Estrutura de Repetição e Tratamento de Exceções

Nivelamento de Lógica de Programação e Programação Orientada a Objeto

Estrutura de Repetição - *for* e *while*

Java possui duas estruturas de controle de repetições: `for` e `while` (`do-while` é uma variação desta última). Enquanto a primeira executa uma quantidade pré-definida conhecida de repetições, a outra define repetições baseadas em uma condição booleana.

Estrutura de Repetição - *for* e *while*

While/Do While

A estrutura `while` executa continuamente um bloco de código baseado na avaliação de uma expressão booleana (verdadeiro/falso). Sua sintaxe-base é:

```
while (expressão) {
    // statement(s)
}
```

Estrutura de Repetição - *for* e *while*

While/Do While

Assim como fornecemos uma variável booleana para a estrutura condicional if, aqui também podemos fornecer uma única variável ou uma expressão completa.

Considere o trecho a seguir,

```
class WhileDemo {
    public static void main(String[] args){
        int count = 1;
        while (count < 11) {
            System.out.println("Count is: " +
count);
            count++;
        }
    }
}
```

Mas, o que acontece quando não temos um valor inicial? Esse é o caso especial da variação `do-while`.

Enquanto no caso padrão da estrutura `while` a primeira execução já está sujeita a avaliação da expressão booleana (ou seja, o bloco pode ser executado `0..*` repetições). Temos que para a variação `do-while` o bloco será executado no mínimo 1 vez, visto que a avaliação da expressão de repetição será apenas ao final do bloco.

Estrutura de Repetição - *for* e *while*

While/Do While

```
class DoWhileDemo {
    public static void main(String[] args){
        int count = 1;
        do {
            System.out.println("Count is: " +
count);
            count++;
        } while (count < 11);
    }
}
```

Estrutura de Repetição - *for* e *while*

Laço de Repetiçao **FOR**

A estrutura de repetição for tem a intenção de percorrer um conjunto definido de valores. O laço for executa a repetição de um bloco de código repetidas vezes até satisfazer uma determinada condição. A sintaxe básica é

```
for (inicialização; terminação; incremento) {
    instrução()
}
```

Estrutura de Repetição - *for* e *while*

Laço de Repetiçao **FOR**

Com base nos exemplos vistos na seção anterior (while), podemos re-escrever um programa que escreva em console todos os números inteiros entre 1-10.

```
class ForDemo {
    public static void main(String[] args){
         for(int i=1; i<11; i++){
              System.out.println("Count is: " + i);
         }
    }
}
```

Estrutura de Repetição - *for* e *while*

Laço de Repetiçao **FOR**

Os laços `for` são especialmente úteis para percorrer *arrays*.
Considere o seguinte exemplo

```
class ForArrayDemo {
    public static void main(String[] args){
         int[] numbers = {1,2,3,4,5,6,7,8,9,10};
         for(int i=0; i < numbers.length; i++){
              System.out.println("Array value is: " + numbers[i]);
         }
    }
}
```

Sintaxe especial do **for** para arrays

Java também oferece uma variação do laço `for`, chamada foreach. Embora a sintaxe seja diferente, continuamos usando a palavra-chave `for`. Veja a seguir

```
class EnhancedForDemo {
    public static void main(String[] args){
         int[] numbers = {1,2,3,4,5,6,7,8,9,10};
         for (int item : numbers) {
             System.out.println("Count is: " + item);
         }
    }
}
```

Sintaxe especial do **for** para arrays

Neste trecho não temos uma variável de controle. O próprio compilador Java entende que estamos percorrendo um array, e associa cada valor contido no array à variável informada. Ou seja, o código anterior é equivalente a

```
class EnhancedForDemo {
    public static void main(String[] args){
         int[] numbers = {1,2,3,4,5,6,7,8,9,10};
         for (int i=0; i < numbers.length; i++) {
             int item = numbers[i];// here is the trick!
             System.out.println("Count is: " + item);
         }
    }
}
```

Tratamento de Exceções

Tratamento de Exceção

Captura e Tratamento de Erros

Tudo seria tão bom se todo código que escrevemos funcionasse sem erros, certo? Porém, na programação, nunca podemos garantir a ausência de erros.

Estamos sujeitos aos cenários inesperados e, assim, todo bom programa deve ser capaz de lidar com os erros.

É a robustez ou a resiliência em tolerar falhas.

Java define `Throwable` como a classe raiz para todos os erros (`Error`) ou exceções (`Exception`).

Vamos nos ater apenas a esta última, que ainda pode ser divida em exceções checadas ou não checadas.

Exceções checadas e não checadas

As e**xceções checadas** são subtipos de Exception e devem, obrigatoriamente, definir um tratamento caso ocorram ou que sejam repassadas na pilha de execução.

Essa obrigação é garantida em tempo de compilação. Essas exceções são comuns em cenários onde é esperado e possível o erro, como na leitura de um arquivo ou numa operação em bancos de dados, ou na conversão de um texto em URL (ex.: *IOException, SQLException*).

Já as **exceções não checadas** são filhas de RuntimeException e não precisam definir tratamento explícito.

Normalmente, representam situações inesperadas como operações ilegais e mal uso da API (ex.: *NullPointerException, IllegalArgumentException*).

Captura e tratamento de exceções

Tratamento de Exceção

Você pode criar suas próprias classes de exceção representando regras e condições específicas.

