Estudos - COO

Tratamento de exceções:

```
try{
        //código que pode gerar exceções
}
catch (classeExceção objetoExceção){
        //codigo de tratamento
}
finally{
        //código a ser executado sempre
        //Instruções de liberação de recursos podem ser colocadas aqui
}
```

Invocar o método **toString** da exceção capturada, que exibe informações básicas sobre a exceção.

```
public static void metodo() throws classeExceção{
        /*
        Informar as exceções que esse método pode lançar. Os
        clientes do método são informados assim de que o método pode
        lançar essas exceções e de que elas deverão ser tratadas.
        */
}
```

Todas as classes de exceção herdam da classe **Exception**

Verificadas (método mnemônico: **verificadas pelo compilador**: se tiver com try e catch beleza, senão, dará erro de compilação)

precisa por **catch** ou **throws**, senão o compilador reclama.

Exemplo:

IOException //Causadas por condições fora do controle do programa

Não Verificadas:

Todos as subclasses de RuntimeException são exceções não verificadas.

O código para tratamento da exceção pode estar:

- no próprio método que a provocou
- ou em algum **método anterior na pilha de execução**.

Packages:

Para criar um pacote, coloque as classes e interfaces dentro de uma **pasta com o mesmo nome** dele e use a declaração **package** na primeira linha do código-fonte.

Pontos positivos:

- facilitar a localização de tipos
- facilitar a reutilização: programas podem importar classes de outros pacotes
- evitar conflitos de nomes
- fazer controle de acesso

Coleções:

For aprimorado

```
for(StackTraceElement element: traceElements){
   System.out.printf("%s\t", element.getClassName());
   System.out.printf("%s\t", element.getLineNumber());
   System.out.printf("%s\n", element.getMethodName());
}

String[] colors = { "MAGENTA", "RED", "WHITE", "BLUE", "CYAN" }
for(String color : colors){
        //do something
}
```

Principais métodos da Interface Collections

- int size()
- boolean isEmpty()
- boolean contains(Object element)
- boolean add(Object element)
- boolean remove(Object element)
- void clear()
- Iterator iterator()
- addAll(Collection c) - União
- removeAll (Collection c) - Diferença
- retainAll (Collection c) - Intersecção

List

É uma Collection que tem uma sequência definida e que pode conter elementos duplicados.

Como os arrays o **índice do primeiro elemento é zero.**

List pode ser implementada por:

- **ArrayList**
- **LinkedList**

List<Integer> L1;

List<Integer> L2;

L1 = new **ArrayList**<Integer>();

L2 = new **LinkedList**<Integer>();

L1.add(índice, valor); //ou
L1.add(valor); //desta forma ele será anexado ao fim da lista

Príncipais métodos de List:

- Object get(int index)
- Object set(int index, Object element)
- void add(int index, Object element)
- Object remove(int index)
- int indexOf(Object o)
- int lastIndexOf(Object o)
- ListIterator listIterator()

Pode ser usado um **Iterator** para navegar entre os elementos.

```
Iterator <Integer> it = L1.iterator();
        while(it.hasNext()) {
                it.next();
        }
}
```

Obs: **Não** se **pode adicionar ou modificar** nenhum elemento da Lista enquanto o iterator estiver percorrendo a mesma. Apenas a **remoção é permitida**.

Set

É uma coleção que contém **elementos únicos não duplicados**. A ordem em que os elementos são armazenados pode não ser a ordem na qual eles foram inseridos no conjunto.

HashSet

Elementos são adicionados de forma espalhada na lista de acordo com um HashCode correspondente a ele, tornando os métodos de **busca, add, remove O(1).**

Set<String> c1 = new **HashSet**<String>();
c1.add("elemento1");

TreeSet

A classe TreeSet implementa a interface **SortedSet** e usa a estrutura de árvore rubro-negra. A ordem é definida pelo método de comparação entre seus elementos. Sua complexidade assintótica para inserção é O(log(n)).

TreeSet<String> tree = new **TreeSet**<String>();
tree.add("Elemento1");

Principais Métodos de TreeSet:

- headSet(Object e) //ponteiro do subconj. de el. anteriores que o passado
- tailSet(Object e) //ponteiro de subconjunto de el. posteriores que o passado (incluindo ele próprio.

Principais Métodos (Set): Mesmos métodos de Collections

Mapas

É uma estrutura de dados que **associa chaves a valores** e **sem chaves duplicadas**

HashMap

Assim como o hashSet seus elementos são espalhados na inserção e sua busca/add/remove é O(1) novamente.

