Fundamentos da Programação Orientada a Objetos

UNIDADE 07

Tratamento de Exceção

*Nesta unidade, abordaremos a manipulação e tratamento de erros. Entretanto, não estamos nos referindo a achar erros de código, o que a IDE poderia encontrar facilmente. Mesmo quando programamos profissionalmente e dominamos a linguagem, há várias situações legítimas em que problemas podem ocorrer. Além disso, o ideal também é sermos **preventivos**, para que um erro de lógica seja imediatamente identificado e corrigido.*

No Java, esse tratamento é realizado por meio do **mecanismo de exceções**, que pode ser usado para:

1. Proteger

Quando escrevemos nossas classes, podemos criar e disparar nossos próprios erros para evitar que uma regra de negócio seja violada. Por exemplo, vamos supor que criemos uma classe **aluno**. Certamente, iremos querer impedir que alguém cadastre um valor negativo no campo da idade ou da matrícula, e podemos fazer isso disparando uma exceção.

2. Tratar

Algumas situações de erro são conhecidas, mas podem ser identificadas e tratadas de maneira elegante em um sistema. Por exemplo, digamos que em uma aplicação de edição de textos, o usuário tente criar um arquivo em uma pasta em que ele não possui acesso. O Java disparará um erro, mas sua aplicação pode identificar essa ocorrência e, ao invés de interromper seu funcionamento, ela irá mostrar uma mensagem de erro ao usuário e simplesmente pedir para que ele escolha outro local.

3. Registrar

Infelizmente, algumas vezes erros completamente imprevistos ocorrerão. Nesse caso, pode ser interessante tratá-los de alguma forma, como registrá-los em um arquivo, para posterior análise.

4. Gerenciar recursos

Podemos utilizar o mecanismo de exceções para garantir que recursos utilizados pela aplicação sejam corretamente finalizados. Por exemplo, pode ser necessário fechar um arquivo aberto, ou encerrar uma conexão de redes.

Tudo isso adicionará um grau adicional de robustez em nossas aplicações, tornando nosso código defensivo e evitando que problemas ainda maiores se propaguem ao longo do tempo.

Tratando exceções no Java

Lidando com as mensagens de erro

1. Execute o código indicado no modelo e responda:
    a. Qual é o problema do código?
    b. Qual é a sua saída?

EXERCÍCIO

Exceções

Figura 1 – Exercício: exceções

```
</>
1
2
3
4
5
6
7
```

```
8
9   public class Exemplo {
10      static void realizarConta() {
11          int x = 0;
12          int y = 10;
13          int z = y/x;
14          System.out.println(z);
15      }
16
17      public static void main(String[] args)
18  {
19          realizarConta();
20      }
21  }
```

Resolução do exercício

Resposta 1 – classe forma:

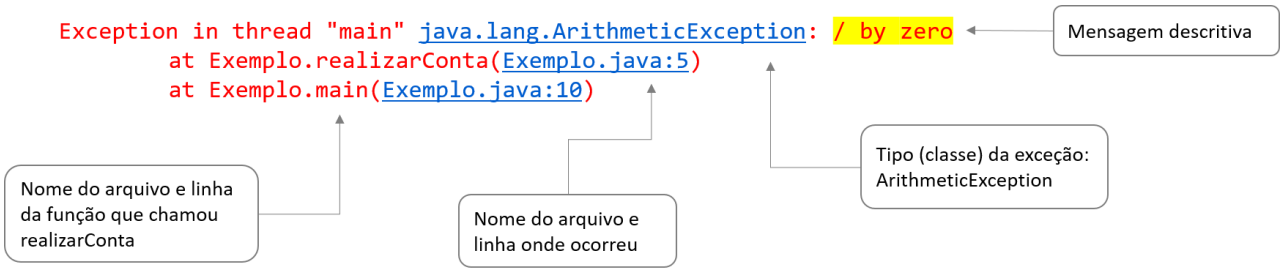
a. Ocorre uma divisão por zero.
b. O seguinte texto é impresso:

```
Exception in thread "main"
java.lang.ArithmeticException: / by zero
        at Exemplo.realizarConta(Exemplo.java:5)
        at Exemplo.main(Exemplo.java:10)
```

Fonte: Autores (2021).

Esse exercício, apesar de simples, nos mostra um exemplo do uso do mecanismo de exceção. Perceba que o código em questão possui um erro, uma vez que tem duas variáveis inteiras, sendo a variável **x** com valor 0. A divisão de **y** por **x** é, portanto, impossível. E como o Java sinaliza esse problema? Ele irá disparar uma **exceção**. Como nesse código não fizemos qualquer tratamento especial desse problema, o método ***main*** foi encerrado e o Java imprimiu em sua saída de erro (System.err) a descrição do erro, conhecida como ***stack trace***. A figura a seguir ajudará você a entender em detalhes essa mensagem.

Figura 2 – Exercício: *stack trace* explicado



Fonte: Autores (2021).

Aqui já nos deparamos com uma excelente característica do mecanismo: os erros são extremamente descritivos. Observe que:

a. O erro possui uma **mensagem descritiva**, que explica o problema ocorrido
b. Além da mensagem, o erro é um objeto e, portanto, possui uma classe. Essa classe normalmente possuirá um nome que permite-nos entender o erro quanto a sua natureza. Neste caso, trata-se de uma ***ArithmeticException***, uma vez que foi uma operação aritmética que causou o problema.
c. Por fim, há o ***stack trace*** propriamente dito. Trata-se da descrição do ponto exato do código em que o erro ocorreu. O Java inicia pelo ponto mais específico e lista, linha a linha, todas os arquivos e métodos que foram chamados para que o código chegasse até ali. Por isso, temos duas linhas descritas na mensagem: a da função realizarConta e a do próprio método main. Isso permite que rastreemos não só o ponto em que o erro ocorreu, mas tenhamos uma boa noção do seu contexto. Afinal, um mesmo método pode ser chamado de diferentes locais no código.

> Não se assuste com o tamanho do **texto da exceção**, pois é perfeitamente normal que ela atinja várias linhas. Aprenda a ler o *stacktrace* e use-o a seu favor.

Tratamento de erros

Pode parecer bastante fácil corrigir esse código: basta informar um valor correto para **y**, não é? Agora, o que poderíamos fazer caso y não fosse diretamente definido, mas sim, lido do teclado? Nesse caso, o erro poderia ocorrer apenas algumas vezes, quando nosso usuário digitasse o valor 0. Vamos explorar algumas formas diferentes para realizar esse tratamento.

Utilizando condicionais

A primeira forma que possuímos para tratar erros é utilizar condicionais que testem o problema antes que ele ocorra, por exemplo:

```
</>
```

```
1   static void realizarConta(int x, int y) {
2           if (x == 0) {
3               System.out.println("Divisão por zero");
4           } else {
5               System.out.println(y/x);
6           }
7       }
8
```

Essa abordagem, porém, apresenta um grande problema. Nem sempre a função que trata o erro é a mesma função que realiza o cálculo. Por exemplo, poderíamos querer manter a função **realizarConta** somente com o cálculo e deixar no ***main*** a impressão do texto. Nesse caso, o ideal é **capturar a exceção** gerada pela divisão.

