01

Introdução a exceções

Transcrição

> Você pode fazer o DOWNLOAD (https://caelum-online-public.s3.amazonaws.com/834-java-excecoes/01/java4-projetos-inicias.zip) dos projetos criados no curso anterior.

Começaremos, de fato, o estudo sobre as **exceções**, um dos tópicos mais difíceis. Anteriormente, vimos a **pilha de execução**, e aprendemos que o método no topo dela, é o que está sendo executado.

Exceções: O que são e para que servem?

As exceções são problemas que acontecem na hora de compilar o código. Considerando que existe uma variedade imensa, elas possuem nomes explicativos e, às vezes, mostram claramente o motivo de seu surgimento, facilitando a identificação delas.

No exemplo a seguir, usaremos classes e métodos vistos anteriormente. Suponhamos que você queira instanciar uma conta do tipo `ContaCorrente`, porém **nula**. Ou seja, sem *agência* e *número de conta*. Em seguida, depositaremos `200` reais nela:

```
public class TesteContas {
```

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

        //instancia para testar exceção!!!!
        ContaCorrente conta = null;
        outra.deposita(200.0);

        //instancia da conta corrente
        ContaCorrente cc = new ContaCorrente(111, 111);
        cc.deposita(100.0);

        //instancia da conta poupança
        ContaPoupanca cp = new ContaPoupanca(222, 222);
        cp.deposita(200.0);
    }
}
```

COPIAR CÓDIGO

Se rodarmos essa classe, no console, aparecerá a seguinte mensagem:

```
Exception in thread "main" java.lang.NullPointerException
        at TesteContas.main(TesteContas.java:7)
```

COPIAR CÓDIGO

Podemos ver que essa referência não foi inicializada corretamente, e esse é um exemplo bem trivial. Em alguns casos, não conseguimos enxergar se essa referência foi inicializada de forma correta ou não, o que pode ocasionar a exibição desse problema no console.

Mas, quando isso acontece, o desenvolvedor precisa entender o que houve e o que ele fez de errado. Considerando que a exceção se chama `NullPointerException` e nós inicializamos a variável `conta` como **nula**, deduzimos que ela faz referência à variável.

Vamos analisar mais um exemplo. Dentro da `main`, colocaremos a seguinte operação:

```
public class TesteContas {
    public static void main(String[] args) {

        int a = 3;
        int b = a / 0;
    }
}
```

COPIAR CÓDIGO

O que vai acontecer? *Lembrando que isso não dará erro de sintaxe*.

Novamente algo será exibido no console:

```
Exception in thread "main" java.lang.ArithmeticException: / by ze
        at TesteContas.main(TesteContas.java:7)
```

COPIAR CÓDIGO

`/ by zero`, ou seja, nem o Java consegue dividir algo por zero. Esse tipo de problema também não deveria acontecer, mas em um ambiente mais complexo, é realmente mais difícil prever.

A **primeira motivação** é que erros e exceções acontecem. Portanto, precisamos saber lidar com eles. Vamos nos colocar em uma nova situação, ainda sobre exemplo anterior:

Na classe **Conta**, existe um método `saca()`, que recebe um valor como parâmetro.

```
public boolean saca(double valor) {
    if(this.saldo >= valor) {
        this.saldo -= valor;
        return true;
    } else {
```

```
            return false;
        }
    }
```

COPIAR CÓDIGO

Se funcionar, devolveremos `true`. Caso contrário, devolveremos `false`. Nessa implementação, partiremos do princípio de que existe apenas um motivo para que o saque não funcione: *saldo insuficiente*. Considerando um cenário real, existem mais motivos que impedem o saque. Pode ser o limite diário, o horário comercial, o banco pode estar fechado nesse momento e achar que nós poderíamos fraudar ou qualquer outro motivo.

Como podemos sinalizar para quem chamar o método `saca()`, que o saque não está funcionando por um motivo específico?

O método retorna somente "Funcionou" ou "Não funcionou", mas queremos que ele nos retorne motivos específicos quando *não funcionar*. Para isso, usamos as **exceções**.

Vamos voltar à classe `Fluxo` e ver como isso funciona junto à pilha.

```
public class Fluxo {

    public static void main(String[] args) {
        System.out.println("Ini do main");
        metodo1();
        System.out.println("Fim do main");
    }

    private static void metodo1() {
        System.out.println("Ini do metodo1");
        metodo2();
        System.out.println("Fim do metodo1");
    }
```

```
    private static void metodo2() {
        System.out.println("Ini do metodo2");
        for(int i = 1; i <= 5; i++) {
            System.out.println(i);
        }
        System.out.println("Fim do metodo2");
    }
}
```

COPIAR CÓDIGO

Introduziremos um erro simples no `metodo2()` para nos ajudar a entender melhor. Declararemos uma nova variável e a dividiremos por zero:

```
private static void metodo2() {
    System.out.println("Ini do metodo2");
    for(int i = 1; i <= 5; i++) {
        System.out.println(i);
        int a = i / 0;
    }
    System.out.println("Fim do metodo2");
}
```

COPIAR CÓDIGO

Sabemos que não é uma boa ideia, mas é importante ver como realmente entendemos essa exceção.

Vamos executar clicando com o direito e selecionando "Run As > Java Application". No console, teremos a seguinte saída:

```
Ini do main
Ini do metodo1
Ini do metodo2
1
Exception in thread "main" java.lang.ArithmeticException: / by
        at Fluxo.metodo2(Fluxo.java:19)
```

```
        at Fluxo.metodo1(Fluxo.java:11)
        at Fluxo.main(Fluxo.java:5)
```

COPIAR CÓDIGO

Repare no que aparece abaixo de `ArithmeticException`. Existe algo familiar?

Exatamente! A **pilha de execução**. Em cima do `main`, temos o `metodo1` e, acima dele, temos o `metodo2`.

Console (Execução normal)

```
Ini do main
Ini do metodo1
Ini do metodo2
1
2
3
4
5
Fim do metodo2
Fim do metodo1
Fim do main
```

Console (Execução com exceção)

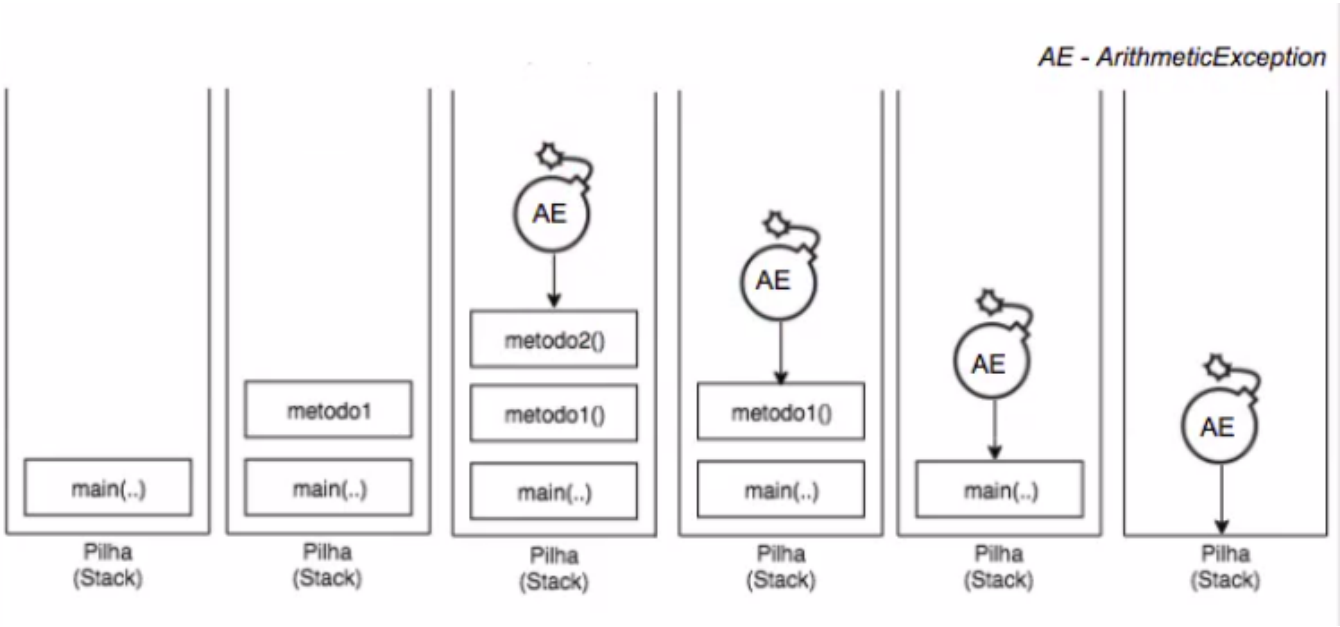
```
Ini do main
Ini do metodo1
Ini do metodo2
1
Exception in thread "main"
java.lang.ArithmeticException: / by zero
    at Fluxo.metodo2(Fluxo.java:19)
    at Fluxo.metodo1(Fluxo.java:11)
    at Fluxo.main(Fluxo.java:5)
```

Para acessibilidade:

Console (Execução Normal)	Console (Execução com Exceção)
Ini do main	Ini do main
Ini do metodo1	Ini do metodo1
Ini do metodo2	Ini do metodo2
1	1
2	Exception in thread "main"
3	java.lang.ArithmeticException: / by zero
4	at Fluxo.metodo2(Fluxo.java:19)
5	at Fluxo.metodo2(Fluxo.java:11)
Fim do metodo2	at Fluxo.main(Fluxo.java:5)
Fim do metodo1	
Fim do main	

Perceba que o Console de Execução com Exceção é semelhante ao Normal, até `1`. Além da exibição do **nome** (`ArithmeticException`) e da **mensagem** (`/ by zero`), após `1`, a execução muda — por causa da exceção — e é exibida a **pilha de execução**.

Assim, percebemos que as exceções mudam o fluxo, pois elas fazem parte do controle dele. Mas, por que o fluxo mudou? A partir do momento em que o `metodo2()` entrou no laço, apareceu a exceção "Não foi possível realizar a divisão por zero". Podemos imaginar que o Java jogou uma "bomba" em cima da pilha, ou seja, no método que está no topo dela.



Para acessibilidade: Podemos imaginar a pilha de execução, como um copo que recebe várias camadas, chamadas de *métodos*. A ordem de surgimento delas está de acordo com o método que é chamado. Como todo método é chamado a partir da `main()`, ele sempre estará no fundo do copo, pois é sempre o primeiro.

A `main()`, por sua vez, chama o `metodo1()`, que chamará o `metodo2()`. Como o `metodo2()` foi o último a ser chamado, ele está no topo do copo. A bomba (*exceção*) cai no `metodo2()` e, como ele não tem nenhum bloco de código que possa tratar essa bomba, o `metodo2()` sai do copo, e cai na função anterior, ou seja, no `metodo1()`.

Por sua vez, o `metodo1()` também não possui o bloco de código para tratar essa exceção, e por isso, o método sai da pilha, transferindo a exceção para `main()`; que assim como os anteriores, não possui o tratamento para a exceção, sendo obrigada a sair da pilha de execução, que será jogada no console.

Então, fica a questão: *será que, dentro desse método, existe algum código que consiga resolver a exceção* `ArithmeticException`? Já escrevemos algum código saiba resolver `ArithmeticException`? Não! Não aprendemos isso ainda.

Assim, o Java descarta o `metodo2()`, porque ele não consegue resolver. Agora, a "bomba" está em cima do `metodo1()`, mas assim como o anterior, esse método também não sabe resolver `ArithmeticException`. Mesmo sem finalizar, o Java também o descarta da pilha. Assim, a exceção chega ao método `main()`. No entanto, considerando que não existe um código para resolver o `ArithmeticException` ela é transferida para o console, no qual vimos o **rastro** dela.

Resumindo: sabemos que existem exceções e que elas mudam o fluxo. Se soluções não forem encontradas dentro da pilha de execução, elas serão impressas no console.

Adiante, veremos como lidar com essa "bomba", fazendo um *tratamento* das exceções.

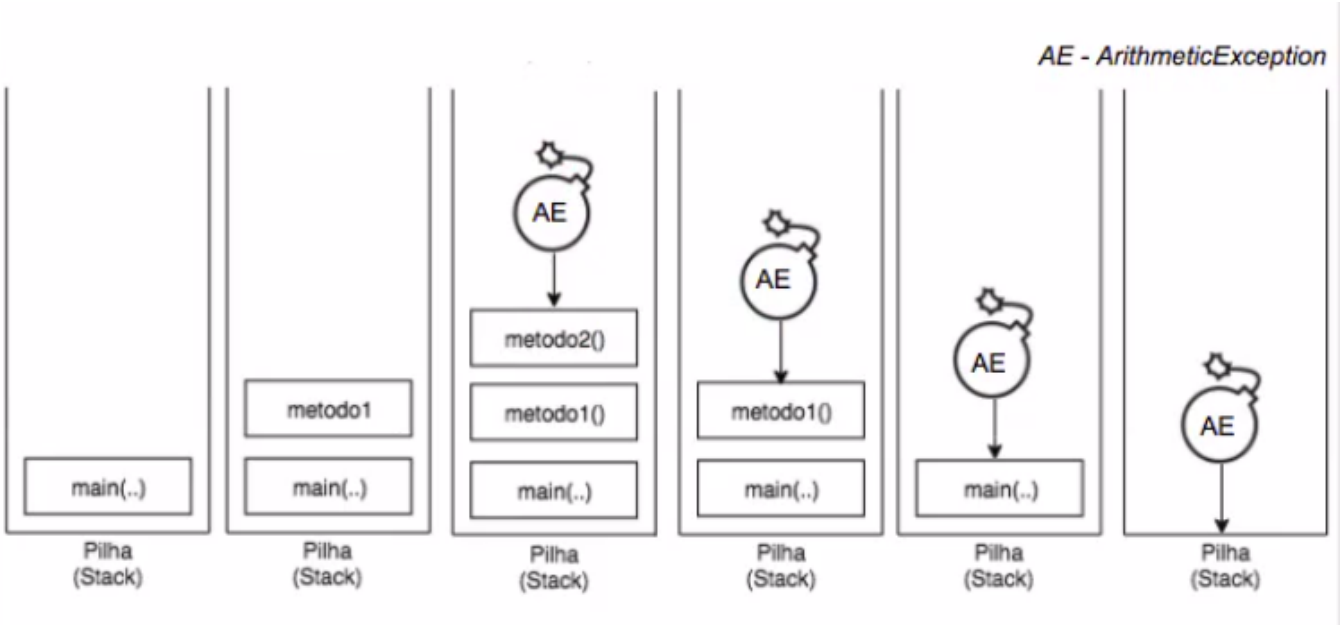
03

Try Catch

Transcrição

Continuaremos a nossa viagem por Java e suas exceções, nos aprofundando cada vez mais no assunto.

Anteriormente, concluímos que exceções possuem *nomes*, e elas acontecem porque algo deu errado ou inesperado, podendo até mudar o fluxo de execução da aplicação.



De acordo com o diagrama acima, a exceção pode ser vista como uma bomba em cima da pilha. Se o método não souber lidar com ela, ele será descartado e, assim, a exceção muda o fluxo de execução.

Para **tratarmos** uma exceção, é preciso sinalizar para a máquina virtual que isso pode acontecer, por meio de um código específico (`try`). Assim, ela entenderá que deve **tentar** executar esse código, entre chaves (`{}`) e com mais cautela.