Map<Integer,String> mapa=new **Hashmap**<Integer, String>();
mapa.put(455,"vermelho");

TreeMap

A classe TreeMap implementa a interface SortedMap e mantém as suas chaves ordenadas

Map<Integer, String > mapa = new **TreeMap**<Integer, String>();
mapa.put(333,"branco");

Principais Métodos de Map(não confirmado):

- Object put(Object key, Object value)
- Object get(Object key)
- Object remove(Object key)
- boolean containsKey(Object key)
- boolean containsValue(Object value)
- int size()
- boolean isEmpty()
- void clear()

Serialização (se alguém puder vai ajudando a escrever)

A serialização é utilizada para **gravar ou ler arquivos** com informações de **objetos inteiros**, ou seja, com todos os seus atributos e métodos. Sempre deve possuir **try e catch** para casos como arquivo inexistente ou problemas com relação a permissão de acesso.

Para utilizar a serialização é necessário primeiramente **criar um fluxo** e a partir dele acessar o arquivo, seja para leitura ou para escrita e a classe a ser serializada **DEVE possuir implements Serializable**.

Escrita:

FileOutputStream caminho = new **FileOutputStream**(“nomeDoArquivo.extensao”);

ObjectOutputStream objArq = new **ObjectOutputStream**(caminho);

objArq.writeObject(objetoASerSerializado);

objArq.close();

Leitura

FileInputStream caminho = new **FileInputStream**(“nomeDoArquivo.extensao”);
ObjectInputStream objArq = new **ObjectInputStream**(caminho);

Objeto = (TipoDoObjeto).objArq.readObject();
objArq.close();

Leitura de Arquivo e Escrita de Arquivo com Caracteres

Para **leitura** de arquivos é utilizada a classe **FileReader**

Para escrita de arquivos é utilizada a classe **FileWriter**

Existem meios mais fáceis de se escrever e ler arquivos em java: FORMATTER e SCANNER, respectivamente.

Formatter

Formatter output = new Formatter(“nomearquivo.txt”);

output.format(“%s”, “String”);

output.close();

Scanner

Scanner input = new Scanner(new File(“nomearquivo.txt”));

while(input.hasNext){

    LeituraDoArquivo;

}

input.close();

Resolução da Prova COO - 2011

1.*Sendo as afirmações:*

I) O bloco finally quase sempre será executado, independentemente de ter ocorrido uma exceção ou de esta ter sido tratada ou não.
II) O bloco finally sempre será executado, inclusive quando o aplicativo fechar antes chamando o método System.exit().
III) Instruções de liberação de recursos podem ser colocadas no bloco finally.

R- Somente as afirmações I e III estão corretas.
*//***System.exit() termina o programa naquele ponto, não dando sequência à execução do programa.**

2. *Sendo as afirmações:*

I) |Para poder comparar objetos podemos implementar igual a un objeto Una clase es a interface Comparator ou implementar a interface Comparable.
II) Para adicionar um objeto a uma tabela hash é b) calculado o hashCode do objeto. Para que isso Cuando el desarrollo de una función está funcione corretamente é necessário verificar que o método hashCode de cada objeto retorne ( ) o mesmo valor para dois objetos, se eles são considerados iguais.
III)O método compareTo da interface Comparator compara dois objetos e retorna um inteiro negativo se o primeiro for menor do que o segundo; zero, se forem idênticos; e um valor positivo, caso contrário.

R-2.C(I e II verdadeiro)
*//***A interface Comparator não implementa compareTo, esse método é implementado por Comparable.**

3.*Dado o mapa a seguir, incluir a linha que falta para apagar todos os pares cuja chave é maior ou igual que 455 (use o método tailMap):*

Map<Integer, String > mapa = new TreeMap<Integer,String>();

mapa.put(455,"vermelho");
mapa.put(333,"branco");
mapa.put(678,"amarelo");
mapa.put(455,"azul");

/*Não aceita*/R- mapa = ((TreeMap<Integer, String>)mapa).tailMap(455);
//mapa é do tipo Map que não contém o método tailMap, sendo necessário fazer um cast para o tipo correto antes de utilizar o método.
(No exercicio era pedido para apagar os maiores que 455,mas esse codigo mantém as maiores que 455 só apaga o 333,Acredito que tem um erro, deveria usar um headMap)

Correção: O java não aceita o cast que foi dado na linha acima... tentem este código: mapa.tailMap(455).clear();

Correção: Linha de codigo q falta :
((TreeMap<Integer, String>) mapa).tailMap(455).clear();

4)*TreeSet x HashSet:*
***R*- Um TreeSet utiliza uma árvore rubro-negra como implementação ;**
Um HashSet usa uma tabela de espalhamento como implementação;
Um TreeSet gasta computacionalmente O(log(n)) para inserir, enquanto o HashSet gasta apenas O(1).

