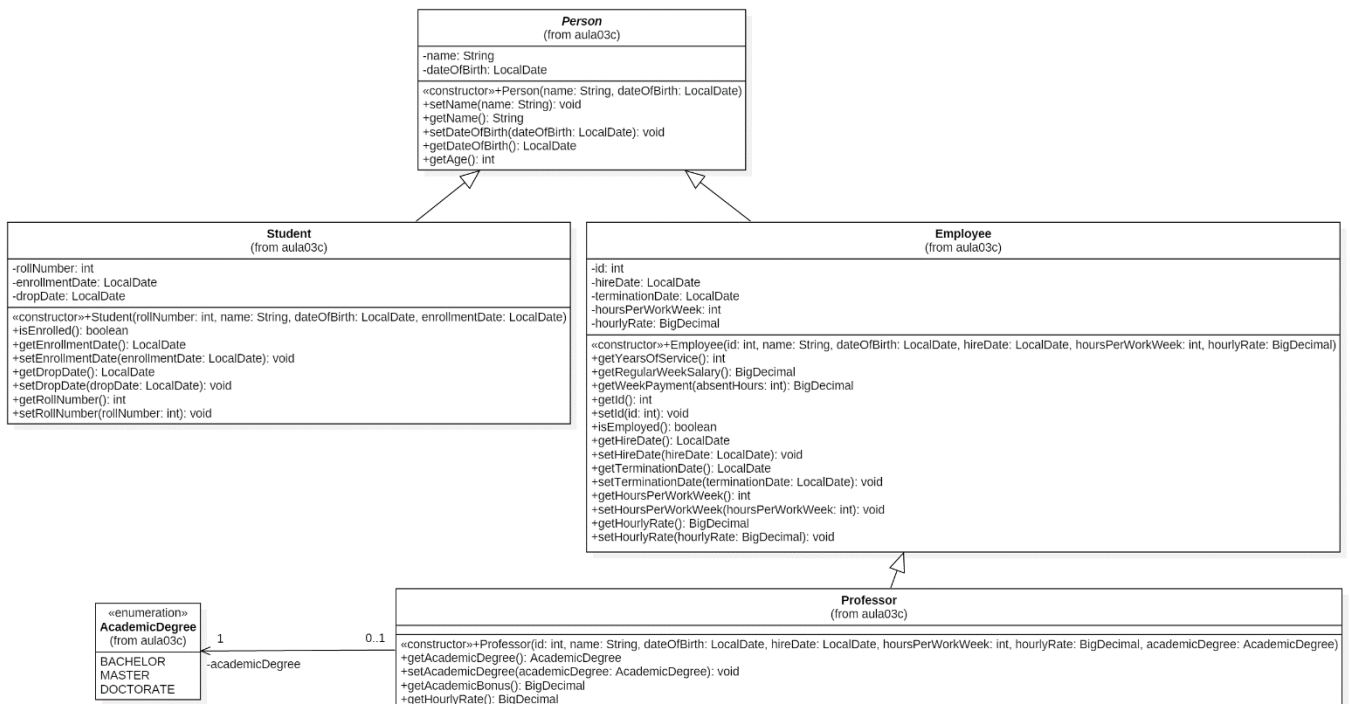


## Roteiro 04 – Atividade 01/06

1. Se você fez todas as atividades até aqui, seu diagrama de classe deve estar conforme a imagem abaixo. Note que a quantidade de construtores foi reduzida, uma vez que a maioria dos atributos foi considerada como obrigatória:



2. Você deve ter reparado que, até o momento, não tratamos a validade das informações fornecidas em construtores e em operações modificadoras (ex: *setters*). Na prática, precisamos rejeitar valores que podem deixar o estado do objeto inconsistente (o estado é o valor de um objeto, formado pela combinação dos valores dos atributos em dado momento). Além disso, precisamos também saber se tudo ocorreu como esperado durante a chamada de uma operação. Fazer isso fora da operação a cada chamada é uma solução pobre, pois quebra a ideia de coesão e responsabilidade (quem utiliza o objeto é que precisaria garantir se as informações estão corretas).

Uma alternativa para tratar erros é implementar operações que retornem um valor booleano (verdadeiro ou falso), caso a ação tenha sucedido ou falhado. Entretanto, nesta solução, nunca saberemos o que efetivamente ocorreu e a interpretação é deixada para o programador que está utilizando o código. Outra alternativa seria retornar valores numéricos que correspondem a diferentes tipos de erro<sup>1</sup>. Embora esta abordagem permita manipular uma quantidade maior de tipos de erros, ela exige documentação adicional para que seja possível interpretar aqueles números. Além disso, perderíamos o parâmetro de retorno da operação.

