02

Reconheça classes de exceções comuns e suas categorias

Para a prova, é necessário conhecer algumas exceptions clássicas do Java. Na sequência, vamos conhecer essas exceptions e entender em que situações elas ocorrem.

ArrayIndexOutOfBoundsException e IndexOutOfBoundsException

Um `ArrayIndexOutOfBoundsException` ocorre quando se tenta acessar uma posição que não existe em um array.

```
class Teste {
    public static void main(String[] args) {
        int[] array = new int[10];
        array[10] = 10; // Aqui ocorre
                        // ArrayIndexOutOfBoundsException.
    }
}
```

COPIAR CÓDIGO

Da mesma maneira, quando tentamos acessar uma posição não existente em uma lista , a exception é diferente, no caso `IndexOutOfBoundsException` :

```
class Teste {
    public static void main(String[] args) {
        ArrayList<String> lista = new ArrayList<String>();
```

```
        //Aqui ocorre IndexOutOfBoundsException
        String valor = lista.get(2);
    }
}
```

COPIAR CÓDIGO

NullPointerException

Toda vez que o operador `.` é utilizado em uma referência nula, um `NullPointerException` é lançado.

```
class Teste {
    public static void main(String[] args) {
        String s = null;
        s.length(); // Aqui ocorre uma NullPointerException
    }
}
```

COPIAR CÓDIGO

ClassCastException

Quando é feito um casting em uma referência para um tipo incompatível com o objeto que está na memória em tempo de execução, ocorre um `ClassCastException`.

```
class Teste {
    public static void main(String[] args) {
        Object o = "SCJP"; // String
```

```
        Integer i = (Integer)o; // Aqui ocorre
                                // ClassCastException.
    }
}
```

COPIAR CÓDIGO

NumberFormatException

Um problema comum que o programador enfrenta no dia a dia é ter que "transformar" texto em números. A API do Java oferece diversos métodos para tal tarefa. Porém, em alguns casos não é possível "parsear" o texto, pois ele pode conter caracteres incorretos.

```
class Teste {
    public static void main(String[] args) {
        String s = "ABCD1";

        // Aqui ocorre um NumberFormatException.
        int i =    Integer.parseInt(s);
    }
}
```

COPIAR CÓDIGO

IllegalArgumentException

Qualquer método deve verificar se os valores passados nos seus parâmetros são válidos. Se um método constata que os parâmetros estão inválidos, ele deve informar quem o invocou que há problemas nos valores passados na invocação. Para isso, é aconselhado que o método lance `IllegalArgumentException` .

```
class Teste {
    public static void main(String[] args) {
        try {
            divideEImprime(5,0);
        } catch (IllegalArgumentException e) {
            // tratamento.
        }
    }

    public static void divideEImprime(int i, int j) {
        if(j == 0) { // Evita dividir por zero.
            throw new IllegalArgumentException();
        }
        System.out.println(i/j);
    }
}
```

COPIAR CÓDIGO

IllegalStateException

Suponha que uma pessoa possa fazer três coisas: dormir, acordar e andar. Para andar, a pessoa precisa estar acordada. A classe `Pessoa` modela o comportamento de uma pessoa. Ela contém um atributo `boolean` que indica se a pessoa está acordada ou dormindo e um método para cada coisa que uma pessoa faz (`dormir()` , `acordar()` e `andar()`).

O método `andar()` não pode ser invocado enquanto a pessoa está dormindo. Mas, se for, ele deve lançar um erro de execução. A biblioteca do Java já tem uma classe pronta para essa situação, a classe é a `IllegalStateException` . Ela significa que o estado atual do objeto não permite que o método seja executado.

```
class Pessoa {
    boolean dormindo = false;

    void dormir() {
        this.dormindo = true;
        System.out.println("dormindo...");
    }
    void acordar() {
        this.dormindo = false;
        System.out.println("acordando...");
    }
    void andar() {
        if(this.dormindo) { // Só pode andar acordado.
            throw new IllegalStateException("Deveria estar
                                            acordado!");
        }

        System.out.println("andando...");
    }
}
```

COPIAR CÓDIGO

ExceptionInInitializerError

No momento em que a máquina virtual é disparada, ela não carrega todo o conteúdo do *classpath*, em outras palavras, ela não carrega em memória todas as classes referenciadas pela sua aplicação.