Capturando exceções

Quando uma exceção ocorre, ela abandona imediatamente o método que a gerou. Em seguida, ela é propagada para a função que chamou esse método e, caso haja um tratamento, essa propagação é interrompida. Caso não haja, ela fará com que esse método também seja abandonado, e esse ciclo pode continuar até que ela deixe o ***main***, finalizando o programa e encerrando a aplicação.

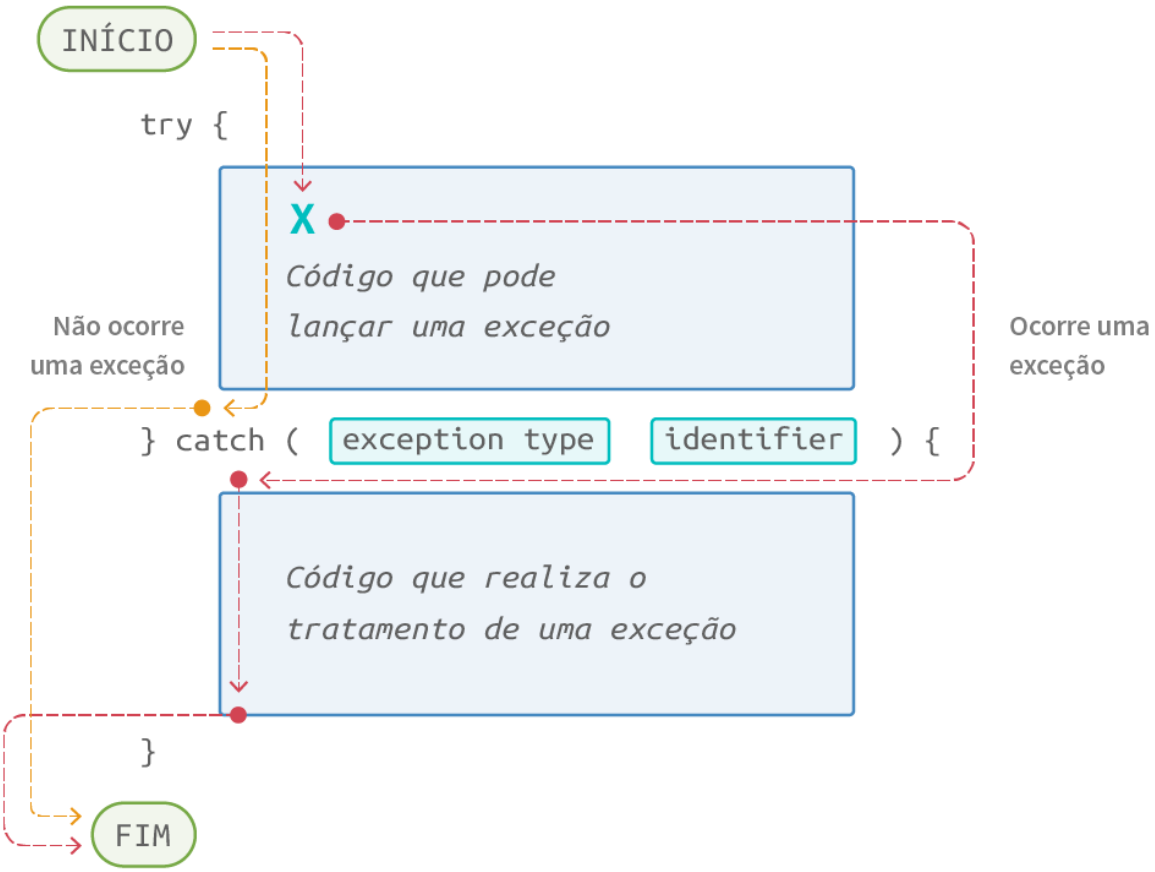
Utilizamos o comando ***try...catch*** para fazer esse tratamento. Primeiramente, vamos verificar como esse esquema de ***try*** (**tente**) e ***catch*** (**capture**) funciona, conforme a figura a seguir.

Observe os blocos de comandos ***try*** e ***catch***:

- Se ocorrer uma exceção dentro do bloco ***try***, o código é desviado para o bloco ***catch*** correspondente.
- Pode haver mais de um bloco ***catch***, cada um pegando uma **exceção diferente** e realizando o código que **trata dessa exceção**.

- Para saber qual desvio ou bloco ***catch*** deve ser executado, precisamos identificar **a classe da exceção** e nomear um **objeto da exceção**, conforme indicado na figura.
- Se não ocorrer uma exceção dentro do bloco ***try***, a execução **pula** os blocos de ***catch*** e finaliza

Figura 3 – Organização dos blocos de comando para ***try...catch***



Fonte: Autores (2021).

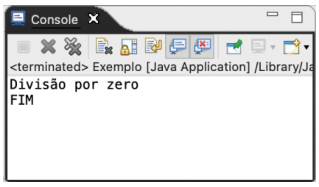
Agora, vamos verificar como esse esquema é implementado, conforme apresentado no exemplo a seguir.

Observe que a classe da exceção é a `ArithmeticException` e o objeto da classe se chama **e**, neste exemplo.

Figura 4 – Uso de *try...catch*

</>

```
1   public class Exemplo {
2       static int realizarConta(int x,
3   int y) {
4           return y / x;
5       }
6
7       public static void main(String[]
8   args) {
9           try {
10              int z = realizarConta(0,
11  10);
12              System.out.println(z);
13          } catch (ArithmeticException
14  e) {
15
16  System.out.println("Divisão por
17  zero");
18          }
19          System.out.println("FIM");
20      }
21  }
22
```



Fonte: Autores (2021).

Iniciamos o método ***main*** com o comando ***try***. Em seu interior, inserimos todo o código que poderia disparar um erro. Caso um erro ocorra, ele será imediatamente disparado para o comando ***catch*** da linha 10. A variável **e**, do tipo **ArithmeticException**, irá conter a descrição do erro gerado. Como a exceção foi capturada e tratada, o programa irá então interromper sua propagação e **continuar para a linha após o *catch***. É por isso que, mesmo com o erro, o texto "**FIM**" da linha 13 ainda é impresso.

E se nenhum erro ocorrer? O programa executará o ***println*** da linha 9 e então pulará o bloco ***catch***, executando também a linha 13.

Múltiplos blocos de *catch*

Vimos que dentro do bloco ***try***, podemos colocar mais de uma linha de código. Mas e se nessas linhas, exceções diferentes forem disparadas? Podemos utilizar múltiplos blocos de ***catch***, conforme demonstrado a seguir.

Execute o programa a seguir e confira se você consegue gerar a exceção `ArithmeticException`.

Figura 5 – Exercício: uso de múltiplos blocos de *catch*

```
</>
```

```
1   public class Exemplo {
2       static int realizarConta(int x, int y) {
3           return y / x;
4       }
5
6       public static void main(String[] args) {
7           try {
8               int z = realizarConta(2, 10);
9               System.out.println(z);
10
11              String x = null;
12              System.out.println(x.length());
13          } catch (ArithmeticException e) {
14              System.out.println("Divisão por zero");
15          } catch (NullPointerException e) {
16              System.out.println(e.getMessage());
17          }
18          System.out.println("FIM");
19      }
20  }
21
```

Fonte: Autores (2021).

