02

Invoque um método que joga uma exceção

Eventualmente, um método qualquer não tem condição de tratar um determinado erro de execução. Nesse caso, esse método pode deixar passar o erro para o próximo método na pilha de execução.

Para deixar passar qualquer erro de execução que não seja uma checked exception, é muito simples: basta não fazer nada.

```
class Teste {

    void primeiro(){
        System.out.println("primeiro antes");
        this.segundo();
        System.out.println("primeiro depois");
    }

    void segundo() {
        String s = null;
        System.out.println("segundo antes");
        s.length();
        System.out.println("segundo depois");
    }
}
```

COPIAR CÓDIGO

O segundo método declara uma variável não primitiva e a inicializa com `null`. Logo em seguida, ele utiliza o operador `.` em uma referência que sabemos estar nula. Nesse ponto, na hora da execução, um `NullPointerException` é gerado.

Perceba que não há `try-catch` no segundo método, então ele não está pegando e tratando o erro, mas sim deixando-o passar. O primeiro método não define o `try-catch`, ou seja, também deixa passar o `NullPointerException`. O resultado é a impressão de `primeiro antes`, `segundo antes`.

Agora, para deixar passar uma checked exception, o método é obrigado a deixar explícito (avisado) que pretende deixar passar. Na assinatura do método, o programador pode deixar avisado que pretende deixar passar determinados erros de execução. Isso é feito através da palavra-chave `throws`.

```
class Teste {

    void primeiro(){
        try {
            System.out.println("primeiro antes");
            this.segundo();
            System.out.println("primeiro depois");
        } catch(IOException e) {
            // tratamento.
            System.out.println("primeiro catch");
        }
        System.out.println("primeiro fim");
    }

    void segundo() throws IOException {
        System.out.println("segundo antes");
        System.in.read(); // pode lançar IOException
        System.out.println("segundo depois");
    }
}
```

COPIAR CÓDIGO

O segundo método invoca o `read()` no `System.in`. Essa invocação pode gerar um `IOException`, de modo que o segundo método tem duas alternativas: ou pega e trata o possível erro ou o deixa passar. Para deixar passar, o comando `throws` deve ser utilizado na sua assinatura do segundo método. Isso indicará que um `IOException` pode ser lançado.

Dessa forma, o primeiro método que invoca o segundo pode receber uma `IOException`. Então, ele também tem duas escolhas: ou pega e trata usando `try-catch`, ou deixa passar usando o `throws`. O resultado é a impressão de `primeiro antes`, `segundo antes`, `primeiro catch` e `primeiro fim`.

Gerando um erro de execução

Qualquer método, ao identificar uma situação errada, pode criar um erro de execução e lançar para quem o chamou. Vale lembrar que os erros de execução são representados por objetos criados a partir de alguma classe da hierarquia da classe `Throwable`, logo, basta o método instanciar um objeto de qualquer uma dessas classes e depois lançá-lo.

Se o erro não for uma checked exception, basta criar o objeto e utilizar o comando `throw` para lançá-lo na pilha de execução (não confunda com o `throws`):

```
class Teste {

    void primeiro(){
        try {
            this.segundo();
        } catch (RuntimeException e) {
            // tratamento.
```

```
        }
    }

    void segundo() {
        throw new RuntimeException();
    }
}
```

COPIAR CÓDIGO

Se o erro for uma checked exception, é necessário também declarar na assinatura do método o comando `throws` :

```
class Teste {

    void primeiro(){
        try {
            this.segundo();
        } catch(Exception e) {
            // tratamento.
        }
    }

    void segundo() throws Exception {
        throw new Exception();
    }
}
```

COPIAR CÓDIGO

Podemos ainda criar nossas próprias exceções, bastando criar uma classe que entre na hierarquia de `Throwable` .

```
class MinhaException extends Exception{}
```

COPIAR CÓDIGO

Em qualquer lugar do código, é opcional o uso do `try` e `catch` de uma unchecked exception para compilar o código. Em uma checked exception, é obrigatório o uso do `try/catch` ou `throws` .

O exemplo a seguir mostra uma unchecked exception sendo ignorada e o erro vazando, e nada será impresso:

```
public class Teste {

    public static void main(String[] args) {
        metodo();
        System.out.println("Apos a invocacao do metodo");
    }

    private static void metodo() {
        int[] i= new int[10];
        System.out.println(i[15]);
        System.out.println("Apos a exception");
    }

}
```

COPIAR CÓDIGO

Ao pegarmos a exception, será impresso também "Apos a invocacao do metodo" uma vez que após o catch, o fluxo volta ao normall:

```
public class Teste {

    public static void main(String[] args) {
        try {
            metodo();
        } catch(RuntimeException ex) {
            System.out.println("Exception pega");
        }
        System.out.println("Apos a invocacao do metodo");
    }

    private static void metodo() {
        int[] i= new int[10];
        System.out.println(i[15]);
        System.out.println("Apos a exception");
    }

}
```

COPIAR CÓDIGO

Podemos ter também múltiplas expressões do tipo `catch`. Nesse caso, será invocada somente a cláusula adequada, e não as outras. No código a seguir, se o `metodo2` jogar uma `ArrayIndexOutOfBoundsException`, será impresso `runtime`:

```
void metodo1() {
    try {
        metodo2();
    } catch(IOException ex) {
        System.out.println("io");
    } catch(RuntimeException ex) {
        System.out.println("runtime");
```

```
        } catch(Exception ex) {
            System.out.println("exception qualquer");
        }
    }
```

COPIAR CÓDIGO

E a ordem faz diferença? Sim, o Java procura o primeiro `catch` que pode trabalhar a `Exception` adequada.

Repare que `RuntimeException` herda de `Exception` e, portanto, deve vir antes da mesma na ordem de *catches*.

Caso ela viesse depois, ela nunca seria invocada, pois o Java verificaria que toda `RuntimeException` é `Exception` e `Exception` teria tratamento de preferência (por sua ordem). O exemplo a seguir não compila por este motivo:

```
void metodo1() {
    try {
        metodo2();
    } catch(IOException ex) {
        System.out.println("io");
    } catch(Exception ex) {
        System.out.println("exception qualquer");
    } catch(RuntimeException ex) {
        // não compila pois jamais será executado
        System.out.println("runtime");
    }
}
```

COPIAR CÓDIGO

Cuidado também com exceptions nos inicializadores:

```
class AcessoAoArquivo {
    // não compila, pois ao instanciar, pode dar IOException,
    // mas o construtor não fala nada
    private InputStream is = new
FileInputStream("entrada.txt");
}
```

COPIAR CÓDIGO

Nesses casos, precisamos dizer no construtor que a `Exception` pode ser jogada:

```
class AcessoAoArquivo {
    private InputStream is = new
FileInputStream("entrada.txt");

    AcessoAoArquivo() throws IOException{
        // estou avisando os clientes dessa classe
        // que ao instanciar pode dar essa exception
        // e agora compila
    }
}
```

COPIAR CÓDIGO