Como forma de lidar com estas limitações, a linguagem Java (e várias outras linguagens) utilizam o conceito de Exceção (*Exception*). **Uma exceção é um evento que ocorre durante a execução de um programa e que interrompe o fluxo normal de instruções.** O nome é bastante adequado, pois ela representa uma situação que normalmente não deveria ocorrer no programa. Na prática, você já vem trabalhando com exceções desde o início desta disciplina. Sempre que algum erro acontece em tempo de execução, o ambiente mostra uma exceção. Isso acontece porque ainda não aprendemos a tratar exceções.

Na última atividade do roteiro anterior (testes com redefinição de operações), quando invocamos a operação `foo()` dentro da própria operação `foo()`, ocorreu a exceção `StackOverflowError` (erro de estouro de pilha). Esta exceção ocorreu porque o método (implementação da operação) ficou recursivo (chamando-se a si mesmo), mas sem uma condição de parada. O programa foi empilhando as chamadas até que o espaço de memória reservada para a pilha de execução do programa estourou.

<sup>1</sup> Técnica conhecida por "Magic Number": [https://en.wikipedia.org/wiki/Magic\\_number\\_\(programming\)](https://en.wikipedia.org/wiki/Magic_number_(programming)).

## Roteiro 04 – Atividade 01/06

Para entender um pouco melhor como funciona o empilhamento das chamadas, implemente o programa abaixo e analise a saída.

```
public class CallStackTest {

    private void method1() {
        System.out.println("entrei em method1()");
        System.out.println("method1() --> method2(): empilha o endereço " +
            "de method1 e vai para o endereço de method2");
        method2();
        System.out.println("voltei para method1()");
    }

    private void method2() {
        System.out.println("entrei em method2()");
        System.out.println("method2() --> method3(): empilha o endereço " +
            "de method2 e vai para o endereço de method3");
        method3();
        System.out.println("voltei para method2()");
        System.out.println("desempilha o endereço de method1 e vai para " +
            "este endereço");
    }

    private void method3() {
        System.out.println("entrei em method3()");
        System.out.println("desempilha o endereço de method2 e vai para " +
            "este endereço");
    }

    public void run() {
        System.out.printf("\n\n\n***** CallStackTest ***** \n\n");
        method1();
    }

}
```

A saída do programa acima é apresentada na figura abaixo. Note que, quando estamos em `method3`, se não tivéssemos empilhado o endereço de `method2`, o programa não saberia para onde retornar. O mesmo acontece quando retornamos ao `method2`: como o programa saberia retornar para `method1`?

```
***** CallStackTest *****

entrei em method1()
method1() --> method2(): empilha o endereço de method1 e vai para o endereço de method2
entrei em method2()
method2() --> method3(): empilha o endereço de method2 e vai para o endereço de method3
entrei em method3()
desempilha o endereço de method2 e vai para este endereço
voltei para method2()
desempilha o endereço de method1 e vai para este endereço
voltei para method1()
```

## Roteiro 04 – Atividade 01/06

3. Agora, faça a seguinte alteração no programa anterior e execute-o novamente:

```
private void method3() {
    System.out.println("entrei em method3()");
    int divisao = 100 / 0;
    System.out.println(divisao);
    System.out.println("desempilha o endereço de method2 e vai para " +
        "este endereço");
}
```

Obviamente, a execução gerou uma exceção (dividir um número por zero é indefinido ou impossível). Logo, a saída foi algo como (obs: eu implementei este teste em uma nova classe chamada `ExceptionNotHandledTest`):

```
Exception in thread "main" java.lang.ArithmeticException: / by zero

***** ExceptionNotHandledTest *****

entrei em method1()
method1() --> method2(): empilha o endereço de method1 e vai para o endereço de method2
entrei em method2()
method2() --> method3(): empilha o endereço de method2 e vai para o endereço de method3
entrei em method3()
  at br.univali.kob.pool.p04a_exception_not_handled.ExceptionNotHandledTest.method3(ExceptionNotHandledTest.java:25)
  at br.univali.kob.pool.p04a_exception_not_handled.ExceptionNotHandledTest.method2(ExceptionNotHandledTest.java:17)
  at br.univali.kob.pool.p04a_exception_not_handled.ExceptionNotHandledTest.method1(ExceptionNotHandledTest.java:9)
  at br.univali.kob.pool.p04a_exception_not_handled.ExceptionNotHandledTest.run(ExceptionNotHandledTest.java:33)
  at br.univali.kob.pool._main.ClassMaterial2017.main(ClassMaterial2017.java:46)
```

Você notou a exceção em `main: java.lang.ArithmeticException: / by zero`? Lembre-se que `run()` está sendo invocado a partir de `main()`. Outro aspecto interessante é como a exceção está sendo mostrada. Como ainda não estamos fazendo nenhum tratamento, o ambiente mostra, por default, o caminho seguido pelo programa do ponto onde a exceção foi disparada até a chamada original (rastro da pilha de chamadas ou **stack trace**).