Observe que o código só é desviado para o ***catch*** em que a exceção ocorreu. Os demais blocos de ***catch*** não serão executados. Além disso, observe que dessa vez utilizamos o objeto "**e**" recebido por parâmetro. Ele permite acessar dados da exceção, como sua mensagem. Esta mensagem é a mesma que sairia no ***stack trace***, caso deixássemos a exceção sair do bloco ***main*** (tente testar isso você mesmo). Além do ***getMessage()***, você pode mandar imprimir o ***stack trace*** completo, tal como ocorre quando o código sai do main, com o comando: ***e.printStackTrace()***.

Multi-*catch*

Muitas vezes, um grupo de exceções apresenta tratamento idêntico, tal como mostrar a mensagem de erro, ou ignorá-la para que o usuário possa repetir a ação. Nesse caso, você pode especificar um grupo de exceções a serem capturadas por meio do operador de |, como apresentado no exemplo a seguir.

Execute o programa a seguir e confira se você consegue gerar a exceção `ArithmeticException`.

Figura 6 – Exercício: uso de *multi-catch*

```
</>
```

```
1   public class Exemplo {
2       static int realizarConta(int x, int y) {
3           return y / x;
4       }
5
6       public static void main(String[] args) {
7           try {
8               int z = realizarConta(2, 10);
9               System.out.println(z);
10
11              String x = null;
12              System.out.println(x.length());
13          } catch (ArithmeticException | NullPointerException
14  e) { // 2 tipos de
15              System.out.println(e.getMessage());
16  // exceção
17
18  // tratados da
19          }
20  // mesma forma
21          System.out.println("FIM");
22      }
23  }
24
```

Fonte: Autores (2021).

Captura hierárquica

Por fim, é importante saber que as exceções também fazem parte de uma **hierarquia de classes**. Isso torna possível capturá-las de maneira hierárquica, como se em cada ***catch*** houvesse um teste de ***instanceof***. Todas as exceções derivam da classe ***exception***, o que nos permitirá reescrever o ***catch*** como:

```
</>
```

```
1           } catch (Exception e) {
2               System.out.println(e.getMessage());
3           }
4
```

É possível até mesmo combinar as três abordagens, desde que as regras mais abrangentes fiquem por último e as mais específicas, por primeiro. O ***try*** entrará no primeiro catch que encontrar, sem executar os demais. Discutiremos mais sobre **exceções hierárquicas** e sua importância em breve, no tópico **disparando exceções**.

Cláusula *finally*

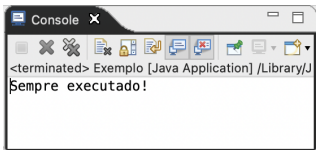
Em algumas situações, podemos querer que um trecho de código seja sempre executado, independentemente de uma exceção ter disso disparada ou não. Para esses casos, podemos associar ao ***try*** a cláusula ***finally***. Veja um exemplo a seguir.

Execute o programa a seguir e confira se você consegue apresentar a mensagem, conforme tela apresentada.

Figura 7 – Exercício: cláusula *finally*

```
</>
```

```
1   public class Exemplo {
2       static int realizarConta(int x,
3   int y) {
4           return y / x;
5       }
6
7       public static void main(String[]
8   args) {
9           try {
10              int z = realizarConta(2,
11  10);
12              if (z == 5) return;
13
14              System.out.println(z);
15
16              String x = null;
17
18  System.out.println(x.length());
19          } catch (Exception e) {
20
21  System.out.println(e.getMessage());
22          } finally {
23
24  System.out.println("Sempre
25  executado!");
26          }
27          System.out.println("FIM");
28      }
29  }
30
```



Fonte: Autores (2021).

Note que o código dentro do ***finally*** foi executado mesmo com a presença do return da linha 9! Como exercício, teste outros valores para o método realizar conta e observe como o ***finally*** sempre será executado – não importando se uma exceção foi disparada, se houve um ***return***, ou mesmo, se o método executou até o final. Por fim, um último detalhe: blocos ***try...finally***, sem nenhum ***catch***, também são permitidos!

Criando e disparando exceções

O mecanismo de exceções também permite que você crie e dispare suas próprias exceções. Além disso, ele fornece um conjunto de classes padrão de exceção, que facilitam o trabalho nos casos comuns, e facilitam a classificação das exceções nos demais casos.

Tipos de exceções

Primeiramente, é importante saber que existem diferentes **tipos de exceções**. A hierarquia de classes das exceções começa com quatro classes importantes.

Figura 8 – Classes básicas de exceção

Fonte: Autores (2021).

A classe ***throwable*** representa qualquer coisa que possa ser disparada como exceção e capturada na cláusula catch. Destacamos três métodos importantes:

- ***getMessage***: observe que o Java especifica que todas as exceções devem possuir uma mensagem descritiva de erro. Mesmo quando criamos nossas exceções, devemos nos preocupar em escrever um texto descritivo do problema ocorrido.
- ***getCause***: em algumas situações, pode ser interessante capturar uma exceção específica e gerar outra mais abrangente. Por exemplo, em um método que faz a carga dos dados da sua aplicação de diferentes locais, você pode querer capturar a exceção específica das várias fontes de dados (SQLException, IOException etc.) e disparar uma exceção mais geral. Nesse caso, é possível incluir a exceção original como causadora
- ***printStackTrace***: imprime o ***stack trace*** da exceção.

Entretanto, quando formos criar nossas próprias exceções, jamais as criaremos como filhas diretas de ***throwable***. No lugar, o Java fornece as classes ***exception*** e ***RuntimeException*** para isso. Elas representam os dois tipos de exceção da linguagem:

- Exceções **verificadas** (filhas de ***exception***): devem ser obrigatoriamente capturadas pelo programador. No caso delas, a cláusula try...catch será obrigatória, ou o método terá de sinalizar que dispara aquela exceção. Geralmente representam problemas comuns, relacionados à situação sendo modelada. Por exemplo, as

classes que carregam arquivos no Java disparam uma exceção verificada do tipo ***IOException***. Isso ocorre porque há vários problemas comuns na carga de arquivos (arquivo não existir, estar corrompido, não ter permissão de leitura etc.).

- Exceções **não verificadas** (filhas de ***RuntimeException***): não precisam ser tratadas pelo programador. Geralmente, representam problemas de programação, que poderiam ser prevenidos de outra forma. A exceção que vimos no exemplo, ***ArithmeticException***, é um tipo de exceção desse tipo.

E a classe **error**? Ela também irá disparar exceções não verificadas, porém, representam erros graves, cujo tratamento não é possível. Frequentemente, ela só é capturada para fins de registro. Elas geralmente são disparadas pela Virtual Machine. Um exemplo de erro é a falta de memória. Não há o que a aplicação fazer caso um ***OutOfMemoryError*** seja disparado.

Disparando exceções não verificadas

Para disparar uma exceção, utilizamos a cláusula ***throw***, com o objeto da exceção a ser disparado. O Java possui uma série de classes padrão de exceção para problemas comuns, veja alguns exemplos na tabela a seguir.