```
private static void metodo2() {
    System.out.println("Ini do metodo2");
    for(int i = 1; i <= 5; i++) {
        System.out.println(i);
        try {
            int a = i / 0;
        }
    }
}
```

COPIAR CÓDIGO

Ainda dessa forma, o código nem compila, porque precisamos avisar para a máquina virtual qual exceção pode acontecer. Para isso, utilizaremos um novo bloco de código, por meio de `catch`, sinalizando que queremos **capturar** um problema. No caso, `ArithmeticException`.

```
try {
    int a = i / 0;
} catch (ArithmeticException ex) {

}
```

COPIAR CÓDIGO

Depois de colocar o nome da exceção, colocamos uma variável.

Dessa forma, o bloco de `try` sinaliza que o código `int a = i / 0` é perigoso e, em caso de exceção, a capturaremos e executaremos no bloco seguinte, por meio de `catch`. Dentro dele, imprimiremos o seguinte:

```
try {
    int a = i / 0;
} catch (ArithmeticException ex) {
```

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

COPIAR CÓDIGO

Salvamos e veremos no console qual será a saída com o `tryCatch`. Teremos o seguinte resultado:

```
Ini do main
Ini do metodo1
Ini do metodo2
1
ArithmeticException
2
ArithmeticException
3
ArithmeticException
4
ArithmeticException
5
ArithmeticException
Fim do metodo2
Fim do metodo1
Fim do main
```

COPIAR CÓDIGO

Note que agora temos o código que sabe como resolver a "bomba". A exceção foi capturada por `catch` e, no console, foi impresso `System.out.println("ArithmeticException")`. No entanto, logo depois apareceu o `2`, ou seja, a máquina virtual continuou com o laço e voltou para a execução normal. Isso é o que acontece em cada iteração. Jogamos cinco "bombas" em cima da pilha e, por meio de `catch`, voltamos à execução normal.

Vimos a sintaxe básica do `tryCatch`. Adiante, estudaremos outras variações. O importante é que já sabemos como resolver alguma exceção, utilizando `tryCa`

No entanto, não queremos mais que ele fique dentro do laço, considerando que ele **não precisa** estar lá. Vamos alterá-lo de lugar, transferindo-o na hora de chamar o `metodo2()`:

```
  private static void metodo1() {
      System.out.println("Ini do metodo1");
      try {
          metodo2();
      } catch(ArithmeticException ex) {
          System.out.println("ArithmeticException");
      }
      System.out.println("Fim do metodo1");
  }
```

COPIAR CÓDIGO

Com essas mudanças, é interessante pensar na saída. Vamos executar!

```
  Ini do main
  Ini do metodo1
  Ini do metodo2
  1
  ArithmeticException
  Fim do metodo1
  Fim do main
```

COPIAR CÓDIGO

Repare que houve erro, na primeira iteração do laço dentro do `metodo2()`. Temos algum código no `metodo2()` que saiba como resolver a "bomba" que foi jogada? Não! Então, o Java saiu abruptamente da linha `int a = i / 0` e voltou para a chamada do `metodo2()`, dentro do `tryCatch`. Repare que na saída **não** apareceu "Fim do metodo2", porque ele foi descartado. E então, voltamos para o `metodo1()`, no qual temos um código para resolver `ArithmeticException`.

Capturamos a exceção (a bomba) da pilha, e imprimimos "ArithmeticException" e, logo depois, voltou à execução normal, imprimindo "Fim do metodo1" e "Fim do main".

A mesma situação pode ser aplicada em `main` ! Veremos adiante! ;)

05

Variação do Catch

Transcrição

Sabemos que a pilha de execução é fundamental para executar as exceções, e que elas existem porque algo anormal pode acontecer em nosso código, podendo ser intencional ou não.

Anteriormente, estudamos sobre o tratamento de exceções por meio de `tryCatch`, solucionando a "bomba" que está em cima da pilha.

Agora, mostraremos como ficou `tryCatch` feito na `main()`, assim como foi dito anteriormente.

```
public static void main(String[] args) {
    System.out.println("Ini do main");
    try {
        metodo1();
    } catch(ArithmeticException ex) {
        System.out.println("ArithmeticException");
    }
    System.out.println("Fim do main");
}
```

COPIAR CÓDIGO

Temos o seguinte resultado:

```
Ini do main
Ini do metodo1
```

```
Ini do metodo2
1
ArithmeticException
Fim do main
```

COPIAR CÓDIGO

Note que "Fim do metodo1" e "Fim do metodo2" não aparecem, porque `main()` é a única que possui código capaz de tratar a exceção. Então é exibido "Fim do main" e é finalizado.

Sabemos que temos uma exceção do tipo `ArithmeticException`. A variável `ex` é uma referência e, com isso, podemos dizer que exceções também são objetos. Então, podemos usar a referência para chamar algum método público da classe. Pegaremos o método `getMessage()`, com o qual conseguiremos pegar a informação apresentada no console, por exemplo, a mensagem `/ by zero`. Depois de pegá-la, iremos guardá-la em uma String e mostrá-la após "ArithmeticException".

```
public static void main(String[] args) {
    System.out.println("Ini do main");
    try {
        metodo1();
    } catch(ArithmeticException ex) {
        String msg = ex.getMessage();
        System.out.println("ArithmeticException " + msg);
    }
    System.out.println("Fim do main");
}
```

COPIAR CÓDIGO

Executaremos o projeto para ver o resultado dessas alterações. Após a execução, no console teremos o seguinte:

```
Ini do main
Ini do metodo1
Ini do metodo2
1
ArithmeticException / by zero
Fim do main
```

COPIAR CÓDIGO

Há mais coisas que podemos fazer. Da mesma forma que a exceção se lembra da mensagem, ela também se lembra por onde passou e deixou seu rastro. Para mostrá-lo, usaremos o método `printStackTrace()` :

```
public static void main(String[] args) {
    System.out.println("Ini do main");
    try {
        metodo1();
    } catch(ArithmeticException ex) {
        //String msg = ex.getMessage();
        //System.out.println("ArithmeticException " + msg);
        ex.printStackTrace();
    }
    System.out.println("Fim do main");
}
```

COPIAR CÓDIGO

Salvaremos e executaremos, obtendo o seguinte retorno, no console:

```
Ini do main
Ini do metodo1
Ini do metodo2
1
java.lang.ArithmeticException: / by zero
        at Fluxo.metodo2(Fluxo.java:25)
        at Fluxo.metodo1(Fluxo.java:17)
```

```
        at Fluxo.main(Fluxo.java:6)
Fim do main
```

COPIAR CÓDIGO

Dessa forma, entendemos que `ex` é uma referência e o tipo da referência é o nome da classe da exceção. Parte do tratamento dela é saber trabalhar com a referência. Normalmente, não precisamos saber muito sobre `ex`. Basta sabermos que `getMessage()` é um método importante para descobrir a mensagem original e que o `printStackTrace()` pode ser útil também. Ainda nesse curso, criaremos as nossas próprias exceções!

Agora, criaremos uma outra situação para fins didáticos!

Antes de começar, será necessário criar a classe `Conta`, com o método `deposita()` vazio. Na classe `Fluxo`, colocaremos um comentário na linha que gera o erro de divisão por zero.

Logo abaixo da linha comentada, criaremos uma referência da classe `Conta` nula, ou seja, ela apontará para nenhum objeto. E chamaremos o método `deposita()` a partir dela.

```
private static void metodo2() {
    System.out.println("Ini do metodo2");
    for(int i = 1; i <= 5; i++) {
        System.out.println(i);
        //int a = i / 0;
        Conta c = null;
        c.deposita();
    }
    System.out.println("Fim do metodo2");
}
```

COPIAR CÓDIGO

Vamos executar o código. Já sabemos que vai dar erro, pois a referência `c` está nula. A saída dessa execução é a seguinte:

```
Ini do main
Ini do metodo1
Ini do metodo2
1
Exception in thread "main" java.lang.NullPointerException
            at Fluxo.metodo2(Fluxo.java:27)
            at Fluxo.metodo1(Fluxo.java:17)
            at Fluxo.main(Fluxo.java:6)
```

COPIAR CÓDIGO

Sabemos que existe um código em `main` capaz de resolver a exceção `ArithmeticException`. Entretanto, repare que não apareceu "Fim do main" na saída, como no exemplo anterior. Ou seja, não conseguimos tirar a bomba da pilha por meio de `catch()`. Isso aconteceu porque fizemos um **catch específico**, que funciona somente para `ArithmeticException`, e o nosso problema se chama `NullPointerException`.

Para resolver, é preciso mudar o nome da exceção dentro de `catch`. Mas, sabendo que o código pode criar outros tipos de exceções, além de `ArithmeticException`, podemos manter a exceção desse `catch` e criar um segundo para a exceção `NullPointerException`.