5.- *Porque quando inserimos a seguinte linha num programa:*
Thread.sleep(2000); //pause for 2 seconds
Acontece um erro de compilação?
R- Pois o método lança uma exceção verificada InterruptedException, ou seja, ela precisa ser tratada com o uso de throws ou catch.

6.*Existe uma classe Livro que tem o método toString que devolve o nome do livro e o preço.*
Dado o seguinte código:

```
List<Livro> list = new LinkedList< Livro>();
Livro s1=new Livro("Est. de Dados",145);
Livro s2=new Livro("Java",150);
Livro s3=new Livro("C++",120);
Livro s4=new Livro("Redes",130);
list.add(0, s1);
list.add(1, s2);
list.add(2, s3);
ListIterator<Livro> it= list.listIterator(list.size());
System.out.print(it.previous());
it.add(s4);
System.out.print(list);
```

O que será mostrado na tela quando o programa é executado?

R- C++ 120[Est. de Dados 145, Java 150, Redes 130, C++ 120]
//System.out.print(it.previous()); mostra **C++ 120**
//System.out.print(list); mostra **[Est. de Dados 145, Java 150, Redes 130, C++ 120]**

8.- *Incluir as linhas de código para escrever tree no arquivo data.ser (use serialização)*

```
private static final String names[] =
{ “amarelo", “verde", “preto", “marrom", “cinza", “branco", “laranja", “vermelho", “verde" };
SortedSet< String > tree = new TreeSet <String> (Arrays.asList( names) );
try{
```

```
FileOutputStream fluxo = new FileOutputStream("data.ser");
ObjectOutputStream objarq = new ObjectOutputStream(fluxo);
objarq.writeObject(names);
```

```
                objarq.close();

}
} catch (FileNotFoundException e) {
e.printStackTrace();
}
catch (IOException e) {
e.printStackTrace();
}
```

9. *Dada a seguinte classe que contem uma lista de nomes de animais e a quantidade correspondente de animais que existem num zoológico:*

```
public class Zoo {
        private List<String> animalList;
        private List<Integer> quantityList;

...
}
```

escrever os seguintes métodos:

//o construtor fica sem parâmetros mesmo?

//sim, pq o exercício só pede pra construir duas listas vazias.. Se elas são recebidas como parâmetro, elas podem ou não estar vazias.. vai depender do que o programador passar como parametro no código.

construtor: que cria duas listas do tipo ArrayList vazias.

```
public Zoo() {
                this.animalList = new ArrayList<String>();
                this.quantityList = new ArrayList<Integer>();
        }
```

método que imprime todos os animais do zoológico que estão com problemas de reprodução (quantidade menor ou igual que 2)

```
public void printAnimals(){
```

```
//usando o for aprimorado
// qual a razão de utilizar get (i-1) nesse caso?
        int index = 0;
                        for (Integer i : quantityList) {
                                if(i <= 2) {
                                        System.out.println(animalList.get(index));
                                }
                                index++;
                        }

}
```

ou

```
public void printAnimals(){
//usando o ListIterator
// ou também dá pra fazer com o iterator(), neste caso, é só mudar de ListIterator pra Iterator
        ListIterator<String> it = animalList.listIterator(0);
        ListIterator<Integer> it1 = quantityList.listIterator(0);
        while(it.hasNext()){
                String s = (String)it.next();
                Integer i = (Integer)it1.next();
                if(i<=2){
                        System.out.println(s + " - " + i);
                }
        }
}
```

método que remove todos os animais do zoológico (incluindo a quantidade correspondente) que estão com problemas de reprodução, ou seja aqueles cuja quantidade é menor ou igual que 2

```
public void removeAnimals(){
        ListIterator<String> it = animalList.listIterator(0);
        ListIterator<Integer> it1 = quantityList.listIterator(0);
        while(it.hasNext()){
                String s = (String)it.next();
                Integer i = (Integer)it1.next();
                if(i<=2){
                        it.remove();
                        it1.remove();
                }
        }
}
```

Escreva um método que devolva o maior número que existe na lista quantityList (use o método de Collections)

```
public Integer maxQuantityList(){
        return Collections.max(quantityList);
}
```

10. *Completar:*

A) Arquivos binários são criados com base em fluxos de ___**Bytes**_____

B) Para entrada baseada em caracteres é usada a classe ____**FileReader**_____

C)Para saída baseada em caracteres é usada a classe ____**FileWriter**_____

D) O compilador assegura que a exceção verificada é capturada (via blocos __**try**___/

___**catch**__) ou
declarada em uma cláusula ___**throws**___.

E) Todos as subclasses de RuntimeException são exceções __**NÃO VERIFICADAS**__.