Figura 9 – Exceções comuns

Exceção	Verificada?	Descrição
CloneNotSupportedException	**Sim**	O método clone() não pode ser chamado nesse objeto
IllegalArgumentException	Não	O parâmetro de um método possui valor inválido
IllegalStateException	Não	Um método foi chamado em um momento indevido
IndexOutOfBoundsException e *ArrayIndexOutOfBoundsException*	Não	O sistema tentou acessar um índice inválido em uma lista ou *array*
IOException	**Sim**	Problema na leitura/escrita de arquivos
NullPointerException	Não	Um método foi chamado em uma variável nula
SQLException	**Sim**	Problema reportado pelo banco de dados

Fonte: Autores (2021).

Vejamos um exemplo: digamos que você está programando o método setIdade, da classe pessoa. Sabemos que pessoas não possuem idades negativas e não podem viver centenas de anos. Então, poderíamos implementar o método da forma apresentada a seguir.

Figura 10 – Exemplo: método setIdade

```
</>
 1   public class Pessoa {
 2       private int idade;
 3       ...
 4       public void setIdade(int idade) {
 5           if (idade <= 0 || idade >= 130) {
 6               throw new IllegalArgumentException("Idade
 7   inválida: " + idade);
 8           }
 9           this.idade = idade;
10       }
11   }
12
```

Fonte: Autores (2021).

Observe o uso da cláusula ***throw*** na linha 6. Ela foi seguida da criação do objeto da exceção, o que pode ser visto com o uso do ***new***. Como essa linha abandonará o método **setIdade** imediatamente caso seja atingida, não é necessário colocar o restante do código em um ***else***.

Note que usamos a exceção padrão, não verificada, ***IllegalArgumentException***, para essa situação. Afinal, geralmente o programa não permitirá que uma idade inválida seja fornecida (isso será barrado na interface gráfica) e, portanto, passar um parâmetro inválido está mais próximo de um erro de programação do que realmente de uma situação natural que deva ser testada o tempo todo.

Testar parâmetros dessa forma é uma boa prática, e tem até um nome: **código de guarda** (***code guard***).

IMPORTANTE

Procure sempre **programar de maneira preventiva**, isso evita que um erro como o da Figura 10 se propague. Acredite: vai ser bem mais fácil diagnosticar e corrigir um problema se uma exceção for disparada no ponto exato em que ele foi gerado, do que o descobrir dias depois, pois há valores errados em seu banco de dados!

Disparando exceções verificadas

Algumas exceções sinalizam problemas legítimos, que deveriam ser constantemente verificados pelo programador. Por exemplo, numa classe **ContaCorrente**, poderíamos querer sinalizar a possibilidade de não haver saldo no momento do saque – situação que tornaria o saque impossível.

Para esses casos, utilizamos as **exceções verificadas**, ou seja, àquelas que são filhas de ***exception***, mas não de ***RuntimeException***. Embora não seja uma regra absoluta, é comum também que as exceções verificadas sejam de tipos criados por nós, já que elas validam regras de negócio mais específicas, e não problemas comuns de programação.

Como implementaríamos então o método sacar? Veja o exemplo a seguir.

Execute o programa-exemplo.

Figura 11 – Exemplo: exceção verificada *SaldoInsuficienteException*

</>

```
1   public class SaldoInsuficienteException extends Exception {
2       public SaldoInsuficienteException(String msg) {
3           super(msg);
4       }
5   }
6
```

</>

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
```

```
15
16  public class ContaCorrente {
17      private double saldo;
18
19      public void sacar(double valor) throws
20  SaldoInsuficienteException {
21          if (saldo - valor <= 0) {
22              throw new
23  SaldoInsuficienteException(String.format(
24                  "O saldo %.2f é insuficiente para sacar o
25  valor %.2f",
26                  saldo, valor)
27              );
28          }
29          this.saldo -= valor;
30      }
31  }
32
```

Fonte: Autores (2021).

Observe que, inicialmente, fizemos a criação da classe ***SaldoInsuficienteException***, filha de ***exception***. Fizemos a chamada ao construtor da classe-pai para repassar a mensagem de erro.

Na classe **ContaCorrente**, iniciamos com o código de guarda, como fizemos no método **setIdade**, testando se o saldo seria suficiente para o saque. Caso não seja, criamos a exceção e a disparamos com o ***throws***. Porém, observe que tomamos o cuidado de gerar uma mensagem bastante descritiva. Isso nos auxiliará a entender em detalhes em que situação o problema ocorreu, caso ele venha a acontecer.

Observe também que há mais um detalhe nesse código: na declaração do método **sacar**, na linha 4, agora encontramos a cláusula ***throws*** e o nome da exceção. Isso indica que esse método dispara essa exceção verificada e que ela deve ser obrigatoriamente tratada. Nesse caso, o programador que chamar o método sacar tem duas opções:

- Adicionar um bloco ***try...catch*** que capture e trate a exceção, impedindo que ela se propague.
- Indicar que o método que usa o método sacar não trata essa exceção e, portanto, também pode dispará-la. Nesse caso, esse método também terá que conter a cláusula ***throws***.

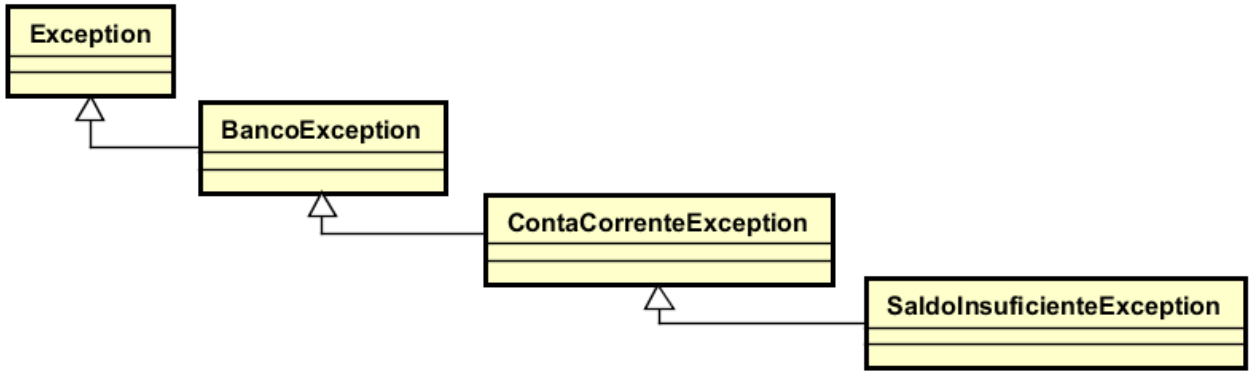
Isso tem um impacto importante: observe que, se uma nova exceção verificada for adicionada a um método, o Java exigirá que **todos** os pontos que chamam aquele método sejam modificados.

Felizmente, a cláusula ***throws*** também é hierárquica. Por isso, é interessante criarmos uma **hierarquia de exceções** em nosso sistema. Por exemplo, a exceção ***SaldoInsuficienteException*** poderia ser filha de ***ContaCorrenteException***, que por sua vez poderia ser filha de ***BancoException*** – essa sim, mãe de todas as exceções de nosso sistema. Poderíamos então reorganizar o método sacar para ter em sua cláusula ***throws*** uma ***ContaCorrenteException***, mesmo que seu throw interno ainda dispare a exceção específica.

Execute o programa-exemplo.

Hierarquia de exceções: a exceção ***SaldoInsuficienteException*** é filha de ***ContaCorrenteException***, que por sua vez é filha de ***BancoException*** – a mãe de todas as exceções de nosso sistema.

Figura 12 – Exemplo: hierarquia de exceções



</>