Analisando o **stack trace**, podemos ver que a exceção ocorreu na classe `ExceptionNotHandledTest` (arquivo "ExceptionNotHandledTest.java", linha 25). Como a exceção não foi tratada neste momento, ela foi propagada para `method2` (quem chamou `method3`). Da mesma forma, ela foi propagada para `method1`, então para `run` e por fim para `main` que está na classe `ClassMaterial2017` (arquivo "ClassMaterial2017.java", linha 46). Como o programa iniciou em `main` (não há mais nenhuma chamada para propagar a exceção), foi neste ponto que o erro foi mostrado.



Uma exceção é um evento excepcional que, ao ocorrer durante a execução de um programa, interrompe o fluxo normal das instruções do programa. Quando um erro ocorre durante a execução de um método, este método cria um objeto (chamado objeto de exceção) e o entrega para sistema de execução da JVM. O objeto de exceção contém informações sobre o erro, incluindo seu tipo e o estado do programa quando o erro ocorreu. Criar um objeto de exceção e entregá-lo para o sistema é chamado disparo uma exceção (*throwing an exception*).

Depois que o método dispara a exceção, o sistema de execução JVM tenta encontrar algum método que contenha um bloco de código para tratá-la. Esta busca é feita por meio da sequência reversa de chamadas que foram feitas para chegar no método onde o erro ocorreu (pilha de chamadas ou pilha de execução), iniciando por ele mesmo. O primeiro bloco de tratamento encontrado captura (catch) a exceção, mas se nenhum dos métodos verificados pelo sistema de execução JVM tiver um bloco de tratamento, então o programa é abortado. Capturar uma exceção significa retirá-la do sistema de execução JVM. O rastro gerado pela sequência de chamadas até a captura da exceção é chamado **stack trace**.

## Roteiro 04 – Atividade 01/06

4. Agora, faça a seguinte alteração no programa anterior e execute-o novamente:

```
private void method3() {
    System.out.println("entrei em method3()");
    try {
        int divisao = 100 / 0;
        System.out.println(divisao);
    } catch (java.lang.ArithmeticException ex) {
        System.out.println("***** mensagem default da exceção: "
            + ex.getMessage());
    }
    System.out.println("desempilha o endereço de method2 e vai para "
        + "este endereço");
}
```

Você entendeu o que aconteceu? Alguma sugestão para explicar o comportamento dos blocos `try-catch`? Note que a saída do programa deve ter sido algo como:

```
***** ExceptionHandledTest1 *****

entrei em method1()
method1() --> method2(): empilha o endereço de method1 e vai para o endereço de method2
entrei em method2()
method2() --> method3(): empilha o endereço de method2 e vai para o endereço de method3
entrei em method3()
***** mensagem default da exceção: / by zero
desempilha o endereço de method2 e vai para este endereço
voltei para method2()
desempilha o endereço de method1 e vai para este endereço
voltei para method1()
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

O programa foi executado sem ter sido abortado. Entretanto, a exceção ainda aconteceu uma vez que o defeito proposital no código não foi retirado. O programa não abortou desta vez porque tratamos a exceção com os blocos `try-catch`. O código que estiver dentro do bloco `try` é executado. Se ocorrer alguma exceção, o controle é desviado para o código do bloco `catch`. Note que dentro do bloco `catch`, nós precisamos explicitar a exceção esperada que será capturada (`java.lang.ArithmeticException ex`). Ao invés do *namespace* completo aqui, você pode utilizar a cláusula `import` e deixar apenas o nome simples da exceção. Como a exceção ocorreu, o programa foi desviado para dentro do bloco `catch` que apenas mostrou a mensagem de erro *default* da exceção (não o **stack trace**). O programa então continua sem problemas (próxima instrução logo após o bloco `catch`), uma vez que a exceção já foi tratada.



O exemplo anterior nos ajuda a entender como a linguagem Java trata as exceções. Entretanto, observe que, tipicamente, quando este tipo de exceção ocorre, o programa deveria ser abortado. Uma vez que o defeito está no código, a exceção será disparada sempre que o método for chamado. Logo, a boa prática é corrigir o defeito.

## Roteiro 04 – Atividade 01/06

5. Experimente trocar a exceção no bloco `catch` para `java.lang.NullPointerException` e execute novamente o programa. O que aconteceu?

O programa foi abortado porque a exceção tratada (`java.lang.NullPointerException`) nunca aconteceu. Esta exceção é disparada quando você tentar acessar um objeto que ainda não foi instanciado (valor `null`). Se a exceção ocorrida estiver sendo considerada pelo bloco `catch`, o código de tratamento da exceção é executado. Caso contrário, a exceção é repassada ao ambiente de execução (no nosso exemplo, ninguém mais tratou a exceção e ela foi parar no método `main`). Se não ocorrer uma exceção, o tratamento da exceção (`catch`) não é executado.

Agora mantenha a exceção errada em `method3` e faça a seguinte alteração em `method1`:

```
private void method1() {
    System.out.println("entrei em method1()");
    System.out.println("method1() --> method2(): empilha o endereço " +
        "de method1 e vai para o endereço de method2");
    try {
        method2();
    } catch (java.lang.ArithmeticException ex) {
        System.out.println("***** capturei a exceção em method1");
        System.out.println("***** mensagem default da exceção: "
            + ex.getMessage());
    }
    System.out.println("voltei para method1()");
}
```

Execute novamente o programa e veja o que acontece. A saída do programa deve ter sido algo como:

```
***** ExceptionHandledTest2 *****

entrei em method1()
method1() --> method2(): empilha o endereço de method1 e vai para o endereço de method2
entrei em method2()
method2() --> method3(): empilha o endereço de method2 e vai para o endereço de method3
entrei em method3()
***** capturei a exceção em method1
***** mensagem default da exceção: / by zero
voltei para method1()
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

Note que a exceção foi tratada, mas apenas quando ela foi propagada para `method1`. A exceção foi disparada no momento em que a linha da divisão foi executada (`method3`). Entretanto, os blocos `try-catch` de `method3` não consideraram a exceção `ArithmeticException`. Logo, ela foi propagada para `method2`, o qual havia chamado `method3`. Como não há tratamento de exceção em `method2`, a exceção foi então propagada para `method1`, onde foi capturada e tratada. Compare este resultado com o código e veja que o programa só retornou à sua execução normal após o tratamento em `method1`.



## Roteiro 04 – Atividade 01/06

6. Altere o `method3`, incluindo o bloco `finally` abaixo. Execute o programa e analise a saída.

```
private void method3() {
    System.out.println("entrei em method3()");
    try {
        int divisao = 100 / 0;
        System.out.println(divisao);
    } catch (java.lang.NullPointerException ex) {
        System.out.println("***** mensagem default da exceção: "
            + ex.getMessage());
    } finally {
        System.out.println("##### a parte finally é sempre executada.");
    }
    System.out.println("desempilha o endereço de method2 e vai para "
        + "este endereço");
}
```

Veja que que a instrução dentro do bloco `finally` foi executada, independentemente da exceção ter ocorrido. O código controlado pelo bloco `try` é executado. Se ocorrer uma exceção, o controle é desviado para o bloco `catch` que avalia se a exceção é para ser tratada. Independentemente do programa ter entrado ou não no bloco `catch`, o código do bloco `finally` é executado. Note que se a exceção não foi tratada agora, ela continua a ser propagada após a execução do bloco `finally`. Experimente também alterar para a exceção correta e executar o programa novamente.

Para confirmar que você entendeu, altere a linha da divisão para: `int divisao = 100 / 3;`, execute o programa e veja que o bloco `finally` continua sendo executado. Desta forma, você utilizar o `finally` sempre que for necessário algum tratamento obrigatório (tenha ou não ocorrido uma exceção). O bloco `finally` não exige que tenha sido definido um bloco `catch`. Você pode utilizar apenas `try` e `finally` em algum trecho de código. A lógica da sua solução é que irá ditar a estrutura mais adequada a ser utilizada.

Você pode ter mais de um bloco `catch` em um mesmo tratamento de exceção, permitindo que a lógica possa ser diferente para cada uma (veja imagem abaixo).

```
private void method3() {
    System.out.println("entrei em method3()");
    try {
        int divisao = 100 / 0;
        System.out.println(divisao);
    } catch (java.lang.NullPointerException ex) {
        System.out.println("***** mensagem default da exceção: "
            + ex.getClass() + ": " + ex.getMessage());
    } catch (java.lang.ArithmeticException ex) {
        System.out.println("***** mensagem default da exceção: "
            + ex.getClass() + ": " + ex.getMessage());
    } finally {
        System.out.println("##### a parte finally é sempre executada.");
    }
    System.out.println("desempilha o endereço de method2 e vai para "
        + "este endereço");
}
```

Mas ela nunca será disparada, pois não utilizamos objetos no bloco `try`.

## Roteiro 04 – Atividade 01/06

Ou ainda tratar múltiplas exceções em um único `catch`, caso o tratamento seja o mesmo para todas elas:

```
private void method3() {
    System.out.println("entrei em method3()");
    try {
        int divisao = 100 / 0;
        System.out.println(divisao);
    } catch (java.lang.NullPointerException | java.lang.ArithmeticException ex) {
        System.out.println("***** mensagem default da exceção: "
            + ex.getClass() + ": " + ex.getMessage());
    } finally {
        System.out.println("##### a parte finally é sempre executada.");
    }
    System.out.println("desempilha o endereço de method2 e vai para "
        + "este endereço");
}
```