```
public static void main(String[] args) {
    System.out.println("Ini do main");
    try {
        metodo1();
    } catch(ArithmeticException ex) {
        //String msg = ex.getMessage();
        //System.out.println("ArithmeticException " + msg);
        ex.printStackTrace();
    } catch(NullPointerException ex) {
```

```
            String msg = ex.getMessage();
            System.out.println("NullPointerException " + msg);
        }
        System.out.println("Fim do main");
    }
```

COPIAR CÓDIGO

Qualquer exceção tem o método `getMessage()`.

Executaremos novamente para ver a diferença.

```
Ini do main
Ini do metodo1
Ini do metodo2
1
NullPointerException null
Fim do main
```

COPIAR CÓDIGO

A mensagem retornada pelo `getMessage()` foi *null*. Ou seja, o segundo bloco `catch` foi chamado. Então, podemos ter quantos blocos `catch` quisermos, desde que eles sejam específicos o suficiente.

A partir do Java `1.7`, chegou mais uma variação do `catch`. Em vez de repetir vários blocos de `catch`, podemos colocar um *pipe* (`|`), que significa "OU":

```
catch(ArithmeticException | NullPointerException ex)
```

COPIAR CÓDIGO

É uma facilidade que veio para não precisarmos repetir código. Utilizando-a, o bloco de código `main()` ficará da seguinte forma:

```
public static void main(String[] args) {
    System.out.println("Ini do main");
    try {
        metodo1();
    } catch(ArithmeticException | NullPointerException ex) {
        String msg = ex.getMessage();
        System.out.println("Exception " + msg);
        ex.printStackTrace();
    }
}
```

COPIAR CÓDIGO

O código acima é válido para as duas situações que trabalhamos. Adiante, falaremos mais sobre a classe `Exception`.

03

Lançando exceções

Transcrição

A partir de agora, criaremos as nossas próprias exceções. Anteriormente, vimos que elas podem ocorrer quando algo inesperado acontece no código, por exemplo, uma divisão por zero ou uma referência nula.

Mas, sabemos que essas exceções não deveriam acontecer. Deveríamos programar de forma correta, evitando-as. Todavia, existem casos em que as exceções nos ajudarão muito.

Na classe `Conta` do projeto criado anteriormente, temos o método `saca()`:

```
public boolean saca(double valor) {
    if(this.saldo >= valor) {
        this.saldo -= valor;
        return true;
    } else {
        return false;
    }
}
```

COPIAR CÓDIGO

O retorno do tipo *boolean* mostra se o saque deu certo ou não. Entretanto, há muitos outros motivos para que um saque não funcione. Saldo insuficiente, saques não permitidos aos domingos, saques não permitidos fora do horário comercial, entre outros.

O `return false` dessa função simplesmente diz que o saque não funcionou, sem especificar o motivo. Podemos pensar em uma forma mais fácil para descrever os motivos, trocando o tipo de retorno de *boolean* para *int*. Assim, podemos atribuir:

- número `1` se `saca()` funcionar;
- um valor negativo para especificar o motivo, se o `saca()` **não** funcionar. Por exemplo:
  - `-1` pode representar *domingo*;
  - `-2` pode representar *sábado*;
  - `-3` pode representar o *limite diário* e assim por diante.

Mas essa programação voltada ao primitivo não cheira muito bem. No mundo Java, exceções possuem *nomes*, são *objetos* e *classes*!

Para resolver o problema do `saca()`, as exceções podem ser uma boa solução. Voltando ao projeto `java-pilha`, criaremos uma cópia da classe `Fluxo` e vamos chamá-la de `FluxoComTratamento`.

Voltando à classe `Fluxo`, deixaremos `try-catch` em `main`, mas alteraremos o `metodo2()`. Apagaremos o laço e deixaremos somente os `System.out.println()`:

```
private static void metodo2() {
    System.out.println("Ini do metodo2");

    System.out.println("Fim do metodo2");
}
```

COPIAR CÓDIGO

De volta ao `try-catch`, usaremos a referência `ex` para chamar algum método que se comunique com o objeto. Se `ex` é uma referência, então `NullPointerException` é um tipo baseado em uma classe.

Criaremos um objeto da classe `ArithmeticException` e o guardaremos na referência `exception`:

```
private static void metodo2() {
    System.out.println("Ini do metodo2");

    ArithmeticException exception = new ArithmeticException();

    System.out.println("Fim do metodo2");
}
```

COPIAR CÓDIGO

Se criamos um objeto da classe `ArithmeticException`, também criamos uma exceção? Não! Nós apenas criamos o objeto, e ainda falta mais um passo para isso.

A referência `exception` aponta para a `ArithmeticException`, que está no **HEAP** (memória de objetos). Precisamos falar para o Java pegar esse objeto, transformar em uma exceção e "jogar" na pilha. O verbo "jogar", em inglês, é "*throw*"! Então, vamos "jogar" o objeto a partir da referência `exception`:

```
private static void metodo2() {
    System.out.println("Ini do metodo2");
    ArithmeticException exception = new ArithmeticException();
    throw exception ;
    System.out.println("Fim do metodo2");
}
```

COPIAR CÓDIGO

O Java reconhece que, quando jogamos uma exceção, saímos abruptamente do código. Se isso acontece, jamais será possível executar a linha que mostra "Fim do metodo2". Por isso, comentaremos essa linha do bloco, deixando-o da seguinte forma:

```
private static void metodo2() {
    System.out.println("Ini do metodo2");
    ArithmeticException exception = new ArithmeticException();
    throw exception ;
    //System.out.println("Fim do metodo2");
}
```

COPIAR CÓDIGO

E vamos executar, obtendo a seguinte saída no console:

```
Ini do main
Ini do metodo1
Ini do metodo2
Exception null
java.lang.ArithmeticException
        at Fluxo.metodo2(Fluxo.java:24)
        at Fluxo.metodo1(Fluxo.java:17)
        at Fluxo.main(Fluxo.java:6)
Fim do main
```

COPIAR CÓDIGO

Perceba que não foi impresso "Fim do metodo1" e, por esse motivo, sabemos que saímos abruptamente do método. Quando `main` recebe `ArithmeticException`, pega essa exceção e mostra a mensagem. Passaremos para o construtor a mensagem "deu errado".

```
private static void metodo2() {
    System.out.println("Ini do metodo2");
    ArithmeticException exception = new ArithmeticException("deu
    throw exception ;
    //System.out.println("Fim do metodo2");
}
```

COPIAR CÓDIGO

Legal! Será que conseguimos transformar qualquer objeto na pilha? Vamos criar um objeto do tipo `Conta`:

```
private static void metodo2() {
    System.out.println("Ini do metodo2");
    Conta c = new Conta();
    ArithmeticException exception = new ArithmeticException("deu
    throw c;
    //System.out.println("Fim do metodo2");
}
```

COPIAR CÓDIGO

Note que não foi possível fazer o `throw c` na pilha, pois só é possível com exceções. Vamos apagar a instância de `Conta`.

Normalmente, quando queremos jogar uma exceção, fazemos isso de maneira mais enxuta, sem guardar a referência.

```
private static void metodo2() {
    System.out.println("Ini do metodo2");
    throw new ArithmeticException("deu errado");
    //System.out.println("Fim do metodo2");
}
```

COPIAR CÓDIGO

Essa é a forma mais comum que encontraremos no dia a dia. A seguir, voltaremos ao método `saca()` para resolver o seu problema, no qual não faz sentido usar `ArithmeticException` ou `NullPointerException`.

01

Hierarquia de exceções

Transcrição

Começando deste ponto? Você pode fazer o DOWNLOAD (https://caelum-online-public.s3.amazonaws.com/834-java-excecoes/04/java4-aula4.zip) completo do projeto anterior e continuar seus estudos a partir daqui.

Anteriormente, comentamos sobre o problema do método `saca()` e que podemos criar a nossa própria exceção. Mas antes, é necessário aprendermos como as exceções se organizam internamente.

Com `throw`, jogamos o objeto da exceção na pilha, e isso só funciona com exceções. Mas, por quê? Vamos ver o que tem dentro da classe `ArithmeticException`., com "Ctrl" pressionado e clicando em cima do método, como se fosse um link.

```
public class ArithmeticException extends RuntimeException {
    private static final long serialVersionUID = 2256477558314496

    public ArithmeticException() {
        super();
    }

    public ArithmeticException(String s) {
        super(s);
    }
}
```

COPIAR CÓDIGO

Aqui temos um ótimo exemplo de *herança*, não é mesmo? Temos também dois construtores: um vazio e outro que recebe uma `String`, que representa a mensagem. O primeiro `super()` significa que estamos subindo a hierarquia para chamar o construtor da classe mãe. Já o segundo, repassa a mensagem do tipo `String` do construtor para o construtor da classe mãe.

Também podemos encontrar nessa classe, um atributo estático `serialVersionUID`, que representa uma identificação da classe, um valor único. Mas isso não faz diferença para nós.

Chamaremos a classe mãe da `ArithmeticException`, a `RuntimeException`! Da mesma forma, com o "Ctrl" pressionado, clicamos em cima da classe. Encontraremos o seguinte:

```
public class RuntimeException extends Exception {
    static final long serialVersionUID = -7034897190745766939L;

    public RuntimeException() {
        super();
    }

    public RuntimeException(String message) {
        super(message);
    }

    public RuntimeException(String message, Throwable cause) {
        super(message, cause);
    }