```
1
2
3
4
5
6
7
8
9
10
11
12
```

```
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
```

```
public class ContaCorrente {
    private double saldo;

    public void sacar(double valor) throws
ContaCorrenteException {
        if (saldo - valor <= 0) {
            throw new
SaldoInsuficienteException(String.format(
                "O saldo %.2f é insuficiente para sacar o
valor %.2f",
                saldo, valor)
            );
        }
        this.saldo -= valor;
    }
}
```

Fonte: Autores (2021).

Observe que isso faz bastante sentido. Normalmente, quem faz ***catch*** no método sacar irá querer pegar qualquer problema relacionado ao saque (saldo insuficiente, limite insuficiente, conta encerrada etc.). Ou seja, estamos indicando na cláusula ***throws*** que o ***ContaCorrenteException*** é o nível ideal para a captura. Porém, ainda fornecemos exceções mais específicas, para caso seja necessário tratar um dos problemas maneira especial. Da mesma forma, lembre-se: o programador que utiliza o método sacar ainda tem a liberdade de capturar uma exceção ainda mais alta (como ***BancoException***), caso queira, pois a cláusula ***catch*** também é hierárquica.

Organize cuidadosamente a hierarquia de classes de exceção do seu sistema.

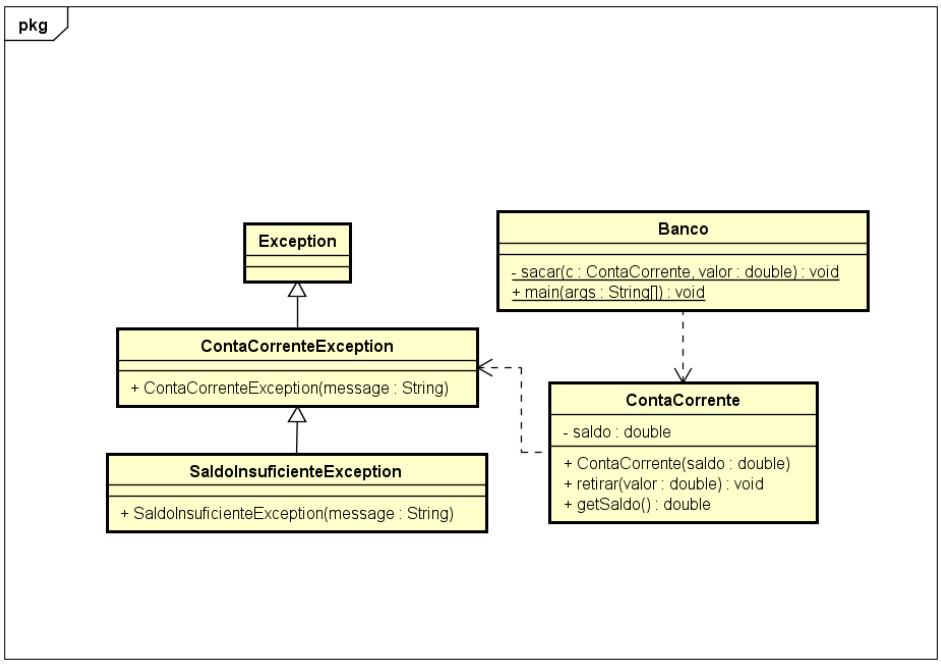
Vamos trabalhar mais nesse exemplo. Realize o exercício a seguir da Figura 13.

EXERCÍCIO

Exceção *SaldoInsuficienteException*

Observe o diagrama de classes do programa para tratamento de exceção verificada, que está em uma estrutura hierárquica.

Figura 13 – Exercício: exceção *SaldoInsuficienteException*



Execute o programa, verificando como é o seu funcionamento passo a passo (faça a depuração do código.o debug).

Responda:

1. Por que o método **retirar()** da classe **ContaCorrente** está declarado para lançar (`throws`) a exceção **ContaCorrenteException**, contudo em caso de erro, lança de fato (`throw new`) a exceção **SaldoInsuficienteException**?
2. Se o método **retirar()** da classe **ContaCorrente** executar o comando "`throw new SaldoInsuficienteException());`" qual será o valor do saldo da conta corrente após esse comando?
3. Na classe **banco**, na execução da linha 13, foi gerada exceção? Por quê?
4. Na classe **banco**, linha 23, de que classe é a instância de exceção "e" tratada?
5. No método ***main*** da classe banco, seria possível retirar os comando ***try-catch***? Por quê?
6. No método ***main*** da classe banco, os comando ***try-catch*** envolvem o método **sacar()**. Por quê?
7. Na classe **banco**, qual o valor apresentado pela execução do bloco `finally` da linha 24? Por que esse valor é apresentado?

</>

```
1
```

```
2
3   // Exceção mais genérica
4   public class ContaCorrenteException
5   extends Exception {
6       public ContaCorrenteException(String
7   message) {
8           super(message);
9       }
10  }
11
```

</>

```
1   // Exceção mais específica
2   public class SaldoInsuficienteException
3   extends ContaCorrenteException {
4       public
5   SaldoInsuficienteException(String message)
6   {
7           super(message);
8       }
9   }
10
```

</>

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
```

```
public class ContaCorrente {
    private double saldo;
    public ContaCorrente(double saldo) {
        this.saldo = saldo;
    }
      // Declaração do o método retirar,
que pode lançar uma exceção
      // mais genérica:
ContaCorrenteException
    public void retirar(double valor)
throws ContaCorrenteException {
        // Em erro específico, lança
exceção específica:
      // SaldoInsuficienteException, filha
de ContaCorrenteException
if (saldo - valor <= 0)
              throw new
SaldoInsuficienteException(String.format(
                    "O saldo R$ %.2f é
insuficiente para sacar o valor R$ %.2f",
                    saldo, valor)
                );
        else
            this.saldo -= valor;
    }
    public double getSaldo() {
        return saldo;
    }
}
```

</>

```
```

```
12
13  public class Banco {
14      // Método sacar DELEGA o tratamento de
15  exceção genérica
16         // (ContaCorrenteException), pois
17  não fará o seu tratamento
18      private static void sacar
19  (ContaCorrente c, double valor) throws
20         ContaCorrenteException {
21          c.retirar(valor);
22      }
23
24      public static void main(String[] args)
25  {
26          ContaCorrente cta = new
27  ContaCorrente(1000);
28
29          try {
30              sacar(cta,200);
31          } catch (ContaCorrenteException e)
32  {
33              System.out.println("Erro: " +
34  e.getMessage());
35          }finally {
36              System.out.println("Saldo
37  Conta: R$ " + cta.getSaldo());
38          }
39
40          try {
41              cta.retirar(2000);
42          } catch (ContaCorrenteException e)
43  {
44
45  System.out.println(e.getMessage());
46          }finally {
47  System.out.println("Saldo Conta: R$ " +
48                  cta.getSaldo());
49          }
50      }
51  }
52
```

Fonte: Autores (2021).