Uma classe é carregada no momento da sua primeira utilização. Isso se dá quando algum método estático ou atributo estático são acessados ou quando um objeto é criado a partir da classe em questão.

No carregamento de uma classe, a JVM pode executar um trecho de código definido pelo programador. Esse trecho deve ficar no que é chamado bloco estático.

```
class A {
    static {
        // trecho a ser executado no carregamento da classe.
    }
}
```

COPIAR CÓDIGO

É totalmente possível que algum erro de execução seja gerado no bloco estático. Se isso acontecer, a JVM vai "embrulhar" esse erro em um `ExceptionInInitializerError` e dispará-lo.

Esse erro pode ser gerado também na inicialização de um atributo estático se algum problema ocorrer. Exemplo:

```
class A {
    static {
        if(true)
            throw new RuntimeException("nao vou deixar
nao...");
    }
}

public class Teste {

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

```
    }
}
```

COPIAR CÓDIGO

Gera o erro de inicialização:

```
Exception in thread "main"
java.lang.ExceptionInInitializerError
    at Teste.main(Teste.java:11)
Caused by: java.lang.RuntimeException: nao vou deixar nao...
    at A.<clinit>(Teste.java:4)
    ... 1 more
```

COPIAR CÓDIGO

StackOverflowError

Todos os métodos invocados pelo programa Java são empilhados na **Pilha de Execução**. Essa pilha tem um limite, ou seja, ela pode estourar:

```
class Teste {
    public static void main(String[] args) {
        metodoSemFim();
    }

    static void metodoSemFim() {
        metodoSemFim();
    }
}
```

COPIAR CÓDIGO

Repare que, nesse exemplo, o `metodoSemFim()` chama ele mesmo (recursão). Do jeito que está, os métodos serão empilhados eternamente e a pilha de execução vai estourar.

NoClassDefFoundError

Na etapa de compilação, todas as classes referenciadas no código-fonte precisam estar no *classpath*. Na etapa de execução também. O que será que acontece se uma classe está no *classpath* na compilação mas não está na execução? Quando isso acontecer será gerado um `NoClassDefFoundError` .

Para gerá-lo, podemos criar um arquivo com duas classes onde uma referencia a outra:

```
class OutraClasse {

}
class Teste {
    public static void main(String[] args) {
        new OutraClasse();
    }
}
```

COPIAR CÓDIGO

Compilamos o arquivo, gerando dois arquivos `.class` . Aí apagamos o arquivo `OutraClasse.class` . Pronto, o Java não será capaz de encontrar a classe, dando um erro, `NoClassDefFoundError` .

OutOfMemoryError

Durante a execução de nosso código, o Java vai gerenciando e limpando a memória usada por nosso programa automaticamente, usando o **garbage collector** (GC). O GC vai remover da memória todas as referências de objetos que não são mais utilizados, liberando o espaço para novos objetos. Mas o que acontece quando criamos muito objetos, e não os liberamos? Nesse cenário, o GC não vai conseguir liberar memória, e eventualmente a memória livre irá acabar, ocasionando um `OutOfMemoryError` .

O código para fazer um erro do gênero é simples, basta instanciar infinitos objetos, sem permitir que o garbage collector jogue-os fora. Fazemos isso com `String` s para que o erro aconteça logo:

```
void metodo() {
    ArrayList<String> objetos = new ArrayList<String>();
    String atual = "";
    while(true) {
        atual += " ficou maior";
        objetos.add(atual);
    }
}
```

COPIAR CÓDIGO