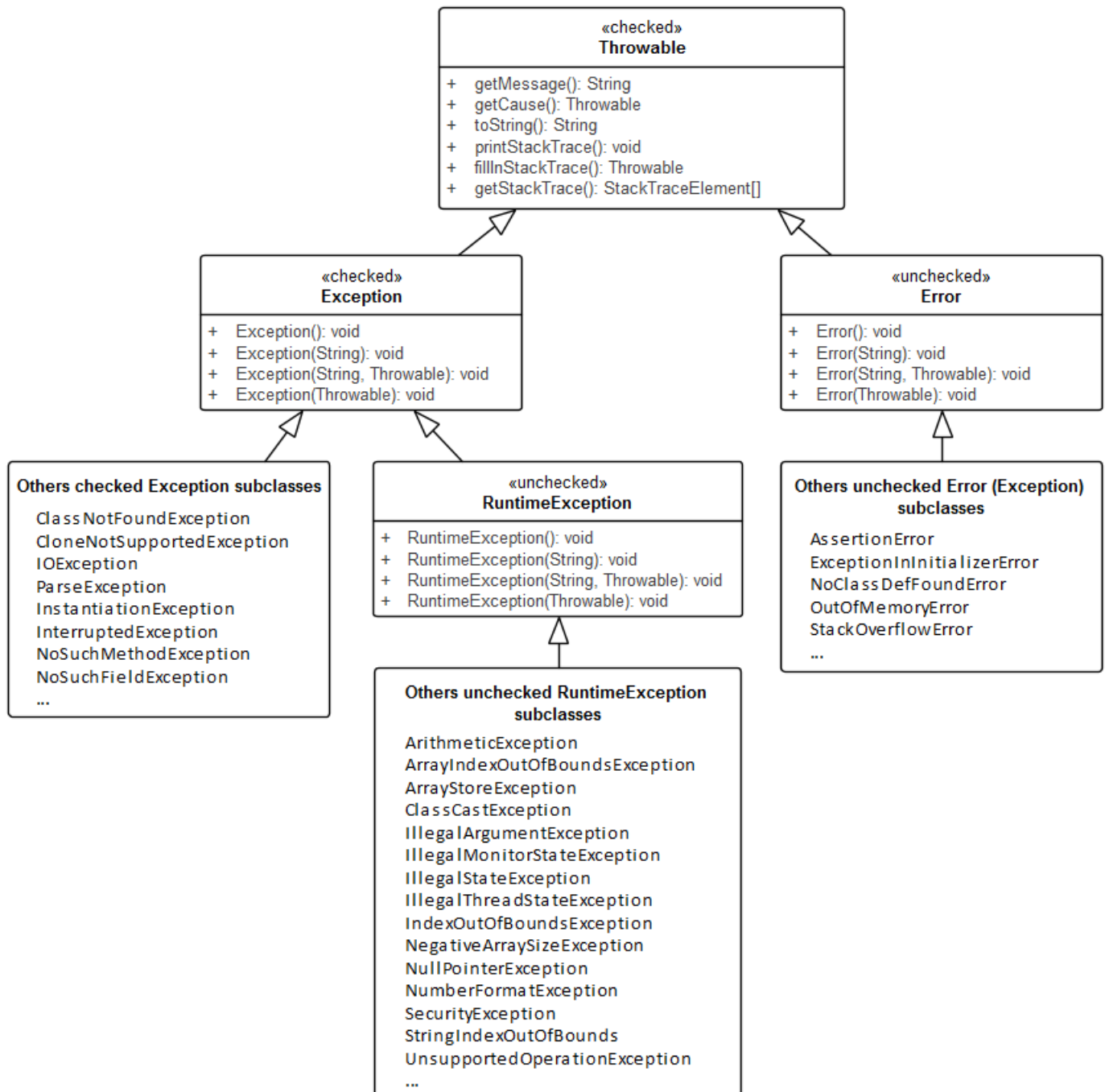
7. Agora, altere o `method3` novamente (imagem abaixo), execute o programa e analise a saída.

```
private void method3() {
    System.out.println("entrei em method3()");
    try {
        int divisao = 100 / 0;
        System.out.println(divisao);
    } catch (java.lang.RuntimeException ex) {
        System.out.println("***** mensagem default da exceção: "
            + ex.getClass() + ": " + ex.getMessage());
    }
    System.out.println("desempilha o endereço de method2 e vai para "
        + "este endereço");
}
```

Note que a exceção foi tratada pelo `method3`, ou seja, o programa considerou que `ArithmeticException` é uma `RuntimeException`. Isso aconteceu porque `ArithmeticException` é realmente uma (subclasse de) `RunTimeException`. Entretanto, esta não é uma boa prática, uma vez que a classe `RunTimeException` é muito genérica. Você deve sempre utilizar a exceção específica a ser tratada. Evite disparar ou capturar exceções `RunTimeException`. A próxima seção apresenta a hierarquia das classes de exceção da linguagem Java.

## Roteiro 04 – Atividade 02/06

1. A linguagem Java possui três tipos gerais de classe para tratar exceções: exceções checadas (*checked exceptions*), exceções não checadas (*unchecked exceptions*) e erros (*errors*). No diagrama de classe apresentado, podemos ver as quatro classes que formam a base para o tratamento de exceções em Java: **Throwable** (checked), **Exception** (checked), **RuntimeException** (unchecked) e **Error** (unchecked). Para maior legibilidade, são apresentadas apenas as principais operações de cada classe. Note também que a listagem das subclasses não segue a notação UML por questões didáticas. Em termos práticos raramente utilizamos a classe **Throwable** diretamente.