Resolução do exercício

Gabarito:

1. Por que o método **retirar()** da classe **ContaCorrente** está declarado para lançar (**throws**) a exceção **ContaCorrenteException**, contudo em caso de erro, lança de fato (**throw new**) a exceção **SaldoInsuficienteException**?

Resp.: como **SaldoInsuficienteException** é herdeira de **ContaCorrenteException**, a declaração do método **retirar()** indica que ele pode lançar uma exceção mais genérica: **ContaCorrenteException**. Contudo, quando há um erro específico, o método pode lançar uma exceção específica, herdeira de **ContaCorrenteException**, como é o caso da exceção **SaldoInsuficienteException**.

2. Se o método **retirar()** da classe **ContaCorrente** executar o comando "**throw new SaldoInsuficienteException());**" qual será o valor do saldo da conta corrente desse comando?

Resp.: se houver o lançamento da exceção, a retirada não ocorre e o saldo mantém o valor.

3. Na classe banco, na execução da linha 13, foi gerada exceção? Por quê?

Resp.: não foi gerada nem lançada exceção, pois o saldo (R$1000,00) era suficiente para a retirada solicitada (R$200,00).

4. Na classe banco, linha 23, de que classe é a instância de exceção "**e**" tratada?

Resp.: mesmo sendo catch (**ContaCorrenteException**), a exceção tratada de fato é a **SaldoInsuficienteException**, herdeira de **ContaCorrenteException**, lançada pelo método **retirar()**.

5. No método **main** da classe **banco**, seria possível retirar os comando **try-catch**? Por quê?

Resp.: Não, pois tanto o método **retirar()** quanto o **sacar()** lançam exceção verificada em tempo de compilação (gera erro de compilação). Ou seja, impõe a necessidade de tratamento com **try-catch**.

6. No método **main** da classe **banco**, os comando **try-catch** também envolvem o método **sacar()**. Por quê?

Resp.: Porque o método **sacar()** chama o método **retirar()**, o que obriga a **sacar()** a tratar (**try-catch**) ou a delegar (**trown**) a exceção lançada por **retirar()**.

7. Na classe banco, qual o valor apresentado pela execução do bloco **finally** da linha 24? Por que esse valor é apresentado?

Resp.: O valor é de R$ 800,00, que é o valor anterior à operação de retirar(), não é executada, uma vez que o saldo é insuficiente para uma retirada de R$ 2.000,00.

Testes unitários

Obviamente, queremos que nossos sistemas sejam robustos e livres de erros. Podemos utilizar o mecanismo de captura de erros a nosso favor, testando por potenciais problemas ou validando situações em vários pontos do código. De fato, há ferramentas inteiras dedicadas aos testes de sistemas, como o **JUnit**. A linguagem Java fornece uma instrução simples com o este objetivo, chamada ***assert***.

O comando ***assert*** também testará uma condição e **disparará uma exceção**, similar ao que ocorre com os códigos que geramos até agora. Porém, há uma diferença: ele só fará isso caso passemos um comando específico para a JVM na hora de executar nossa aplicação. Veja como ativar as exceções do *assert*, habilitando essa verificação na JVM (no Eclipse e no IntelliJ).

Comando ***assert***: **apenas dispara exceção** caso passemos um parâmetro específico na hora de executar nossa aplicação na **Virtual Machine** (**VM**).

- **No Eclipse**: botão direito sobre classe → *Run As* → *Run Configurations*... → *Arguments* → VM *Arguments*: **-ea**.
- **No IntelliJ**: *Run* → *Edit Configurations* → *Application* → VM *options*: **-ea**.

Então, para que utilizamos? Para validar pressupostos de programação, ou seja, para escrever testes que validem nosso próprio código (**testes unitários**), em uma destas situações:

1. **Pré-condições**: situações que devem ser verdadeiras quando o método foi invocado.
2. **Pós-condições**: situações que devem ser verdadeiras quando o método foi finalizado.
3. **Invariantes**: situações que permanecem inalteradas em todo programa. Geralmente, testada ao fim de métodos públicos.

Que tal observarmos um exemplo prático?

Figura 14 – Exemplo: carrinho de compras com asserções

```
</>
```

```
22
23  class CarrinhoDeCompras {
24      private static int maximo = 10;
25      private static int quantidade = 0;
26      private static int inseridos = 0;
27      private static int removidos = 0;
28      private static double preco_unitario = 10.00;
29
30      public static void inserir()
31      {
32          assert (quantidade < maximo); //PRE-CONDICAO
33          quantidade++;
34          inseridos++;
35          assert (quantidade == inseridos - removidos); //
36  INVARIANTE
37      }
38
39      public static void remover()
40      {
41          assert(quantidade > 0); // PRE-CONDICAO
42          quantidade--;
43          removidos++;
44          assert (quantidade >= 0); //POS-CONDICAO
45          assert (quantidade == inseridos - removidos); //
46  INVARIANTE
47      }
48
```

Fonte: Autores (2021).

Esse código possui as seguintes asserções:

- **Pré-condições**: na inserção, o carrinho não deveria ter mais itens do que a quantidade máxima. Na remoção, há a necessidade de haver pelo menos um item.
- **Pós-condições**: o método remover jamais poderá deixar o carrinho com uma quantidade negativa de itens. Isso só ocorrerá na presença de um erro de programação. Por isso, deixamos esse pressuposto explícito.
- **Invariantes**: observe que indiferentemente se inserimos ou removemos, a quantidade total de itens do carrinho precisará ser igual à quantidade de itens inseridos subtraído dos itens removidos. Testes de invariantes são bastante úteis quando testamos redundâncias de informação em nossas classes, ou seja, informações que podem ser obtidas de duas formas diferentes – como no exemplo.

Você pode estar se perguntando: no caso das pré-condições, não seria melhor utilizar exceções no lugar das pré-condições?

A resposta é: depende. Quando analisar se deve ou não utilizar uma asserção, questione:

1. O que está sendo testado é uma regra de negócio? Ou você está se prevenindo como programador de um erro seu? Se for o primeiro caso, prefira exceções.
2. O que está sendo testado é uma situação comum? Ou é algo que só ocorrerá no caso de bugs? Se for o primeiro caso, prefira exceções.
3. Se você remover o teste, valores incorretos poderão ser inseridos no sistema? Se a resposta for positiva, prefira exceções.

Refletindo a respeito, o que você deduz sobre o carrinho de compras? Sim, o sistema estaria mais bem modelado com exceções e códigos de guarda nessas condições.

Mas voltemos ao caso do método **setIdade**, visto no exemplo da Figura 10, item “disparando exceções não verificadas”. Obviamente, uma idade negativa é um problema, mas será que deveríamos testar a idade máxima com tanto rigor? Podemos usar uma **asserção para sinalizar uma situação estranha**, enquanto mantemos o teste para o que certamente é o fruto de um problema.