    public RuntimeException(Throwable cause) {
        super(cause);
    }
```

```
    protected RuntimeException(String message, Throwable cause, b
        super(message, cause, enableSuppression, writableStackTra
    }
}
```

COPIAR CÓDIGO

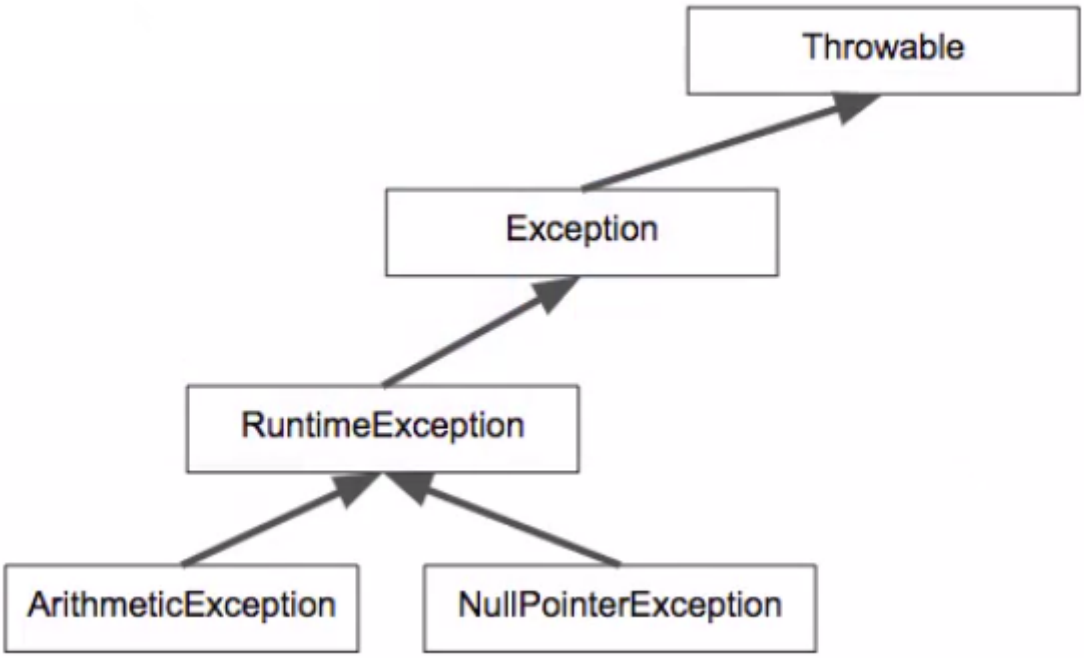
Repare que essa classe também possui uma classe mãe, a `Exception`, além de construtores. Voltando à classe `Fluxo`, vemos que não importa o tipo da referência `ex`, sendo `ArithmeticException` ou `NullPointerException`, nós conseguimos chamar o método `getMessage()`, mesmo que nenhum desses dois tipos tenha o método `getMessage()`.

A classe `RuntimeException` também não tem esse método. A classe `Exception` tem vários construtores, mas nenhum sinal desse método. Essa classe estende ou herda de `Throwable`. Usamos a palavra *throw* para jogar o objeto da exceção na pilha e ela não funciona para classes e objetos que não sejam exceções. Isso acontece porque a nossa classe `Conta` não estende `Throwable`. Para jogar um objeto na pilha, a classe precisa estender `Throwable`. Vamos clicar nela para ver o que ela tem.

A classe `Throwable` implementa a interface `Serializable` que se relaciona com o atributo `serialVersionUID`, mas esse não é o nosso foco.

Ainda dentro de `Throwable`, encontramos o atributo `detailMessage` que especifica detalhes, as mensagens. Entretanto a classe `Throwable` é enorme, e nem tudo o que ela possui nos interessa. Por essa razão, para saber os membros dessa classe, utilizamos o atalho "Ctrl + O".

Em uma janela mais enxuta, conseguimos então visualizar os seus membros, os métodos e atributos. E então, encontramos os métodos `getMessage()` e `printStackTrace()`. Ou seja, todos os métodos do mundo de exceções foram implementados na classe mãe `Throwable`.



Para acessibilidade: A hierarquia abordada possui uma classe mãe `Throwable` está no topo do diagrama e, dela, é formada uma hierarquia de exceções voltada para o desenvolvedor/desenvolvedora Java.

As classes `ArithmeticException` e `NullPointerException` herdam de `RuntimeException`, considerando que nessa classe só existem construtores, e ela herda de `Exception` que também só possui construtores. `Exception` é a última classe que herda de `Throwable`.

Na hierarquia, objetos que fazem parte de qualquer classe que estenda `Throwable`, podem ser jogados na pilha. Mas, por que a classe `ArithmeticException`, por exemplo, não estende diretamente a `Throwable`? Veremos isso adiante.

A partir do diagrama, concluímos que para criarmos a nossa própria exceção, basta nos colocarmos dentro dessa hierarquia. Para começar, criaremos `MinhaExcecao`, que estende a classe `RuntimeException`.

Clicaremos com o botão direito no `package` e selecionaremos "New > Class > MinhaExcecao". Vamos colocá-la na hierarquia:

```
public class MinhaExcecao extends RuntimeException {

}
```

COPIAR CÓDIGO

Tudo o que é necessário para ser uma exceção, já está aqui! Vamos usá-la. Na classe `Fluxo`, dentro do `metodo2()`, faremos o seguinte:

```
private static void metodo2() {
    System.out.println("Ini do metodo2");
    throw new MinhaExcecao("deu muito errado");

    //System.out.println("Fim do metodo2");
}
```

COPIAR CÓDIGO

Repare que ainda não criamos nenhum construtor dentro de `MinhaExcecao` que recebe uma string. Entretanto, a exceção já funciona se tirarmos a string, em função da hierarquia. Então, vamos criar um construtor para receber uma mensagem.

```
public class MinhaExcecao extends RuntimeException {
    public MinhaExcecao(String msg) {

    }
}
```

COPIAR CÓDIGO

Agora, a classe `Fluxo` já compila! Mas perderemos a `msg` do construtor em `MinhaExcecao`. Daremos uma olhada na classe `RuntimeException`. Ela possui o

construtor que recebe uma `message`. E note que ele chama o `super` e manda a mensagem para a classe mãe. Vamos fazer isso também! É uma boa prática para não perder a mensagem.

```
public class MinhaExcecao extends RuntimeException {
    public MinhaExcecao(String msg) {
        super(msg);
    }
}
```

COPIAR CÓDIGO

Voltando ao `Fluxo`, repare que a linha `throw new MinhaExcecao("deu muito errado")` está compilando novamente. Compilaremos essa classe para ver a exceção acontecendo.

```
Ini do main
Ini do metodo1
Ini do metodo2
Exception in thread "main" MinhaExcecao: deu muito errado
        at Fluxo.metodo2(Fluxo.java:23)
        at Fluxo.metodo1(Fluxo.java:17)
        at Fluxo.main(Fluxo.java:6)
```

COPIAR CÓDIGO

Legal! Temos o rastro da pilha e o nome da exceção iguais a `ArithmeticException` e `NullPointerException`. Mas uma coisa mudou. Repare que **não** apareceu "Fim do main". Essa linha não foi executada porque não capturamos a exceção!

Em `catch`, só estamos capturando as exceções `ArithmeticException` e `NullPointerException`, e não `MinhaExcecao`. Então, é preciso adicionar mais um *pipe* (`|`) para conseguirmos colocar mais um tipo de exceção:

```
public static void main(String[] args) {
    System.out.println("Ini do main");
    try {
        metodo1();
    } catch(ArithmeticException | NullPointerException | MinhaExc
        String msg = ex.getMessage();
        System.out.println("Exception " + msg);
        ex.printStackTrace();
    }
    System.out.println("Fim do main");
}
```

COPIAR CÓDIGO

Vamos testar? Antes, lembre-se de salvar e executar novamente!

```
Ini do main
Ini do metodo1
Ini do metodo2
Exception deu muito errado;
MinhaExcecao: deu muito errado
        at Fluxo.metodo2(Fluxo.java:23)
        at Fluxo.metodo1(Fluxo.java:17)
        at Fluxo.main(Fluxo.java:6)
Fim do main
```

COPIAR CÓDIGO

Agora, já sabemos como criar exceções. Em nosso projeto do Banco, daremos um nome melhor para deixar claro o que pode acontecer. Mas, antes de voltar realmente para resolver o problema da `Conta`, temos que aprender mais um pouco sobre a hierarquia e a diferença entre as classes `RuntimeException` e `Exception`.

03

Entendendo erros

Transcrição

Anteriormente, estudamos hierarquia e vimos que é a classe do topo — `Throwable` — quem realmente faz as coisas. `Exception` e `RuntimeException` não possuem utilidade e cada uma só possui alguns construtores. Mas mesmo assim, nós estendemos `RuntimeException`, assim como `ArithmeticException` e `NullPointerException`.

Mas, não seria mais fácil cortar caminho e estender diretamente o `Throwable`? Considerando que `RuntimeException` e `Exception` não possuem utilidade? Errado! Com certeza existe um porquê disso.

Existe uma outra hierarquia de classes que estende `Throwable`, como a classe `Error`. Entretanto, nós não a conhecemos muito bem, porque é uma hierarquia pensada para desenvolvedores de máquina virtual. Nós, desenvolvedores Java, não trabalhamos com essas classes diretamente. Quem cria e joga esses objetos na pilha é a máquina virtual, quando algo muito grave acontece.

A máquina virtual pode não conseguir mais executar o código porque não tem mais memória o suficiente, e então ela jogará um erro internamente. Vamos simular esse erro?

Voltando ao código, faremos uma cópia da classe `Fluxo` e vamos chamá-la de `FluxoComError`. Dentro dessa classe, apagaremos todo o corpo do `metodo2()` e o chamaremos, dentro dele mesmo! Assim:

```
private static void metodo2() {
    System.out.println("ini do metodo 2");
    metodo2();
    System.out.println("fim do metodo 2");
}
```

COPIAR CÓDIGO

Quando o Java entrar no `metodo2()`, imprimirá a mensagem "ini do metodo 2" e vai chamar de novo. A pilha irá crescer, pois o `metodo2()` ficará se chamando.

Ao executarmos esse código, vemos várias linhas de "ini do metodo 2", porque o método fica chamando ele mesmo. Essa ação se repete até não ter mais espaço na pilha. Nessa execução, temos um dos erros mais famosos, o **StackOverflowError**.

Aprendemos que existe uma hierarquia de classes utilizada internamente pela máquina virtual. Mas ainda temos as seguintes questões:

- *Qual a diferença entre* `Exception` *e* `RuntimeException`*?*
- *Por que não podemos eliminá-las?*

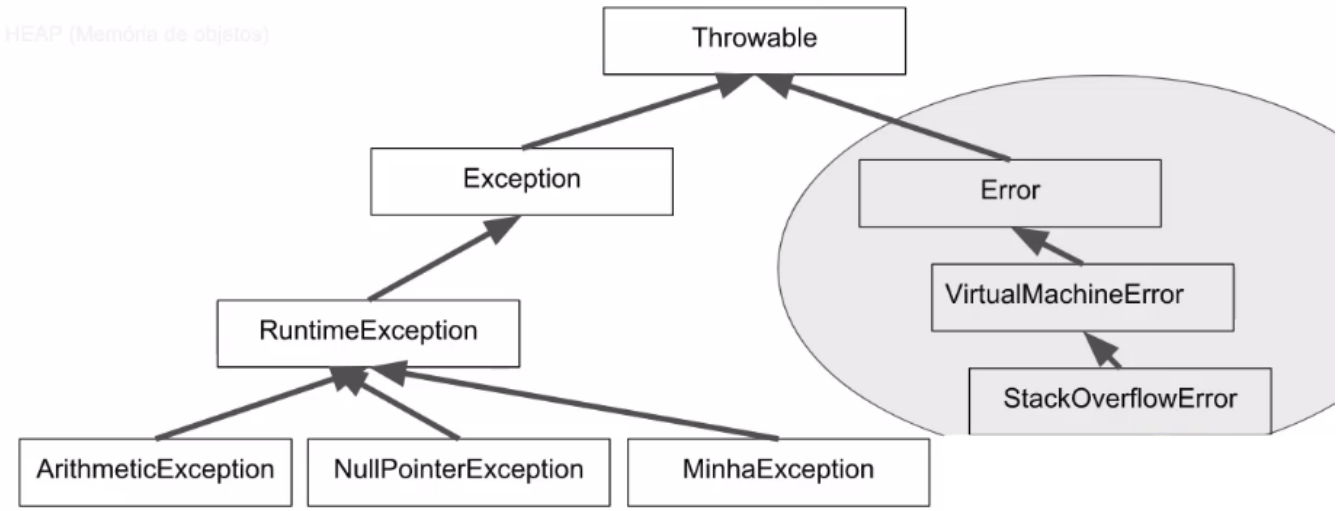
Veremos as respostas, a seguir!

04

Checked e unchecked

Transcrição

Anteriormente, falamos sobre a hierarquia dos erros.



Para acessibilidade: A hierarquia abordada contém uma classe mãe chamada `Throwable`. A partir dela, formam-se duas categorias: uma para exceções onde o desenvolvedor pode gerenciar e uma outra categoria voltada para erros da máquina virtual.

As exceções `ArithmeticException`, `NullPointerException` e `MinhaException` herdam de `RuntimeException`, que por sua vez, herda de `Exception` e `Exception` herda de `Throwable`, formando a primeira categoria de exceções. Já a segunda categoria de erros possui `StackOverflowError`, que herda de `VirtualMachineError`, que herda de `Error`, que herda de `Throwable`, formando a segunda categoria.

Lembrando que a segunda categoria é a hierarquia utilizada pela *máquina virtual*. Entretanto, o que nos interessa é a hierarquia da `Exception`, a primeira categoria. E por que a classe `ArithmeticException` não estende diretamente da classe `Exception`? Por que a `MinhaExcecao` não estende a classe `Exception`? Vamos tentar resolver esse enigma.

Acessaremos a classe `MinhaExcecao`, e estenderemos diretamente da classe `Exception`. Repare que o código já para de compilar. Apareceu algum problema. Vamos ver a classe `Fluxo`.

Existe um erro de compilação na linha do `throw new MinhaExcecao("deu muito errado")`. Se retirarmos a palavra `throw` dessa frase, o problema desaparecerá. O problema é que o Java faz uma separação. Duas categorias de exceções são criadas dentro das exceções para o desenvolvedor de aplicações.

A primeira categoria é a que estende de `RuntimeException`, e a outra é a que estende diretamente de `Exception`. O compilador faz uma verificação sintática para ver quem dá `throw` nessas exceções. Isso significa que a exceção exige que fique explícito na assinatura do método que estamos *jogando* a exceção:

```
private static void metodo2() throws MinhaExcecao {
    System.out.println("Ini do metodo2");
    throw new MinhaExcecao("deu muito errado");

    //System.out.println("Fim do metodo2");
}
```

COPIAR CÓDIGO

Dessa forma, dizemos que o método joga uma exceção do tipo `MinhaExcecao`. Agora, o método volta a compilar. Quando usamos `throw new` de uma exceção que estende *diretamente* da classe `Exception`, o compilador exige que coloquemos, explicitamente, *throws* na assinatura do método.

A ideia é que o "perigo" — a exceção — fique explícita na assinatura do método. Com isso, temos duas categorias novas de `Exception`. A primeira é a que estende de `RuntimeException` e se chama ***Unchecked***. A segunda categoria é a que estende diretamente de `Exception`, chamada de ***Checked***.

Por que Checked e Unchecked?

Na categoria *Unchecked*, o compilador não dá muita importância. Se dermos `throws` ou não, ele não toma atitude, ou seja, ele não **verifica** — *unchecked* (não verificado pelo compilador).

Já a categoria *Checked* é verificada pelo compilador. No `metodo2()`, somos obrigados a colocar `throws` na assinatura do método, pois a exceção do tipo `MinhaExcecao` estende diretamente de `Exception` e, por isso, é verificada pelo compilador.

Legal! Mas repare que a chamada do `metodo2()`, no `metodo1()`, não compila mais. Agora, isso acontece porque o compilador detecta que existe uma exceção *checked* na assinatura e, por isso, é necessário também deixar explícito o `throws MinhaExcecao` na assinatura do `metodo1()`.

```
private static void metodo1() throws MinhaExcecao {}
```

COPIAR CÓDIGO

Podemos colocar `throws` no método ou transferir a exceção para a categoria *unchecked*. Repare que colocamos `throws` nos dois métodos, mas não o colocamos em `main()` e, mesmo assim, o código compilou. Isso aconteceu porque estamos fazendo um tratamento da exceção com o `try-catch`. Ou seja, temos duas formas de resolver uma exceção *checked*.

Ou colocamos o `throws` na assinatura, ou fazemos um `try-catch`. Por exemp

```
private static void metodo1() {
    System.out.println("Ini do metodo1");
    try {
        metodo2();
    } catch(MinhaExcecao ex) {}
    System.out.println("Fim do metodo1");
}
```

COPIAR CÓDIGO

Assim, o compilador não se manifestou, pois estamos resolvendo o problema da exceção. Se estamos resolvendo, não é necessário colocar o `throws` na assinatura do método. Considerando que foi só um exemplo, vamos deixar como estava, pois já temos um `try-catch` em `main()`.

O conceito de exceções existe em várias linguagens, mas o que acabamos de estudar — *checked* e o *unchecked* — é algo específico do mundo Java.

Entendemos que todas as categorias são exceções, são como bombas que caem na pilha e mudam o fluxo na hora da execução, mas a diferença entre eles é na hora de compilar. O *Checked* e o *Unchecked* estão relacionados ao processo de compilação.

09

O que aprendemos?

Se você fez o exercício [Será que o Miguel entendeu a aula? (https://cursos.alura.com.br/course/java-excecoes/task/37907)](), vai lembrar o que aprendemos. Para fixar ainda mais, listamos os tópicos dessa aula:

- Existe uma hierarquia grande de classes que representam exceções. Por exemplo, `ArithmeticException` é filha de `RuntimeException`, que herda de `Exception`, que por sua vez é filha da classe mais ancestral das exceções, `Throwable`. Conhecer bem essa hierarquia significa saber utilizar exceções em sua aplicação.
- `Throwable` é a classe que precisa ser extendida para que seja possível jogar um objeto na pilha (através da palavra reservada `throw`)
- É na classe `Throwable` que temos praticamente todo o código relacionada às exceções, inclusive `getMessage()` e `printStackTrace()`. Todo o resto da hierarquia apenas possui algumas sobrecargas de construtores para comunicar mensagens específicas
- A hierarquia iniciada com a classe `Throwable` é dividida em **exceções** e **erros**. Exceções são usadas em códigos de aplicação. Erros são usados exclusivamente pela máquina virtual.
- Classes que herdam de `Error` são usadas para comunicar erros na máquina virtual. Desenvolvedores de aplicação não devem criar erros que herdam de `Error`.
- `StackOverflowError` é um erro da máquina virtual para informar que a pilha de execução não tem mais memória.
- Exceções são separadas em duas grandes categorias: aquelas que são obrigatoriamente verificadas pelo compilador e as que não são verificadas.
- As primeiras são denominadas *checked* e são criadas através do pertencimento a uma hieraquia que não passe por `RuntimeException`.

- As segundas são as *unchecked*, e são criadas como descendentes de `RuntimeException .`

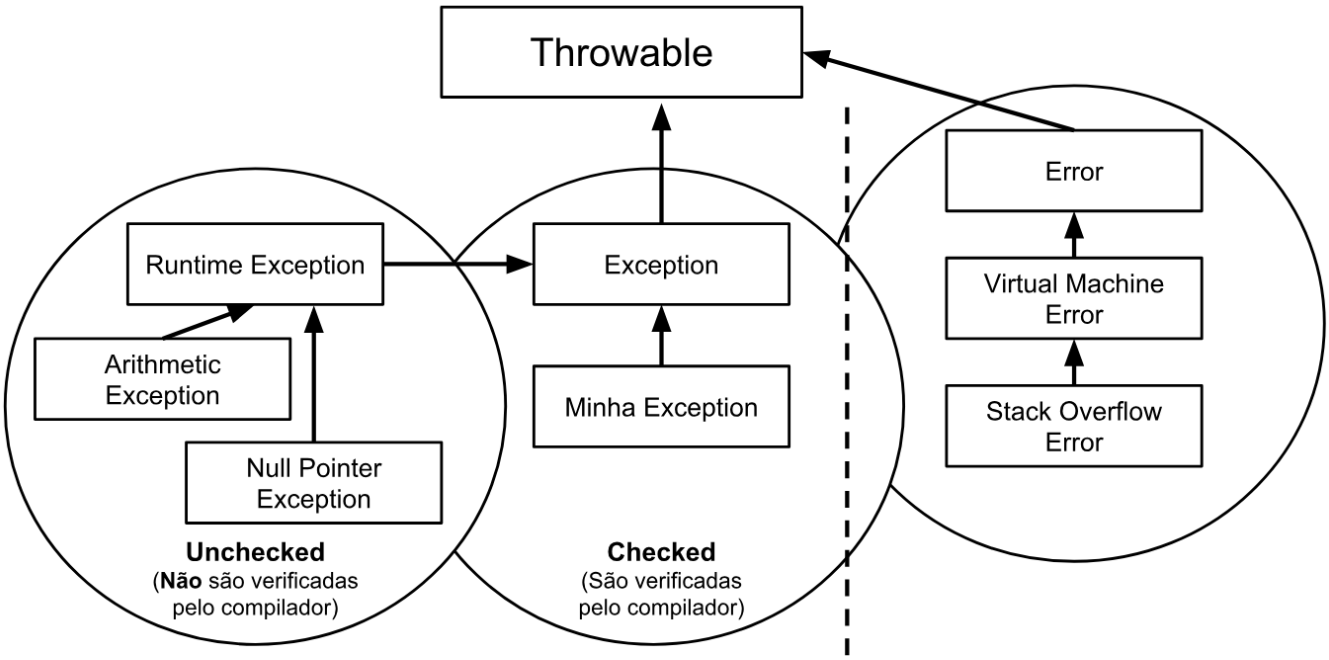
01

Capturando qualquer exceção

Transcrição

Começando deste ponto? Você pode fazer o DOWNLOAD (https://caelum-online-public.s3.amazonaws.com/834-java-excecoes/05/java4-aula5.zip) completo do projeto anterior e continuar seus estudos a partir daqui.

Aqui está o nosso diagrama, um pouco melhorado:



Na hora de rodar (executar) **não** tem diferença!!!

À direita, temos erros que são da máquina virtual, nos quais não mexeremos, mas é importante saber que existem. À esquerda, temos as exceções.

No mundo Java, não é correto falar sobre erros com desenvolvedores, considerando que trabalhamos com *exceções*, e os erros são da máquina virtual.

O que nos interessa é o lado esquerdo do diagrama, referente às exceções. Vimos que existe diferença entre as exceções *Checked* e *Unchecked*, dependendo de qual classe estendemos. Se estendemos de `RuntimeException`, temos uma exceção do tipo *Unchecked*, ou seja, o compilador não toma atitude. Tendo `throw` ou não, ele não se importa.

Se estendermos diretamente da classe `Exception`, o compilador ficará de olho e nos obrigará a colocar `throws` na assinatura do método, para sinalizar quem chama o método, que ele é perigoso ou tratar a exceção no próprio método com o `try-catch`. Essa exceção é do tipo *checked*.

> Na hora de executar o código, não tem diferença! Todos são como bombas que caem na pilha.

##Mas, para quê serve o Checked e o Unchecked?##

Essa é uma pergunta difícil, pois o significado de Checked e Unchecked para o nosso código, mudou durante a vida do Java.

A polêmica das exceções está relacionada ao *Checked*. Hoje, existem aplicações que simplesmente não usam exceções desse tipo, e é muito comum utilizar bibliotecas que só têm exceções *Unchecked*, nas quais o compilador não nos obriga a tomar alguma atitude.

Então, para que serve *checked*, considerando que, atualmente, a maioria utiliza *Unchecked*? Mostraremos na classe `Conta`.

```
public class Conta {
    void deposita() {

    }
}
```

COPIAR CÓDIGO

Na assinatura do método `deposita()`, colocaremos `throws` e a exceção criada. Lembrando que a exceção que criamos `MinhaExcecao()` é do tipo *Checked*.

```
public class Conta {
    void deposita() throws MinhaExcecao{

    }
}
```

COPIAR CÓDIGO

Repare que não é necessário fazer alterações no método, e ele já compilou. Vamos criar uma nova classe, instanciar a `Conta` e chamar o método.