Para mais detalhes sobre as classes de exceção e suas respectivas operações, veja a documentação javadoc disponibilizada pela Oracle.

- <http://docs.oracle.com/javase/8/docs/api/java/lang/Throwable.html>
- <http://docs.oracle.com/javase/8/docs/api/java/lang/Exception.html>
- <http://docs.oracle.com/javase/8/docs/api/java/lang/RuntimeException.html>
- <http://docs.oracle.com/javase/8/docs/api/java/lang/Error.html>



## Roteiro 04 – Atividade 02/06

**Exceções checadas (*checked exception*)** tem este nome porque elas são checadas em tempo de compilação. Ou seja, se um método dispara uma exceção checada então ele deve tratar a exceção utilizando `try-catch` ou declará-la utilizando a palavra reservada `throws`. Caso contrário o programa apresentará um erro de compilação. Por exemplo, considere que um método construtor que lê um arquivo texto e monta uma lista de *strings* em memória (figura abaixo). Repare que foram utilizadas duas classes `FileReader` e `BufferedReader` (ambas do pacote `java.io`).

```
public TextFile(String fileName) throws FileNotFoundException, IOException {
    String line;
    BufferedReader reader = new BufferedReader(new FileReader(fileName));
    try {
        while ((line = reader.readLine()) != null) {
            lines.add(line);
        }
    } finally {
        reader.close();
    }
}
```

O construtor da classe `FileReader` pode disparar uma exceção checada `FileNotFoundException` se não existir um arquivo com o nome fornecido, se o nome for referente a um diretório (e não a um arquivo) ou se por algum outro motivo, o arquivo não puder ser aberto para leitura. Como nosso construtor chama este método, temos duas alternativas: utilizar `try-catch(FileNotFoundException)` ou apenas declarar a exceção na assinatura da operação com a cláusula `throws` (se a exceção ocorrer, ela será propagada automaticamente para o método que invocou este construtor). Optamos pela segunda porque não há nada a ser feito com a exceção neste momento. Repare que o programador que utilizar a classe `TextFile` terá que tomar a mesma decisão. O método `read()` da classe `BufferedReader` pode disparar uma exceção checada `IOException` quando houver algum erro de leitura/escrita (IO = input/output). Deste modo, também precisamos adicionar a exceção na lista de exceções que podem ser disparadas ou propagadas pelo nosso construtor.

Veja um exemplo de como chamar o construtor da classe `TextFile`. Como este método optou por tratar as exceções, ele deve garantir que todas as exceções declaradas na cláusula `throws` do construtor tenham sido tratadas. Note também que `FileNotFoundException` aparece primeiro. Isso porque, ela é uma subclasse de `IOException`. Deste modo, se a ordem fosse invertida, o tratamento para arquivo inexistente nunca seria executado. O próprio compilador já nos avisa deste erro.

```
private void textFileTest() {
    try {
        TextFile text = new TextFile("C:\\Users\\marcello\\Student.html");
        List<String> lines = text.getAllLines();
        for (String line : lines) {
            System.out.println(line);
        }
    } catch (FileNotFoundException ex) {
        System.out.println("Nome de arquivo inválido ou inexistente.");
    } catch (IOException ex) {
        System.out.println("Não foi possível ler o arquivo.");
    }
}
```

Note também as mensagens apresentadas no tratamento das exceções. Quando o objetivo é apresentar um erro para o usuário final, o texto da mensagem deve estar em conformidade com o negócio (texto amigável). Mensagens técnicas demais, com trechos de código ou que representam problemas em baixo nível são, tipicamente, inadequadas para serem interpretadas por um usuário final.

## Roteiro 04 – Atividade 02/06

As exceções checadas representam condições inválidas em áreas fora do controle imediato do programa (problemas de entradas do usuário inválidas, banco de dados, falhas de rede, arquivos inexistentes), mas que um programa bem escrito deveria antecipar. Todas as exceções checadas são subclasses de `Exception` (mas observe que `RuntimeException` e todas as suas subclasses não são checadas).

**Exceções não checadas (*unchecked exception*)** não são checadas em tempo de compilação, apenas em tempo de execução. Ou seja, se um método dispara uma exceção não checada, nunca haverá erro de compilação, mesmo quando esta exceção não for tratada pelo programa (não é obrigatório que ela seja declarada com `throws`, nem que ela seja capturada nos blocos `try-catch`). Tipicamente, esta exceção ocorre por defeitos de codificação (ex: valores inválidos para argumentos, operações aritméticas inválidas). Desta forma, é comum que ao ocorrer uma exceção não checada, o programa deveria ser abortado e o erro corrigido. Cabe ao programador julgar as condições antecipadamente, o que pode causar tais exceções e tratá-las apropriadamente. Nos passos anteriores, trabalhamos com as exceções `ArithmeticException` e `NullPointerException`. Revise os programas para confirmar que não declaramos estas exceções com `throws` e nem fomos obrigados a capturar a exceção em algum momento. Note também que estas exceções acontecem usualmente por defeitos de programação. Todas as exceções não checadas são subclasses da classe `RuntimeException` ou da classe `Error`.