Figura 15 – Exemplo: Método setIdade com asserção

```
</>
```

```
16
17  public class Pessoa {
18      private int idade;
19
20      public void setIdade(int idade) {
21          //PRÉ-CONDIÇÃO
22          assert(idade < 80);
23
24          if (idade <= 0 || idade >= 120) {
25              throw new IllegalArgumentException("Idade
26  inválida: " + idade);
27          }
28          this.idade = idade;
29      }
30  }
31
```

Fonte: Autores (2021).

Observe que a exceção se manteve, mas com um parâmetro **menos** rigoroso do que a asserção. Afinal, a pessoa mais velha do mundo em 2021, fez 118 anos em janeiro – mais do que isso, obviamente, é um erro de cadastro. Menos, pode até ser uma situação possível, embora pouco provável. Se não há uma regra de negócios explicitamente mapeada indo contra essa situação, esse cadastro **deveria** ser possível.

Por outro lado, a asserção está testando um pré-condição que o time de programação considerou quando elaborou esse sistema: tal situação é tão improvável que eles acham que não irá acontecer. Isso pode ter implicações importantes, como uma caixa de textos do sistema não ser larga o suficiente para um usuário com três dígitos em sua idade, ou mesmo não ter uma opção para data tão antiga caso um calendário seja apresentado.

Isso permitirá que o sistema sinalize um problema no desenvolvimento, se futuramente essa regra for violada.

Pratique, no exercício da Figura 16, os conceitos de asserção: como ativar essa exceção e situações de uso.

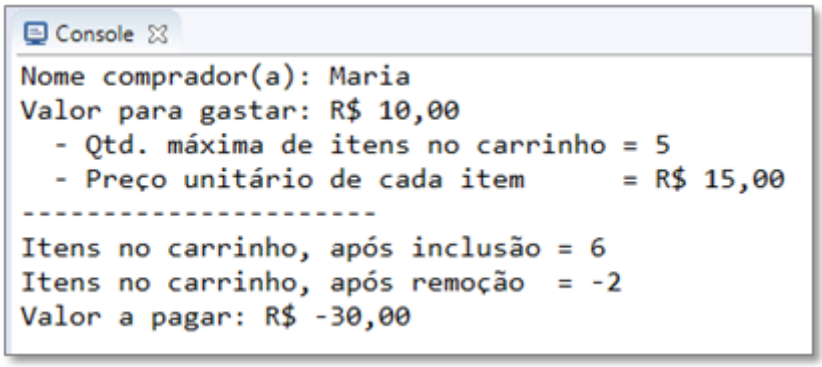
EXERCÍCIO

Asserção

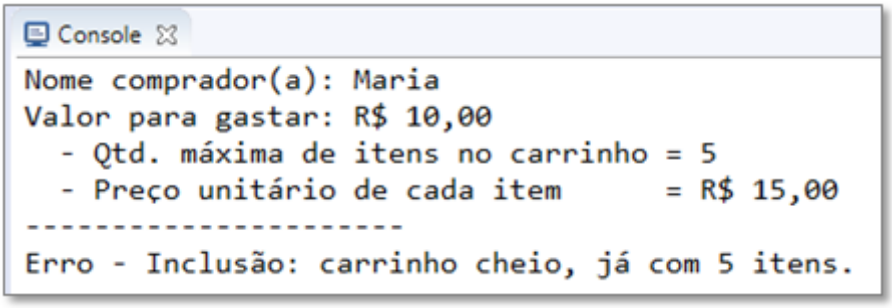
Execute o programa a seguir, para verificar as situações.

1. O programa original **não aponta erro**, pois a **JVM não foi habilitada** para pegar erros de **assert**: observe o valor a pagar negativo e o total de itens no carrinho também negativo:

Figura 16 – Exercício: asserção

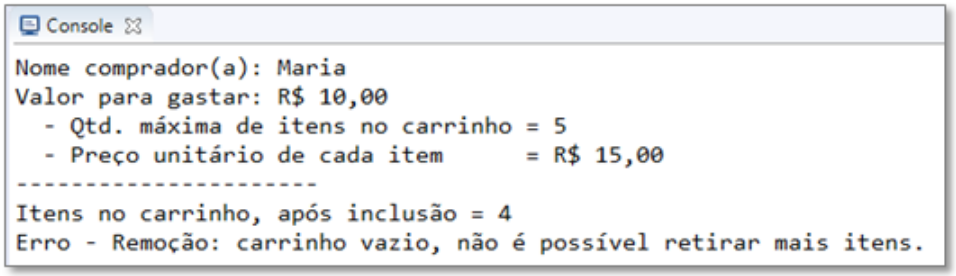


2. **Habilite a JVM** para pegar erros de **assert.** Agora é possível verificar os erros pegos pelo **assert**: veja que tentamos incluir seis itens, contudo o carrinho comporta apenas cinco itens.

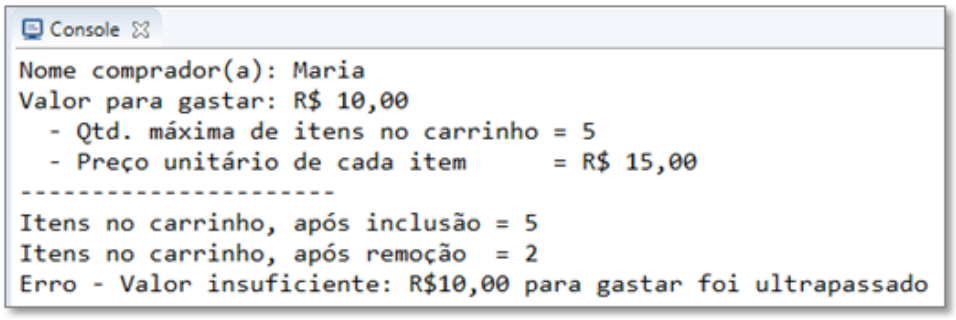


3. **Altere o código do programa** para obter os erros de:

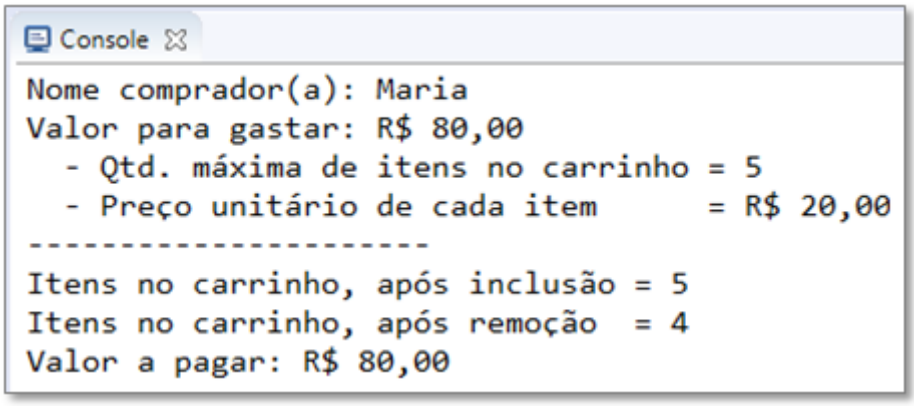
a. **Carrinho vazio**, quanto tentamos remover itens de um carrinho vazio.



