02

Crie um bloco try-catch e determine como exceções alteram o fluxo normal de um programa

O programador pode definir um tratamento para qualquer tipo de erro de execução. Antes de definir o tratamento, propriamente, é necessário determinar o trecho de código que pode gerar um erro na execução. Isso tudo é feito com o comando `try-catch` .

```
try {
    // trecho que pode gerar um erro na execução.
} catch (Throwable t) { // pegando todos os possíveis erros de
                        //execução.
    // tratamento para o possível erro de execução.
}
```

COPIAR CÓDIGO

A sintaxe do `try-catch` tem um bloco para o programador definir o trecho de código que *pode* gerar um erro de execução. Esse bloco é determinado pela palavra `try` . O programador também pode definir quais tipos de erro ele quer pegar para tratar. Isso é determinado pelo argumento do `catch` . Por fim, o tratamento é definido pelo bloco que é colocado após o argumento do `catch` .

Durante a execução, se um erro acontecer, a JVM redireciona o fluxo de execução da linha do bloco do `try` que gerou o erro para o bloco do `catch` . Importante! As linhas do bloco do `try` abaixo daquela que gerou o erro não serão executadas.

Fazer um `catch` em `Throwable` não é uma boa prática, pois todos os erros possíveis são tratados pela aplicação. Porém, os `Error` s não deveriam ser

tratados pela aplicação, já que são de responsabilidade da JVM. Assim, também não é boa prática dar `catch` em `Error` s.

Modificando o argumento do `catch` , o programador define quais erros devem ser pegos para serem tratados.

```
try {
    // trecho que pode gerar um erro na execução.
} catch (Exception e) { // pegando todas as exceptions.
    // tratamento para o possível erro de execução.
}
```

COPIAR CÓDIGO

Para a prova, é fundamental saber quando o programador pode ou não pode usar o `try-catch` . A única restrição de uso do `try-catch` envolve as checked exceptions. Qual é a regra? O programador só pode usar `try-catch` em uma checked exception se o código do bloco do `try` pode realmente lançar a checked exception em questão.

```
try {
    System.out.println("não acontece SQLException");
} catch(SQLException e){ // pegando SQLException.
    // tratamento.
}
```

COPIAR CÓDIGO

Esse código **não** compila pois o trecho envolvido no bloco do `try` nunca geraria a checked `SQLException` . O compilador avisa com um erro de "unreachable code". Já o exemplo a seguir compila, pois pode ocorrer um `FileNotFoundException` :

```
try {
    new java.io.FileInputStream("a.txt");
} catch(java.io.FileNotFoundException e){
    // tratamento.
}
```

COPIAR CÓDIGO

O código a seguir não tem nenhum problema, pois o programador pode usar o `try-catch` em qualquer situação para os erros de execução que não são checked exceptions.

```
try {
    System.out.println("Ok");
} catch (RuntimeException e) { // pegando RuntimeException
                               // (unckecked).
    // tratamento.
}
```

COPIAR CÓDIGO

Podemos pegar tudo, exceptions e erros:

```
try {
    System.out.println("Ok");
} catch (Throwable e) {
    // tratamento
}
```

COPIAR CÓDIGO

Quando a exception é pega, o fluxo do programa é sair do bloco `try` e entrar no bloco `catch`, portanto, o código a seguir imprime `peguei` e `continuando normal`:

```
String nome = null;
try {
    nome.toLowerCase();
    System.out.println("segunda linha do try");
} catch(NullPointerException ex) {
    System.out.println("peguei");
}
System.out.println("continuando normal");
```

COPIAR CÓDIGO

Mas, se a exception que ocorre não é a que foi definida no catch, a chamada do método para e volta, jogando a exception como se não houvesse um `try/catch`. O cenário a seguir demonstra essa situação e não imprime nada:

```
String nome = null;
try {
    nome.toLowerCase();
    System.out.println("segunda linha do try");
} catch(IndexOutOfBoundsException ex) {
    System.out.println("peguei");
}
System.out.println("continuando normal");
```

COPIAR CÓDIGO

Lembre-se sempre do polimorfismo, portanto, pegar `IOException` é o mesmo que pegar todas as filhas de `IOException` também. O código a seguir trata o caso de o arquivo não existir além de todas as outras filhas de `IOException`:

```
try {
    new java.io.FileInputStream("a.txt");
} catch(java.io.IOException e){
    // tratamento.
}
```

COPIAR CÓDIGO

Bloco finally

Tem coisas que não podemos deixar de fazer em hipótese alguma. Seja no sucesso ou no fracasso, temos obrigação de cumprir com algumas tarefas.

Imagine um método que conecta com um banco de dados. Não importa o que aconteça, no fim desse método a conexão deveria ser fechada. Durante a comunicação com o banco de dados, há o risco de ocorrer uma `SQLException` .

```
void metodo(){
    try {
        abreConexao();
        fazConsultas();
        fechaConexao();
    } catch (SQLException e) {
        // tratamento
    }
}
```

COPIAR CÓDIGO

Nesse código, há um grande problema: se um `SQLException` ocorrer durante as consultas, a conexão com o banco de dados não será fechada. Para tentar resolv

esse problema, o bloco do `catch` poderia invocar o método `fechaConexao()`. Então, se acontecesse um `SQLException` o bloco do `catch` seria executado e, consequentemente, a conexão seria fechada.

Mas ainda não solucionamos o problema, pois outro tipo de erro poderia acontecer nas consultas. Por exemplo, uma `NullPointerException` que não está sendo tratada. Para resolver o problema de fechar a conexão, um outro recurso do Java será utilizado, o bloco **finally**. Esse bloco é sempre executado, tanto no sucesso quanto no fracasso por qualquer tipo de erro.

```
void metodo(){
    try {
        abreConexao();
        fazConsultas(); // Não precisa mais fechar a conexao
                        // aqui.
    } catch(SQLException e) {
        // tratamento
    } finally {
        fechaConexao(); // fechando a conexao no sucesso ou no
                        // fracasso.
    }
}
```

COPIAR CÓDIGO

Para melhor entender o fluxo do `try-cacth` com o `finally`, veja o próximo exemplo.

```
class A {
    void metodo() {
        try{
```

```
            //A
            //B
        }catch(SQLException e){
            //C
        }finally{
            //D
        }
        //E
    }
}
```

COPIAR CÓDIGO

- Em uma execução normal, sem erros nem exceções, ele executaria `A`, `B`, `D`, `E`.
- Com `SQLException` em `A`, ele executaria `C`, `D`, `E`.
- Com `NullPointerException` em `A`, ele executaria apenas `D` e sairia.
- Se `A` fosse um `System.exit(0);`, ele apenas executa `A` e encerra o programa.
- Se ocorresse um erro `A`, executaria apenas `D` (dependendo do erro).

Uma outra maneira um pouco menos convencional de usar o `finally` é sem o bloco `catch`, como no exemplo a seguir.

```
class A{
  void metodo() {
    try {
      System.out.println("imprime algo");
    } finally {
      // sempre permite fechar
    }
```

```
        }
    }
```

COPIAR CÓDIGO