**Error e suas subclasses** são exceções não checadas que representam condições excepcionais "externas" a uma aplicação e que não podem ser antecipadas. Tipicamente, a aplicação não tem como se recuperar destas condições (ex: falha de hardware). Entretanto, é possível implementar um tratamento para a exceção. No roteiro anterior, tivemos a exceção `StackOverflowError`. Note que não temos o que fazer quando esta exceção é disparada. Ou seja, deveríamos registrar a ocorrência no log e o programa deveria ser abortado.

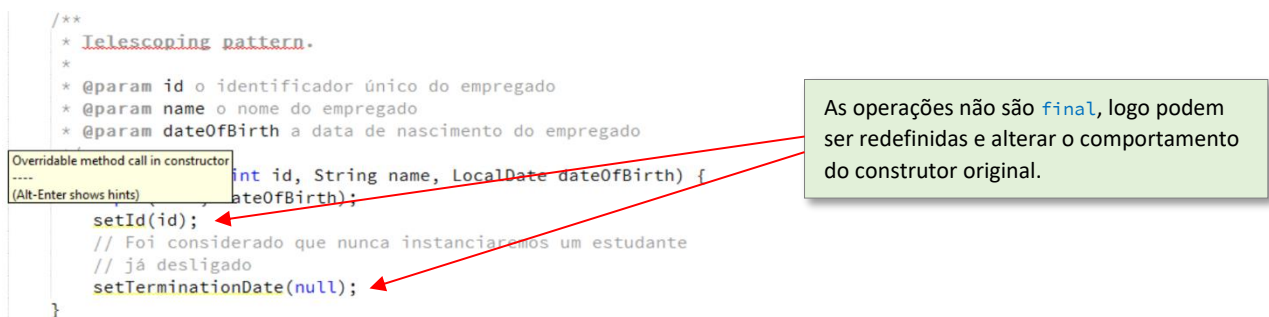
## Roteiro 04 – Atividade 03/06

- Até agora, aprendemos como tratar exceções. Entretanto, durante a programação de determinados métodos, nós também precisamos indicar que algo está errado. Por exemplo, considere que o método `setHoursPerWorkWeek(int)` da classe `Employee` deve aceitar somente valores entre 1 e 40. Atualmente, nossa implementação não trata esta regra de negócio e qualquer valor que seja diferente desta faixa colocará o objeto em um estado inválido (inconsistente). Uma forma de resolver este problema é verificar a regra no momento em que o valor está sendo modificado. Algo como:

```
/**
 * @param hoursPerWorkWeek a quantidade de horas contratadas por semana de trabalho
 *
 * @throws IllegalArgumentException se a quantidade de horas não estiver dentro da faixa esperada
 */
public void setHoursPerWorkWeek(int hoursPerWorkWeek) {
    if (hoursPerWorkWeek < 1 || hoursPerWorkWeek > 40) {
        throw new IllegalArgumentException("Hours per Work Week (" + hoursPerWorkWeek +
            ") is out of range [1..40]");
    }
    this.hoursPerWorkWeek = hoursPerWorkWeek;
}
```

Note que a validação é realizada antes de modificar o valor do atributo. Desta forma, se ocorrer uma exceção durante a validação, o objeto não ficará inconsistente (caso alguém capture e trate a exceção). Para implementar a validação, utilizamos a exceção não checada `IllegalArgumentException` (o objetivo aqui é identificar defeitos de programação). Ela é adequada em situações em que algum dos argumentos passados para o método é inválido. Para forçar que a exceção seja disparada quando um erro for detectado, utilizamos a cláusula `throw`. Note que seu diagrama de classe não foi alterado, pois a mudança foi apenas no método (implementação da operação que não teve sua assinatura modificada).

Realizar a validação no **setter** parece conveniente para nosso padrão **Telescoping**. Uma vez que utilizamos **setters** internamente para inicializar os valores no construtor, a validação também está garantida na instanciação do objeto. Entretanto, você deve lembrar da questão levantada nos roteiros anteriores sobre o problema de invocarmos operações que podem ser redefinidas dentro de um construtor.