Para isso, é necessário definir uma classe como subtipo de Exception ou RuntimeException, como em

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;

public class PrimeiroPrograma {
  public static void main(String[] args) {
      try {
          File file = new File("nonexistentfile.txt");
          FileReader reader = new FileReader(file);
      } catch (FileNotFoundException e) {
          System.out.println("File not found: " + e.getMessage());
      }
  }
}
```

Captura e tratamento de exceções

As boas práticas recomendam que o bloco `try` seja o mais conciso possível, isolando o ponto onde pode ocorrer a exceção.

```java
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;

public class PrimeiroPrograma {
  public static void main(String[] args) {
      try {
          File file = new File("nonexistentfile.txt");
          FileReader reader = new FileReader(file);
      } catch (FileNotFoundException e) {
          System.err.println("File not found: " + e.getMessage());
      } catch (Exception e) {
          System.err.println("Erro inesperado: " + e.getMessage());
      }
  }
}
```

Captura e tratamento de exceções

As boas práticas recomendam que o bloco `try` seja o mais conciso possível, isolando o ponto onde pode ocorrer a exceção.

```java
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;

public class PrimeiroPrograma {
  public static void main(String[] args) {
      try {
          File file = new File("nonexistentfile.txt");
          FileReader reader = new FileReader(file);
      } catch (FileNotFoundException e) {
          System.err.println("File not found: " + e.getMessage());
      } catch (Exception e) {
          System.err.println("Erro inesperado: " + e.getMessage());
      }
  }
}
```

Captura e tratamento de exceções

Tratamento de Exceção

Além do try-catch também existe o finally que sempre é executado, tanto em sucesso do try como em erro e captura no catch. Esse bloco serve para fechar o stream de leitura do arquivo, tanto em sucesso como em erro.

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

public class PrimeiroPrograma {
  public static void main(String[] args) {
      FileReader reader = null;
      try {
          File file = new File("nonexistentfile.txt");
          reader = new FileReader(file);
      } catch (FileNotFoundException e) {
          System.err.println("File not found: " + e.getMessage());
      } catch (Exception e) {
          System.err.println("Erro inesperado: " + e.getMessage());
      } finally {
          if (reader != null) {
              try {
                  reader.close();
              } catch (IOException e) {
                  System.err.println("Erro ao fechar o arquivo: " +
e.getMessage() + ".");
              }
          }
      }
  }
}
```

Try-with-resources

Esse recurso surgiu no Java 7 e se beneficia da interface *AutoCloseable*. Esta é uma interface que contém o método *close* e que deve ser chamado ao final do seu uso. Vamos ver como fica o código anterior reescrito com o try-with-resources.

```
import java.io.File;
import java.io.FileReader;
import java.io.IOException;

public class PrimeiroPrograma {
  public static void main(String[] args) {
      File file = new File("nonexistentfile.txt");
      try (FileReader reader = new FileReader(file)) {
          //leitura do arquivo
      } catch (IOException e) {
          System.err.println("Erro inesperado: " + e.getMessage());
      }
  }
}
```

Exercícios

Exercício 1: Agendamento

Agenda de compromissos de trabalho. Só deve permitir agendamentos em dias e horários de trabalho.

Desafio

Pesquisar e usar o try-with-resources para manipulação do Scanner.

Exercício 2: Divisão Segura

Crie um programa que peça ao usuário dois números inteiros.

O programa deve calcular a divisão do primeiro pelo segundo. Use um bloco `try-catch` para tratar a exceção `ArithmeticException` caso o segundo número seja zero.

Exercício 3: Validação de Idade com LocalDate

Crie um programa que peça ao usuário sua data de nascimento no formato YYYY-MM-DD.

Use a classe `LocalDate` para converter a entrada e calcule a idade.

Use um bloco `try-catch` para tratar a exceção `DateTimeParseException` caso o formato da data seja inválido.

Exercício 4: Horário de Reunião com LocalDateTime

Peça ao usuário para inserir uma data e hora para uma reunião no formato YYYY-MM-DDTHH:mm:ss.

Utilize `LocalDateTime` para converter a entrada.

Se o formato for incorreto, capture a exceção `DateTimeParseException`.

Exercício 5: Múltiplos Tipos de Exceção

Crie um programa que peça ao usuário uma data de nascimento e um número.

No mesmo bloco `try`, tente converter a data para **LocalDate** e dividir 10 pelo número digitado.

Use múltiplos blocos catch para tratar **DateTimeParseException**, **InputMismatchException** **ArithmeticException** separadamente.

Exercício 6: do-while - Adivinhe o Número

Escreva um programa que gera aleatoriamente um número entre 1 e 10. O usuário deve adivinhar o número. O programa deve informar se o número inserido é muito alto, muito baixo ou correto. O jogo continua até que o usuário adivinhe corretamente.

Utilize a biblioteca do java *Random*.

Exemplo:

```
random.nextInt(10) + 1;
```

Exercício 7: for - Tabuada

Escreva um programa que solicita ao usuário para inserir um número e imprime a tabuada desse número de 1 a 10.

Exemplo de saída:

```
1 x 1 = 1
1 x 2 = 2
1 x 3 = 3
1 x 4 = 4
1 x 5 = 5
1 x 6 = 6
1 x 7 = 7
1 x 8 = 8
1 x 9 = 9
1 x 10 = 10
```

Exercício 8: while - Números Primos

Escreva um programa que solicita ao usuário para inserir um número e verifica se esse número é primo ou não.

Obrigada