```
public class TestaContaComExcexaoChecked {
    public static void main(String[] args) {
        Conta c = new Conta();
        c.deposita();
    }
}
```

COPIAR CÓDIGO

Como o método `deposita()` é vazio, nenhuma exceção será criada. Mas, basta colocar o `throws MinhaExcecao` na assinatura e o compilador saberá que o método tem uma exceção do tipo *Checked* na sua assinatura. Por isso, temos que tomar uma atitude: ou colocamos `throws MinhaExcecao` na assinatura de `main()`, ou utilizamos `try-catch`. Se escolhermos a segunda opção, obteremos o seguinte:

```
public class TestaContaComExcexaoChecked {
    public static void main(String[] args) {
        Conta c = new Conta();
        try {
            c.deposita();
        } catch(MinhaExcecao ex) {
```

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

COPIAR CÓDIGO

Agora, esse código compila! Mas pelo fato de `deposita()` jogar uma exceção *checked*, não tem como fugir! Quem chama o método `deposita()` sabe que ele pode ser perigoso e que pode ocasionar uma exceção. Isso não acontece com exceções *unchecked*.

Assim, entendemos essas duas categorias de exceções. Se você gosta de avisar aquele desenvolvedor que usará a sua classe, para que ele faça um tratamento, pois algumas exceções podem ocorrer, use *checked*. Se você acha que não precisa disso e que o desenvolvedor pode fazer o tratamento quando ele achar melhor, use *unchecked*.

Esse é um belo início para estar ciente da diferença entre o *checked* e o *unchecked*.

Voltaremos à classe `FluxoComErro`, pois ela possui um erro. Ele está dessa forma:

```
public static void main(String[] args) {
    System.out.println("Ini do main");
    try {
        metodo1();
    } catch(ArithmeticException | NullPointerException | MinhaExc
        String msg = ex.getMessage();
        System.out.println("Exception " + msg);
        ex.printStackTrace();
    }
    System.out.println("Fim do main");
}

private static void metodo1() {
```

```
        System.out.println("Ini do metodo1");
        metodo2();
        System.out.println("Fim do metodo1");
    }
```

COPIAR CÓDIGO

O `metodo1()` é chamado dentro de `try`, mas ele não tem `throws MinhaExcecao` na sua assinatura. Está claro para o compilador que, se é verificada, essa exceção não pode acontecer nesse código. Considerando ser impossível que ela aconteça, temos que tirá-la de `catch`.

Por último, na classe `Fluxo`, o `metodo1()` possui o `throws` em sua assinatura e, por esse motivo, o compilador nos obriga a colocar a exceção em `catch`, sem reclamar. Inclusive, nesse `catch`, temos três exceções diferentes. Será que existe alguma forma de fazer um `catch` mais genérico, que funcione para qualquer exceção? **Sim**, existe!

O que todas as exceções do mundo Java para o desenvolvedor têm em comum, é que todas estendem a classe **Exception**. Por isso, podemos fazer um *catch polimórfico*. Em vez de definir cada exceção específica, aumentando cada vez mais o `catch`, podemos fazer assim:

```
    public static void main(String[] args) {
        System.out.println("Ini do main");
        try {
            metodo1();
        } catch(Exception ex) {}
```

COPIAR CÓDIGO

Dessa forma, **qualquer** exceção que possa acontecer será capturada. O tópico atual é bem difícil, e ainda temos assuntos para estudar, mas a base mais sólida e conceitual para trabalhar com exceções foi criada.

Agora sim, podemos voltar à classe `Conta` do nosso projeto e arrumar o método `saca()`.

03

Qual catch?

Já vimos nesse curso duas formas de capturar várias exceções através do bloco `catch`. (1) A forma tradicional, que funciona desde inicio do Java, simplesmente repete o bloco `catch` para cada exceção:

```
try {
 metodoPerigosoQuePodeLancarVariasExcecoes();
} catch(MinhaExcecao ex) {
 ex.printStackTrace();
} catch(OutraExcecao ex) {
 ex.printStackTrace();
}
```

COPIAR CÓDIGO

E (2) a forma mais atual, que foi introduzido no Java 7, permite definir as várias exceções no mesmo bloco (*multi catch*):

```
try {
 metodoPerigosoQuePodeLancarVariasExcecoes();
} catch(MinhaExcecao | OutraExcecao ex) { //multi-catch
 ex.printStackTrace();
}
```

COPIAR CÓDIGO

Você vai encontrar ambas as formas no seu dia a dia de desenvolvedor Java. Agora, veja **assinatura** do "método perigoso" em questão:

```
//funciona, podemos colocar duas exceções na assinatura
public void metodoPerigosoQuePodeLancarVariasExcecoes()
```

```
    //código omitido
}
```

COPIAR CÓDIGO

Vimos mais uma variação do *catch*, não sintática, mas conceitual. Qual afirmação abaixo pode ser usada para capturar todas exceções desse "método perigoso"?

Importante: Assumindo que ambas as exceções são do tipo *checked!*

A

```
try {
    metodoPerigosoQuePodeLancarVariasExcecoes();
}
```

B

```
try {
    metodoPerigosoQuePodeLancarVariasExcecoes();
} catch(RuntimeException ex) {
    ex.printStackTrace();
}
```

```
try {
    metodoPerigosoQuePodeLancarVariasExcecoes();
} catch(Exception ex) {
    ex.printStackTrace();
}
```

Correto. Criamos um `catch` genérico que captura qualquer exceção, incluindo exceções *checked*.
Isso pode parecer uma boa prática, mas normalmente não é. Como regra geral, sempre tente ser mais especifico possível no bloco `catch` favorecendo vários blocos *catch* ou usando *multi-catch*.

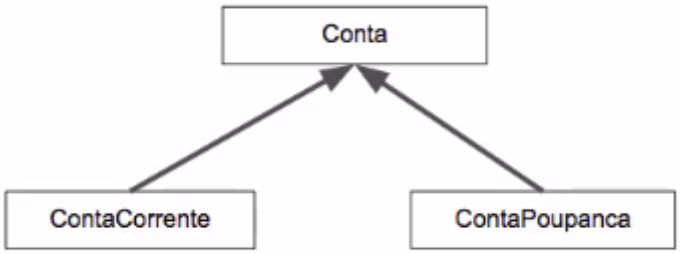
PRÓXIMA ATIVIDADE

04

Sacando com Unchecked Exception

Transcrição

Vamos retornar ao projeto final do curso anterior `bytebank-herdado-conta`, no qual temos a seguinte hierarquia:



A classe mãe é a `Conta`, e as contas que herdam ou estendem da classe mãe são as classes filhas `ContaCorrente` e `ContaPoupanca`. Temos interesse em mexer no método `saca()`, da classe mãe `Conta`. No entanto, o sobrescrevemos na classe `ContaCorrente` e, talvez, você tenha sobrescrito também na classe `ContaPoupanca`.

Ou seja, se alterarmos a assinatura de `saca()` na classe mãe, temos que ter cuidado para alterá-lo na classe filha. Não se esqueça que a *herança* é um relacionamento forte, e que mudanças na classe mãe afetarão as filhas.

Sabendo disso, vamos refatorar o nosso código. Na classe `Conta`, não queremos mais que o método `saca()` devolva um *boolean*. Se funcionar, OK! Se não funcionar, jogaremos uma exceção. Primeiro, esse método retornará `void`.

```
public void saca(double valor) {
    if(this.saldo >= valor) {
```

```
            this.saldo -= valor;
            return true;
        } else {
            return false;
        }
    }
```

COPIAR CÓDIGO

Com essa simples mudança de tipo de retorno, várias partes do código pararam de compilar, inclusive no método `transfere()` que usa o `saca()`. Para interromper o erro, voltaremos como estava, e criaremos a nossa própria exceção.

Com o botão direito no pacote, clicaremos em "New > Class". Para que essa classe, que chamaremos de `SaldoInsuficienteException`, represente uma exceção, é necessário colocá-la na hierarquia, utilizando a herança.

```
public class SaldoInsuficienteException extends RuntimeException

}
```

COPIAR CÓDIGO

Como já aprendemos, criaremos um construtor e passaremos uma mensagem como parâmetro. Depois, chamaremos o construtor da classe mãe e passaremos a mensagem.

```
public class SaldoInsuficienteException extends RuntimeException
    public SaldoInsuficienteException(String msg) {
        super(msg);
    }
}
```

COPIAR CÓDIGO

Pronto! Essa é a nossa exceção. Considerando que ela está pronta, podemos voltar à classe `Conta`, que não irá mais devolver `boolean`, mas sim `void`. Ou seja, não teremos mais retorno.

```
public void saca(double valor) {
    if(this.saldo >= valor) {
        this.saldo -= valor;
    }

}
```

COPIAR CÓDIGO

O trabalho com exceções é feito no início do método. Primeiro, testamos se é possível fazer a operação. Se sim, ok. Se não, já lançamos a exceção. Por isso, inverteremos a ordem da regra de negócio.

Se o `saldo` for menor do que `valor` ou insuficiente, teremos um problema. Caso esse não seja a situação, será possível sacar. Dê uma olhada no código abaixo:

```
public void saca(double valor) {
    if(this.saldo < valor) {
        //problema
    }
    this.saldo -= valor;
}
```

COPIAR CÓDIGO

Esse é um cenário muito comum! `If` no início do método para verificar se há problemas. Agora é a hora de jogar a exceção, utilizando `throw new`:

```
public void saca(double valor) {
    if(this.saldo < valor) {
        throw new SaldoInsuficienteException("");
```

```
        }
        this.saldo -= valor;
    }
```

COPIAR CÓDIGO

Para saber qual foi o saldo, passamos `Saldo: + this.saldo` como parâmetro da exceção e também o valor sacado.

```
throw new SaldoInsuficienteException("Saldo: " + this.saldo + ",
```

COPIAR CÓDIGO

O método `saca()` já está legal! No entanto, temos alguns erros de compilação tanto na classe `Conta`, quanto na classe `ContaCorrente`. Como vimos, o método `transfere()` usa o `saca()`, que devolvia um booleano; e o retorno era colocado em `if`. Entretanto, isso podia ser feito porque `if` recebe algo que é booleano. Agora o método `saca()` não retorna. Portanto, essa chamada não funcionará mais.

Por isso, apagaremos `if`. Se `saca()` não funcionar, será jogado como uma bomba na pilha, sairá abruptamente do método e cairá no método `transfere()`. Se nada resolver essa bomba, ela também sairá abruptamente de `transfere()`.

```
    public void transfere(double valor, Conta destino) {
        this.saca(valor);
        destino.deposita(valor);
    }
```

COPIAR CÓDIGO

Então, só será possível depositar o dinheiro, se `saca()` funcionar. Caso não funcione, sairá abruptamente desse método. Com isso, a classe `Conta` já estará funcionando!

Agora, daremos uma olhada na `ContaCorrente`. Nessa classe, o compilador verifica se estamos sobrescrevendo o método. Vamos arrumar:

```
@Override
public void saca(double valor) {
    double valorASacar = valor + 0.2;
    super.saca(valorASacar);
}
```

COPIAR CÓDIGO

Aparentemente, está compilando! Criaremos uma classe de teste `TesteSaca`. Depois, vamos criar a conta e depositar um valor para sacar.

```
public class TesteSaca {
    public static void main(String[] args) {
        Conta conta = new ContaCorrente(123, 321);

        conta.deposita(200.0);
        conta.saca(200.0);

        System.out.println(conta.getSaldo());
    }
}
```

COPIAR CÓDIGO

Nesse caso, não deve acontecer nenhuma exceção. Mas, dê uma olhada no console, após ter executado o teste:

```
Exception in thread "main" SaldoInsuficienteException: Saldo: 200
        at Conta.saca(Conta.java:25)
        at ContaCorrente.saca(ContaCorrente.java:13)
        at TesteSaca.main(TesteSaca.java:8)
```

COPIAR CÓDIGO

Essa exceção aconteceu porque o método `saca()` adiciona um valor, e então não tem como sacar. Vamos alterar o valor do saque para `190.0` e testar novamente. O que temos no console?

```
9.8000000000011
```

COPIAR CÓDIGO

Agora, vamos sacar um valor bem maior que depositamos.