b. **Valor insuficiente**, quando não há valor em carteira suficiente para pagar pelos itens do carrinho.



c. **Compra bem-sucedida**, quando todas as restrições são atendidas e Maria consegue levar os itens comprados no carrinho.




```
</>
```

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
```

```
22
23  package Carrinho;
24
25  public class Carrinho {
26      private int     qtdMaxima;
27      private int     quantidade = 0;
28      private int     inseridos = 0;
29      private int     removidos = 0;
30      private double  precoUnitario;
31
32      public Carrinho (int qtdMaxima, double
33  precoUnitario) {
34          this.qtdMaxima = qtdMaxima;
35          this.precoUnitario =
36  precoUnitario;
37      }
38      public void printCarrinho() {
39          System.out.println("  - Qtd.
40  máxima de itens no carrinho = " +
41  qtdMaxima);
42          System.out.printf ("  - Preço
43  unitário de cada item      = R$ %.2f\n",
44  precoUnitario);
45      }
46
47      public void inserir () {
48          assert  (quantidade < qtdMaxima ):
49  String.format("Inclusão: carrinho cheio,
50  já com %d itens.\n", quantidade); //PRE
51  CONDICAO
52          quantidade++;
53          inseridos++;
54          assert (quantidade == inseridos -
55  removidos ); // INVARIANTE
56      }
57      public void remover()
58      {
59          assert quantidade > 0 : "Remoção:
60  carrinho vazio, não é possível retirar
61  mais itens."; // PRE CONDICAO
62          quantidade--;
63          removidos++;
64          assert quantidade == inseridos -
65  removidos : "quantidade != inseridos -
66  removidos"; // INVARIANTE
```

```
    }
    public int getQuantidade () {
        return quantidade;
    }
    public double totalizar (double
precoMaximo) {
        double precoTotal = precoUnitario
* quantidade;
        assert (precoTotal <= precoMaximo
) : String.format("Valor insuficiente:
R$%.2f para gastar foi ultrapassado\n",
precoMaximo); // POS CONDICAO
        return precoTotal;
    }
}
```

</>

```
26
27  package Carrinho;
28
29  public class Comprador {
30      private String nome;
31      private double valorTotalCarteira;
32      private Carrinho carrinho;
33
34      public Comprador(String nome, double
35  valorTotalCarteira, int qtdCarrinho,
36  double precoUnitario) {
37          this.nome = nome;
38          this.valorTotalCarteira =
39  valorTotalCarteira;
40          this.carrinho = new
41  Carrinho(qtdCarrinho, precoUnitario);
42      }
43      public double getValorTotalCarteira()
44  {
45          return valorTotalCarteira;
46      }
47      public void printComprador() {
48          System.out.println("Nome
49  comprador(a): " + nome);
50          System.out.printf ("Valor para
51  gastar: R$ %.2f\n",
52
53  valorTotalCarteira);
54          carrinho.printCarrinho();
55          System.out.println("----------
56  --------");
57      }
58      public static void main(String[] args)
59  {
60          int totalInclusao = 6;
61          int totalRemocao  = 8;
62
63          Comprador maria = new Comprador
64  ("Maria", 10.0, 5, 15.0);
65          maria.printComprador();
66          try {
67              for(int i= 0; i< totalInclusao
68  ; i++) // insere
69
70  // "totalInclusao" de itens no carrinho
```

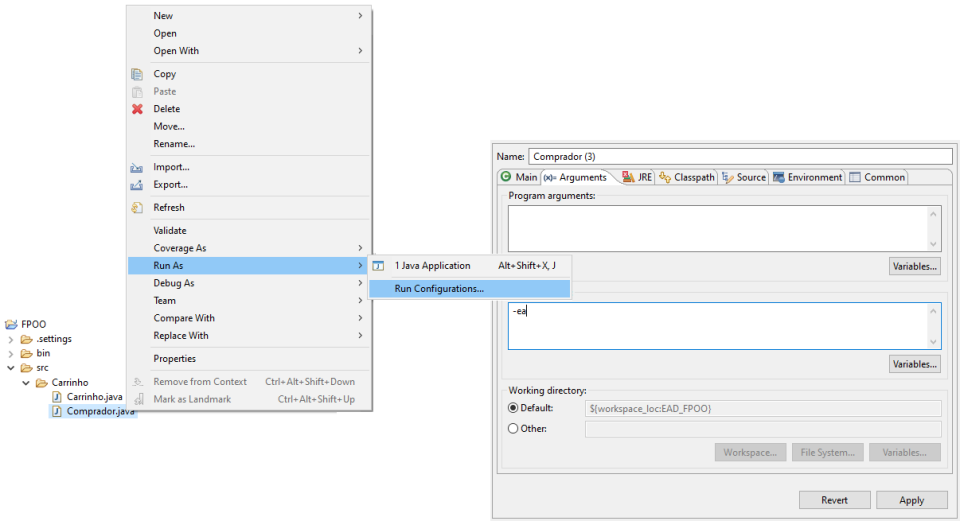
```
                maria.carrinho.inserir();
            System.out.println("Itens no
carrinho, após inclusão = " +

maria.carrinho.getQuantidade());
            for(int i= 0; i< totalRemocao
; i++) // insere

// "totalRemocao" de itens no carrinho
                maria.carrinho.remover();
            System.out.println("Itens no
carrinho, após remoção  = " +

maria.carrinho.getQuantidade());
            System.out.printf ("Valor a
pagar: R$ %.2f\n" ,

maria.carrinho.totalizar(maria.getValorTotalCar
        }catch(AssertionError e) {
            System.out.println("Erro - " +
e.getMessage());
        }
    }
}
```

Fonte: Autores (2021).

Resolução do exercício

Gabarito:

2. Habilite a JVM:



3. Altere o código do programa para obter os erros de:

a. Carrinho vazio: alterar na classe **comprador**.

```
</>
```

```
1   public static void main(String[] args){
2       int totalInclusao = 4;
3       int totalRemocao  = 8;
4   Comprador maria = new Comprador ("Maria",
5   10.0, 5, 15.0);
6   ...
7
```

b. Valor insuficiente: alterar na classe **comprador**.

```
</>
```

```
1   public static void main(String[] args){
2           int totalInclusao = 5;
3           int totalRemocao  = 3;
4           Comprador maria = new Comprador
5   ("Maria", 10.0, 5, 15.0);
6   ...
7
```

c. Compra bem-sucedida: alterar na classe **comprador**.

```
</>
```

```
1
2
3
```

```
4
5   public static void main(String[] args){
6       int totalInclusao = 5;
7       int totalRemocao  = 1;
8   Comprador maria = new Comprador ("Maria",
9   80.0, 5, 20.0);
10  ...
11
```

Referências

GODOY, V. **Programação orientada a objetos I**. Curitiba: IESDE, 2019.

HORSTMANN, C. S.; CORNELL, G. **Core Java – Volume I**. 8. ed. São Paulo: Pearson, 2010.

SCHILDT, H. **Java para iniciantes**. Porto Alegre: Bookman, 2015.