numeroContas++;
      numero = numeroContas;
      saldo = 0;
    }

    public boolean deposito(float valor) {
      if (valor > 0) {
          saldo = saldo + valor;
          depositosTotais = depositosTotais + valor;
          return true;
      } else {
          return false;
      }
    }

    public boolean saque(float valor) {
      if ((valor > 0) && (valor <= saldo)) {
          saldo = saldo - valor;
          depositosTotais = depositosTotais - valor;
          return true;
      } else {
          return false;
      }
    }

    public float saldo() {
      return saldo;
    }
}
```

4 – Considere as definições da classe `List`, da interface `ISet` e da exceção `SetException` abaixo. Implemente uma classe `Set` em Java que dê suporte a operações sobre conjuntos previstas pela interface `ISet` (inserção, remoção, busca e impressão) e permita que o programa `Main` possa ser executado corretamente (sem modificações) e produza o resultado especificado. Note que apenas as operações `print` imprimem algo (tanto de `Set` quanto de `SetException`).

```
import java.util.Vector;

public class List {

    private Vector<Integer> list;

    public List() {
        list = new Vector<Integer>();
    }

    public void insert(Integer value) {
        list.add(value);
    }

    public Integer removeAt(int position) {
        return list.remove(position);
    }

    public int indexOf(Integer value) {
        return list.indexOf(value);
    }

    public void print() {
        for(int i=0; i < list.size(); i++) {
            System.out.println(list.get(i));
        }
    }
}
```

```
class SetException extends Exception {

    private static final int INSERT_ERROR = 1;
    private static final int REMOVE_ERROR = 2;
    private int exceptionType;

    private SetException(int type) {
        super();
        exceptionType = type;
    }

    public static SetException insertError() {
        return new SetException(INSERT_ERROR);
    }

    public static SetException removeError() {
        return new SetException(REMOVE_ERROR);
    }

    public void print() {
        if (exceptionType == INSERT_ERROR) {
            System.out.println("Insert Error");
        } else {
            System.out.println("Remove Error");
        }
    }
}
```

```
public interface ISet {

    public void insert(Integer value)
                throws SetException;

    public Integer remove(Integer value)
                   throws SetException;

    public boolean search(Integer value);

    public void print();

}
```

```
public class Main {

    public static void main(String argv[]) {

        ISet set = new Set();
        try {
            set.insert(2);
            set.insert(3);
            set.insert(1);
            set.insert(2);
            set.insert(4);
        } catch (SetException e) {
            e.print();
        }
        set.print();
        try {
            set.remove(2);
            set.remove(3);
            set.remove(4);
            set.remove(1);
            set.remove(2);
        } catch (SetException e) {
            e.print();
        }
        set.print();
    }
}
```

Saída do programa Main:

```
Insert Error
2
3
1
Remove Error
1
```

```
public class Set implements ISet{

    private List set;

    public Set() {
      this.set = new List();
    }

    public void insert(Integer value) throws SetException {
      if (search(value)) {
          throw SetException.insertError();
      } else {
          set.insert(value);
      }
    }

    public Integer remove(Integer value) throws SetException {
      if (search(value)) {
          return set.removeAt(set.indexOf(value));
      } else {
          throw SetException.removeError();
      }
    }

    public boolean search(Integer value) {
      return (set.indexOf(value) >= 0);
    }

    public void print() {
      set.print();
    }
}
```

5 – Determine o que será impresso por cada um dos 8 blocos de comandos da classe `Misterio`. Se algum bloco provocar um erro (de compilação ou de execução) indique apenas a palavra "ERRO" e avalie os blocos restantes (supondo que o bloco defeituoso foi removido).

```
public class Classe1 {

    protected int valor;

    public Classe1(int valor) {
        this.valor = valor;
    }

    public void metodo1() {
        System.out.println(this.valor);
    }

    public void metodo2() {
        this.valor++;
        metodo1();
    }
}
```

```
public class Classe2 extends Classe1 {

    public Classe2(int valor) {
        super(valor+1);
    }

    public void metodo1(int valor) {
        System.out.println(super.valor + valor);
    }

    public void metodo2() {
        super.valor--;
        metodo1();
    }
}
```

```
public class Classe3 extends Classe1 {

    public Classe3(int valor) {
        super(valor*valor);
    }

    public void metodo1() {
        super.metodo2();
        super.metodo1();
    }

    public void metodo2(int valor) {
        this.valor = super.valor*valor;
        super.metodo1();
    }
}}
```

```
public class Misterio {

    public static void main(String argv[]) {

        Classe1 classe1;
        Classe2 classe2;
        Classe3 classe3;

        // Bloco 1
        classe1 = new Classe1(1);
        classe1.metodo1();

        // Bloco 2
        classe2 = new Classe2(2);
        classe2.metodo1();

        // Bloco 3
        classe2.metodo2();

        // Bloco 4
        classe3 = new Classe3(3);
        classe3.metodo1();

        // Bloco 5
        classe1 = new Classe2(2);
        classe1.metodo2();

        // Bloco 6
        classe1.metodo1(3);

        // Bloco 7
        classe3 = classe1;
        classe3.metodo1();

        // Bloco 8
        classe3 = new Classe3(4);
        classe3.metodo2(3);
    }
}
```

Respostas:

Bloco 1: **1**	Bloco 2: **3**	Bloco 3: **2**	Bloco 4: **ERRO** `LOOP INFINITO`
Bloco 5: **2**	Bloco 6: **ERRO** `METODO INEXISTENTE`	Bloco 7: **ERRO** `TIPOS INCOMPATIVEIS`	Bloco 8: **48**

Rascunho (não será corrigido):