```
Exception in thread "main" SaldoInsuficienteException: Saldo: 200
        at Conta.saca(Conta.java:25)
        at ContaCorrente.saca(ContaCorrente.java:13)
        at TesteSaca.main(TesteSaca.java:8)
```

COPIAR CÓDIGO

Essa exceção começou no método `saca()` da `Conta`. Depois, passou pelo método `saca()` da classe filha `ContaCorrente`, que também não foi resolvido. Por fim, passou pelo método `main()`, no qual não foi resolvido.

Agora, nos resta fazer o tratamento usando `try-catch` e usar uma exceção do tipo *checked*, pois do jeito que está agora, o compilador não nos obriga a fazer um tratamento.

06

Sacando com Checked Exception

Transcrição

Anteriormente, criamos a nossa própria exceção `SaldoInsuficienteException`, do tipo *Unchecked*. Ou seja, ela estende a classe `RuntimeException` e o compilador não nos obriga a fazer nenhum tratamento.

Testaremos também a mesma exceção como *Checked*. Ela estenderá diretamente da classe `Exception` e verá o tratamento. Mas, como faremos esse tratamento?

Repare que, do jeito que a exceção está agora, o compilador não reclama por não ter um tratamento, visto que ela é *unchecked*, mas não tem problema se criarmos um `try-catch`:

```
public static void main(String[] args) {
    Conta conta = new ContaCorrente(123, 321);

    conta.deposita(200.0);
    try {
        conta.saca(210.0);
    } catch(SaldoInsuficienteException ex) {
        System.out.println("Ex: " + ex.getMessage());
    }
    System.out.println(conta.getSaldo());
}
```

COPIAR CÓDIGO

Sem o tratamento, a saída no console era assim:

```
Exception in thread "main" SaldoInsuficienteException
        at Conta.saca(Conta.java:25)
        at ContaCorrente.saca(ContaCorrente.java:17)
        at TesteSaca.main(TesteSaca.java:8)
```

COPIAR CÓDIGO

E com o tratamento, temos a seguinte saída:

```
Ex: Saldo: 200.0, Valor: 210.2
200.0
```

COPIAR CÓDIGO

Já que deu erro, no final foi impresso o valor do saldo de `200.0`. Agora, vamos mudar a classe para o tipo *checked*.

```
public class SaldoInsuficienteException extends Exception {

}
```

COPIAR CÓDIGO

Depois que salvarmos essa alteração, aparecerá um problema no método `saca()`, da classe `Conta`, na qual alguém joga `SaldoInsuficienteException`. Levando isso em consideração, precisamos deixar claro na assinatura do método.

```
public void saca(double valor) throws SaldoInsuficienteException
    if(this.saldo < valor) {
        throw new SaldoInsuficienteException("Saldo: " + this.sal
    }
    this.saldo -= valor;
}
```

COPIAR CÓDIGO

Ao salvar, o compilador passa a reclamar no método logo abaixo `transfere()` :

```
public void transfere(double valor, Conta destino) {
    this.saca(valor);
    destino.deposita(valor);
}
```

COPIAR CÓDIGO

O `transfere()` chama o `saca()` , ou seja, se `saca()` é perigoso, `transfere()` precisa tomar uma atitude. Escolhemos a opção que deixa claro na assinatura do método que a exceção pode acontecer.

```
public void transfere(double valor, Conta destino) throws SaldoIn
    this.saca(valor);
    destino.deposita(valor);
}
```

COPIAR CÓDIGO

Certo. Agora, podemos ver que o erro está na classe `ContaCorrente` . O método `saca()` dessa classe chama o `super.saca()` que no caso, possui o `throws` na assinatura do método dizendo que ele é perigoso. Por causa disso, temos que tomar uma atitude: fazer um `try-catch` ou deixar o `throws` explícito na assinatura.

Dentro da nossa `ContaCorrente` , não faremos o `try-catch` então, ficará assim:

```
@Override
public void saca(double valor) throws SaldoInsuficienteException{
    double valorASacar = valor + 0.2;
    super.saca(valorASacar);
}
```

COPIAR CÓDIGO

Veja que uma exceção *checked* dá trabalho, pois o compilador fica verificando todas as classes que chamam o método perigoso. A classe `TesteContas` também está com problemas. Mas, como não estávamos utilizando essa classe, adicionaremos `throws` na assinatura do método `main()`.

```
public static void main(String[] args) throws SaldoInsuficienteEx
```

COPIAR CÓDIGO

Na hora de executar, nada vai mudar. Na execução, o *unchecked* é igual ao *checked*.

Então, se você ainda está com dúvida ou inseguro com esse tópico, não se desespere. É normal. Exceções é um tópico complicado e, com o tempo, praticaremos e nos sentiremos mais seguros com o código.

Ainda falta explicar mais uma ideia em nosso tratamento. Mas para isso, usaremos um outro exemplo, a seguir.

10

Para saber mais: Nomenclatura

No vídeo, usamos uma exceção com o nome `SaldoInsuficienteException`. Discutir nomes pode ser algo subjetivo e exige conhecimentos sobre o assunto. Ou seja, é pauta de longas discussões, mas acreditamos que um nome um pouco mais genérico para nossa exceção também seria uma solução adequada.

Por exemplo, a exceção poderia se chamar `SacaException` ou `ContaException`. Repare que usamos o nome do método ou da classe. Para detalhar mais o problema (valor do saldo, etc) podemos utilizar a mensagem da exceção, como já fizemos no curso:

```
throw new SacaException("Valor invalido: Saldo: " + this.saldo
```

COPIAR CÓDIGO

Dessa forma, caso tenha outro problema, basta alterar a mensagem.

De qualquer forma, saiba que encontrar o nome perfeito para as suas classes e métodos não é uma tarefa fácil e pode tomar o seu tempo. Em alguns casos, já encontramos nomes nas classes que deixaram claro que isso é apenas algo provisório e que deve ser alterado quando houver um consenso no nome.

01

Finally

Transcrição

> Começando deste ponto? Você pode fazer o DOWNLOAD (https://caelum-online-public.s3.amazonaws.com/834-java-excecoes/06/java4-aula6.zip) completo do projeto anterior e continuar seus estudos a partir daqui.

Fecharemos o tópico de exceções, trabalhando um novo exemplo para entender mais um detalhe do tratamento de erro.

Quando escrevemos uma aplicação um pouco mais complexa, muito provável que não funcione sozinha. Ela depende de outras aplicações. O melhor e mais claro exemplo que temos, é o nosso próprio celular!

Todos os aplicativos do Android usam Java, entretanto eles não possuem todos os dados no próprio aplicativo. Assim, é necessário que o aplicativo acesse o servidor para pedir informações.

Nessa comunicação com o servidor, podem acontecer diversos problemas. Estamos falando de uma comunicação na internet e, durante o processo, pode ser que ocorram erros que a aplicação não pode prever. O certo é que ela saiba como lidar com isso.

A ideia é introduzir o tratamento de exceções como se funcionassem em uma comunicação real, de grande porte. Para isso, preparamos uma classe chamada `Conexao`, a fim de simular uma conexão na rede e mostrar as dificuldades de s trabalhar com esses recursos relacionados com o tratamento de erros.

```
public class Conexao {

    public Conexao() {
        System.out.println("Abrindo conexao");
    }

    public void leDados() {
        System.out.println("Recebendo dados");
            //throw new IllegalStateException();
    }

    public void fecha() {
        System.out.println("Fechando conexao");
    }
}
```

COPIAR CÓDIGO

Vamos criar a classe no projeto, selecionando "New > Class", e colar o código acima.

Ótimo! Temos um **construtor** para simular uma **abertura de conexão**, um **método** para **ler os dados** do servidor e um **método**, não menos importante, para **fechar a conexão**.

Hora de testar! Criaremos uma nova classe com o método `main()`, que será chamada de `TesteConexao`:

```
public class TesteConexao {
    public static void main(String[] args) {

    }
}
```

COPIAR CÓDIGO

Qual é o nosso desafio? Ao trabalhar com conexões, precisamos realizar um tratamento de erro, como vamos tratar antecipadamente as exceções. Criaremos logo de cara, um `try` e, dentro dele, ficará a instância de `Conexao` .

```
public static void main(String[] args) {
    try {
        Conexao con = new Conexao();
    }
}
```

COPIAR CÓDIGO

A partir da conexão criada, podemos chamar o método `leDados()` e, depois, fecharemos a conexão com o método `fecha()` :

```
public static void main(String[] args) {
    try {
        Conexao con = new Conexao();
        con.leDados();
        con.fecha();
    }
}
```

COPIAR CÓDIGO

Pode acontecer um erro, então é necessário criar um `catch` para pegar esse possível erro. Na classe `Conexao` , existe um comentário de uma classe que ainda não conhecemos, a `IllegalStateException` . Essa é uma exceção padrão do mundo Java que já existe e indica que o objeto utilizado tem um estado inconsistente.

O `IllegalStateException` estende de `RuntimeException` , ou seja, é uma exceção do tipo *unchecked*. Vamos tirar o comentário dessa linha e salvar.

Voltando na `TesteConexao` , faremos um `catch` de `IllegalStateException` :

```
public static void main(String[] args) {
    try {
        Conexao con = new Conexao();
        con.leDados();
        con.fecha();
    } catch(IllegalStateException ex) {
        System.out.println("Deu erro na conexao");
    }
}
```

COPIAR CÓDIGO

Quando trabalhamos com recursos de abrir conexões com servidores, é importante se preocupar com o **fechamento** desse recurso, ou seja, o recurso que você abre, você deve fechar! Se não tratarmos bem esse recurso do sistema operacional, todo o sistema pode ser afetado.

> Esse é um conceito muito importante: Se abrimos uma conexão com o banco de dados, temos que fechá-la.

Por esse motivo, chamamos o `con.fecha()` e, na teoria, tem que aparecer no console a mensagem "Fechando conexão", certo? Ao testarmos, essa é a mensagem que temos no console:

```
Abrindo conexao
Recebendo dados
Deu erro na conexao
```

COPIAR CÓDIGO

Como podemos ver, deu erro na linha `con.leDados()`, então a execução parou, saiu abruptamente desse bloco, foi para o `catch` e não chegamos na linha `con.fecha()`. Por essa razão, chamaremos o método que fecha a conexão dentro do `catch`.

Entretanto, não podemos simplesmente fechar a conexão chamando o método a partir da variável `con`, pois ela só é visível dentro de `try`.

Faremos o seguinte: declaremos uma variável **nula** do tipo `Conexao` e, dentro do bloco `try`, vamos inicializá-la.

```
public static void main(String[] args) {
    Conexao con = null;
    try {
        con = new Conexao();
        con.leDados();
        con.fecha();
    } catch(IllegalStateException ex) {
        System.out.println("Deu erro na conexao");
        con.fecha();
    }
}
```

COPIAR CÓDIGO

Ao testarmos, teremos as seguintes mensagens no console:

```
Abrindo conexao
Recebendo dados
Deu erro na conexao
Fechando conexao
```

COPIAR CÓDIGO

Funcionou! Conseguimos fechar a conexão, mesmo dando erro. Repare que chamamos o método `fecha()` duas vezes, e isso não é muito legal.

Nossa intenção é que a conexão feche a qualquer custo, dando erro ou não. Mas, o `try-catch` tem um lugar próprio para isso. Estamos falando do bloco **`finally`**, um bloco *opcional* que podemos colocar no final e que **sempre** será executado

com ou sem erro. Dessa forma, não será mais necessário repetir o código para fechar a conexão.

```
public static void main(String[] args) {
    Conexao con = null;
    try {
        con = new Conexao();
        con.leDados();
    } catch(IllegalStateException ex) {
        System.out.println("Deu erro na conexao");
    } finally {
        con.fecha();
    }
}
```

COPIAR CÓDIGO

A questão é que não tem como fugir do `finally`, se ele estiver no código, sempre será executado. Ao executar o código, novamente, com as mudanças, veremos que não houve alteração. Foi exibida a mensagem "Fechando conexao", mesmo com o erro.

Muito bem! Agora, repare que a abertura e o fechamento de conexão se tornou um código difícil de ler. Não temos muita lógica aqui. Por isso, é interessante simplificar esse código, considerando que existe mais um problema, que veremos a seguir.

03

Try with resources

Transcrição

Estudaremos mais alguns detalhes sobre `try`, `catch` e `finally`. Um `try` sozinho **nunca** é válido. Ele exige pelo menos um `catch` ou um `finally`!

O código a seguir é válido, mesmo sem o `catch`:

```
Conexao con = null;
try {
    con = new Conexao();
    con.leDados();
} finally {
    con.fecha();
}
```

COPIAR CÓDIGO

Nesse caso, nós não queremos capturar a exceção em caso de erro. Queremos somente fechar a conexão. Se executarmos esse código, teremos a seguinte saída:

```
Abrindo conexao
Recebendo dados
Fechando conexao
Exception in thread "main" java.lang.IllegalStateException
        at Conexao.leDados(Conexao.java:10)
        at TesteConexao.main(TesteConexao.java:9)
```

COPIAR CÓDIGO

A exceção "explodiu" porque não temos um código para tratá-la. Então, a regra é que `try` deve ter *zero*, *um* ou *mais* `catch`. Podemos repetir `catch`, desde que as exceções sejam diferentes. Entretanto, o `try` pode ter um único `finally`.

Retomando o assunto pendente, concluímos que o código está, no mínimo, complexo para leitura. Além disso, encontramos um problema pior. Para entendê-lo, copiaremos a linha `throw new IllegalStateException()` e colaremos dentro do construtor `Conexao()`, após a linha que imprime "Abrindo conexão", a fim de criar um problema na construção do objeto.

```
public class Conexao {
    public Conexao() {
        System.out.println("Abrindo conexao");
        throw new IllegalStateException();
    }

    public void leDados() {
        System.out.println("Recebendo dados");
        throw new IllegalStateException();
    }

    // método fecha()
}
```

COPIAR CÓDIGO

Isso significa que, se ele chamar o construtor, será jogada uma exceção. Consequentemente, o objeto nunca será criado e também não terá uma referência inicializada.

Ao executarmos novamente a classe `TesteConexao`, tentaremos abrir a conexão, sabendo que dará erro.

```
Abrindo conexao
Deu erro na conexao
```

```
Exception in thread "main" java.lang.NullPointerException
        at TesteConexao.main(TesteConexao.java:13)
```

COPIAR CÓDIGO

Falamos que `finally` sempre será executado, com ou sem erro. Realmente tentamos fechar a conexão, só que a conexão está nula, ela nunca foi inicializada. Consequentemente, recebemos `NullPointerException`.

O que deveríamos fazer para simplificar esse código? Devemos chamar a `con.fecha()` se a conexão for diferente de `null`.

```
public static void main(String[] args) {
    Conexao con = null;
    try {
        con = new Conexao();
        con.leDados();
    } catch(IllegalStateException ex) {
        System.out.println("Deu erro na conexao");
    } finally {
        System.out.println("finally");
        if(con != null) {
            con.fecha();
        }
    }
}
```

COPIAR CÓDIGO

Então, teremos a seguinte saída:

```
Abrindo conexao
Deu erro na conexao
finally
```

COPIAR CÓDIGO

O `finally` foi executado, porém ele não entrou no `if` e, com isso, a conexão não foi fechada porque ela realmente é nula.

Nós precisamos simplificar esse código, mas como faremos isso?

A simplificação que faremos a seguir só entrou no Java `1.7`. A ideia é, na hora de usar o `try`, inicializarmos a variável dentro de `()`, juntando duas linhas em uma só:

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

    try (Conexao conexao = new Conexao()) {

    }
}
```

COPIAR CÓDIGO

Como se fosse um método, o `try` recebe a inicialização da variável `conexao`. Entretanto, esse código acima não compila, pois o Java exige que a `Conexao` siga um contrato, porque se colocarmos o cursor em cima da palavra que tem o erro, veremos que a classe `Conexao` **precisa** implementar uma interface chamada `AutoCloseable`.

```
public class Conexao implements AutoCloseable {}
```

COPIAR CÓDIGO

Essa é uma interface, e a ideia das interfaces é definir um contrato e, caso você "assine" um contrato, será obrigado a implementar o método. Nesse caso, o método que somos obrigados a implementar da interface, é o método `close()`.

Então, em vez de chamar o método `fecha()`, chamaremos `close()`. Vamos apagar o `fecha()` e mover a linha que imprime para o novo método:

```
public class Conexao implements AutoCloseable{
    public Conexao() {
        System.out.println("Abrindo conexao");
        throw new IllegalStateException();
    }

    public void leDados() {
        System.out.println("Recebendo dados");
        throw new IllegalStateException();
    }

    @Override
    public void close() {
        System.out.println("Fechando conexao");
    }
}
```

COPIAR CÓDIGO

O `AutoCloseable` exige que tenhamos o método `close()`, mas podemos deixar o método menos perigoso, retirando o `throws Exception`. Assim, simplificaremos um pouco o código e não será necessário mais um tratamento de erro para quem faz a chamada.

Além disso, vamos focar no problema da conexão, na qual não há exceção na construção do objeto.

```
    public Conexao() {
        System.out.println("Abrindo conexao");
        //throw new IllegalStateException();
    }
```

COPIAR CÓDIGO

Voltamos ao `TesteConexao` e vemos que o problema foi resolvido. O novo `try` exige que a classe do objeto que está sendo construído, implemente a interface

`AutoCloseable` . Mas antes de continuar, para evitar confusão, vamos comentar o nosso código anterior. Construiremos um novo `try-catch` .

Chamaremos o método `leDados()` , dentro do `try` :

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

    try (Conexao conexao = new Conexao()) {
        conexao.leDados();
    }
}
```

COPIAR CÓDIGO

Essas três novas linhas de código, se referem ao antigo bloco:

```
//    Conexao con = null;
//    try {
//        con = new Conexao();
//        con.leDados();
//    }
```

COPIAR CÓDIGO

Ao executar esse novo código, obteremos a seguinte saída:

```
Abrindo conexao
Recebendo dados
Fechando conexao
Exception in thread "main" java.lang.IllegalStateException
        at Conexao.leDados(Conexao.java:11)
        at TesteConexao.main(TesteConexao.java:8)
```

COPIAR CÓDIGO

Repare que a conexão foi fechada automaticamente. Então concluímos que essas três linhas do console se referem aos antigos blocos `try` e `finally`. Não precisamos mais escrever explicitamente o bloco `finally`, pois o novo `try` já nos garante que o recurso que está sendo aberto dessa forma será fechado automaticamente, desde que ele implemente a interface `AutoCloseable`.

Isso já foi feito, sem ter que nos preocupar com o fechamento da conexão. O que ainda não fizemos é o `catch`. Veja que a exceção aconteceu no método `leDados()` e caiu no console e ninguém resolveu esse problema. Vamos criá-lo agora! Lembrando que ele pode ou não ser adicionado, diferente de `finally`, que tem o fechamento do recurso.

> O `try` com `finally` é *válido* sem o `catch`.

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

    try (Conexao conexao = new Conexao()) {
        conexao.leDados();
    } catch(IllegalStateException ex) {
        System.out.println("Deu erro na conexao");
    }
}
```

COPIAR CÓDIGO

Vamos executar novamente para ver a diferença na execução:

```
Abrindo conexao
Recebendo dados
Fechando conexao
Deu erro na conexao
```

COPIAR CÓDIGO

Legal! Podemos concluir que essas cinco linhas novas substituem o código anterior e, por isso, o código ficou muito melhor. Vale a pena aprender esse novo `try`. Além disso, veremos quando acontece uma exceção durante a construção do objeto `conexao`.

```
Abrindo conexao
Deu erro na conexao
```

COPIAR CÓDIGO

O `close()` não foi chamado porque o objeto nem conseguiu ser construído. Esse é um código mais recente que, talvez, precisará de manutenção, porém aprendemos a usar o `try-catch` de uma forma mais enxuta.

Agora é a hora de colocar os aprendizados em prática, fazendo os exercícios a seguir!

08

Para saber mais: Exceções padrões

No vídeo, usamos a exceção `IllegalStateException`, que faz parte da biblioteca do Java e indica que um objeto possui um estado inválido. Você já deve ter visto outras exceções famosas, como a `NullPointerException`. Ambos fazem parte das exceções padrões do mundo Java que o desenvolvedor precisa conhecer.

Existe outra exceção padrão importante que poderíamos ter usado no nosso projeto. Para contextualizar, faz sentido criar uma conta com uma agência que possui valor negativo? Por exemplo:

```
Conta c = new ContaCorrente(-111, 222); //faz sentido?
```

COPIAR CÓDIGO

Não faz sentido nenhum. Por isso, poderíamos verificar os valores no construtor da classe. Caso o valor esteja errado, lançamos uma exceção. Qual? A `IllegalArgumentException`, que faz parte das exceções do biblioteca do Java:

```
public abstract class Conta {

    //atributos omitidos

    public Conta(int agencia, int numero){

        if(agencia < 1) {
            throw new IllegalArgumentException("Agencia inváli
        }

        if(numero < 1) {
```

```
            throw new IllegalArgumentException("Numero da conta
        }

        //resto do construtor foi omitido
    }
```

COPIAR CÓDIGO

A `IllegalArgumentException` e `IllegalStateException` são duas exceções importantes, que o desenvolvedor Java deve usar. Em geral, quando faz sentido, use uma exceção padrão em vez de criar uma própria.

Tudo bem?

10

Conclusão

Transcrição

Chegamos ao fim desse curso sobre **Exceções**, um dos tópicos mais complexos do mundo Java. A partir delas, tudo ficará mais fácil. Elas reúnem muitas áreas de Java, como o conhecimento sobre as classes, herança, polimorfismo, reutilização de código, etc. Há uma sintaxe especial focada somente nas exceções, composta por:

- `try` ;
- `catch` ;
- `finally` ;
- `throws` ;
- `throw` .

Todas essas palavras-chave são relacionadas com as exceções que existem para mudar o fluxo da aplicação.

Demos uma atenção maior às exceções, porque se algo anormal acontecer em seu programa, precisamos saber lidar com isso e entender. Por isso, começamos essa aula aprendendo como funciona a pilha de execução *Stack*, entendendo como funciona a organização da execução de um método ou de vários métodos no mundo Java.

Quando uma exceção acontece, ela manipula aquela pilha de execução. No mundo Java, existe uma hierarquia que separa erros da máquina virtual e exceções para o

desenvolvedor. Dentro das exceções para o desenvolvedor, existem duas categorias:

- *checked* é o tipo em que o compilador verifica;
- *unchecked* é o tipo em que o compilador não verifica.

Na hora da execução, acaba sendo tudo igual: uma "bomba" que cai na pilha e muda o fluxo.

Vimos também como criar as nossas próprias exceções, criar uma classe e encaixá-la na hierarquia usando a palavra-chave `throw` junto com o `throws`, deixando claro ao compilador se tal método é perigoso ou não. Vimos todo o tratamento com `try`, `catch` e `finally` e, no final, vimos uma nova cláusula para abrir um recurso junto com o `try` e simplificar o nosso código.

Saber disso é um prato cheio para qualquer desenvolvedor Java que quer ter sucesso, mas fique tranquilo se você ainda está com dúvida, pois esse tópico é enorme e cheio de detalhes. Com o tempo, essas palavras-chave ficarão mais naturais.

Muito obrigado por ter assistido a esse curso! Te convido a continuar os cursos da carreira Java, pois ainda há algo a aprender para realmente fecharmos essa parte mais conceitual sobre o mundo Java! Até o próximo curso! :)