The screenshot shows the constructor of the `Employee` class. It includes Javadoc comments for parameters `id`, `name`, and `dateOfBirth`. The constructor body calls `setId(id)`, `setTerminationDate(null)`, and `setHoursPerWorkWeek(hoursPerWorkWeek)`. Annotations include "Telescoping pattern.", "Overrideable method call in constructor", and "(Alt-Enter shows hints)". A callout box points to the `setId(id)` and `setTerminationDate(null)` calls, stating: "As operações não são final, logo podem ser redefinidas e alterar o comportamento do construtor original."

```
/**
 * Telescoping pattern.
 *
 * @param id o identificador único do empregado
 * @param name o nome do empregado
 * @param dateOfBirth a data de nascimento do empregado
 */
Employee(int id, String name, LocalDate dateOfBirth) {
    setId(id);
    // Foi considerado que nunca instanciaríamos um estudante
    // já desligado
    setTerminationDate(null);
}
```

Na prática, devemos evitar invocar operações sem o modificador `final` dentro de um construtor, pois ao redefinir alguma destas operações, o novo método pode trazer resultados inesperados. Também avalie se um `setter` deve poder ser mesmo redefinido (não `final`). Desta forma, devemos utilizar `setters` dentro de um construtor somente quando o `setter` for declarado como `final`. Em uma solução adequada de herança, dificilmente precisaremos redefinir um `setter` e suas validações. Caso contrário, é possível que a subclasse não seja realmente uma boa subclasse para sua hierarquia. Também devemos evitar de passar um argumento `null` para uma operação (veja a última linha do construtor mostrado anteriormente). Seria mais adequado inicializarmos o atributo no construtor e garantir que em um `setter`, ele receba um valor válido.

No construtor, podemos então inicializar cada atributo com os argumentos recebidos sem utilizar operações. Ao final, podemos chamar uma operação privada que valida o estado do objeto, tal como `validateState()`. Esta operação pode invocar operações de validação para cada atributo utilizando uma ordem definida pela lógica de negócio. Cada `setter`, por sua vez, precisaria chamar esta operação. Embora, todos os atributos sejam testados a

## Roteiro 04 – Atividade 03/06

cada *setter*, isso pode ser desejável. Por exemplo, quando a data de demissão é alterada, precisamos verificar se ela está consistente com a data de admissão. Veja como ficaria a nossa classe **Person**:

```
public Person(String name, LocalDate dateOfBirth) {
    this.name = name;
    this.dateOfBirth = dateOfBirth;
    validateState();
}

...

public void setName(String name) {
    this.name = name;
    validateState();
}

...

private void validateName() {
    // validações aqui...
}

private void validateState() {
    validateName();
    // validações aqui...
}
```



Métodos que verificam e garantem a validade do estado de um objeto (valores dos atributos) são utilizados também para implementar **invariantes da classe**. Invariantes são condições que devem ser sempre satisfeitas (a qualquer momento) para todos os objetos de uma classe. A única exceção é quando um objeto está em transição de um estado para outro (por exemplo, múltiplos atributos precisam ser alterados antes de validar o estado). Veremos mais sobre invariantes futuramente na disciplina.

E a classe **Employee**:

```
public Employee(int id, String name, LocalDate dateOfBirth, LocalDate hireDate,
    int hoursPerWorkWeek, BigDecimal hourlyRate) {
    super(name, dateOfBirth);
    this.id = id;
    this.hireDate = hireDate;
    this.hoursPerWorkWeek = hoursPerWorkWeek;
    this.hourlyRate = hourlyRate;
    validateState();
}

...

public void setHoursPerWorkWeek(int hoursPerWorkWeek) {
    this.hoursPerWorkWeek = hoursPerWorkWeek;
    validateHoursPerWorkWeek();
}

...

private void validateHoursPerWorkWeek() {
    if (hoursPerWorkWeek < 1 || hoursPerWorkWeek > 40) {
        throw new IllegalArgumentException("Hours per Work Week (" + hoursPerWorkWeek
            + ") is out of range [1..40]");
    }
}

private void validateState() {
    // outras validações aqui...
    validateHoursPerWorkWeek();
    // outras validações aqui...
}
```

A decisão de invocar a validação apenas deste atributo ou do estado como um todo dependerá da lógica de negócio. Fique atento a isso.

Implemente as alterações e teste o programa. Quando você decidir capturar a exceção, não esqueça de explicitar a exceção específica: `try-catch(IllegalArgumentException)`. O objetivo é definir métodos de validação para cada atributo e invocá-los sempre que os atributos forem modificados, preferencialmente, sem duplicação de código. No exemplo acima, estamos garantindo que a quantidade de horas será sempre um valor inteiro positivo, maior do que 0 e menor ou igual a 40. As exceções mais comuns utilizadas com validações são: `IllegalArgumentException` e `IllegalStateException`. Outra boa prática é sempre avaliar a validade dos argumentos passados para um método, mesmo que ele não modifique o estado do objeto. Nestes casos, além das exceções anteriores, `NullPointerException` também é tipicamente empregada.

Note também que o valor recebido como argumento foi apresentado junto com a mensagem. Isso facilita a análise do erro. Lembre-se disso quando estiver implementando suas validações.

## Roteiro 04 – Atividade 03/06

Para validações em um baixo nível de abstração, baseadas nas informações trocadas entre objetos (condições a serem satisfeitas, validade de argumentos, etc.), utilizaremos sempre exceções não checadas (subclasses de `RuntimeException`).

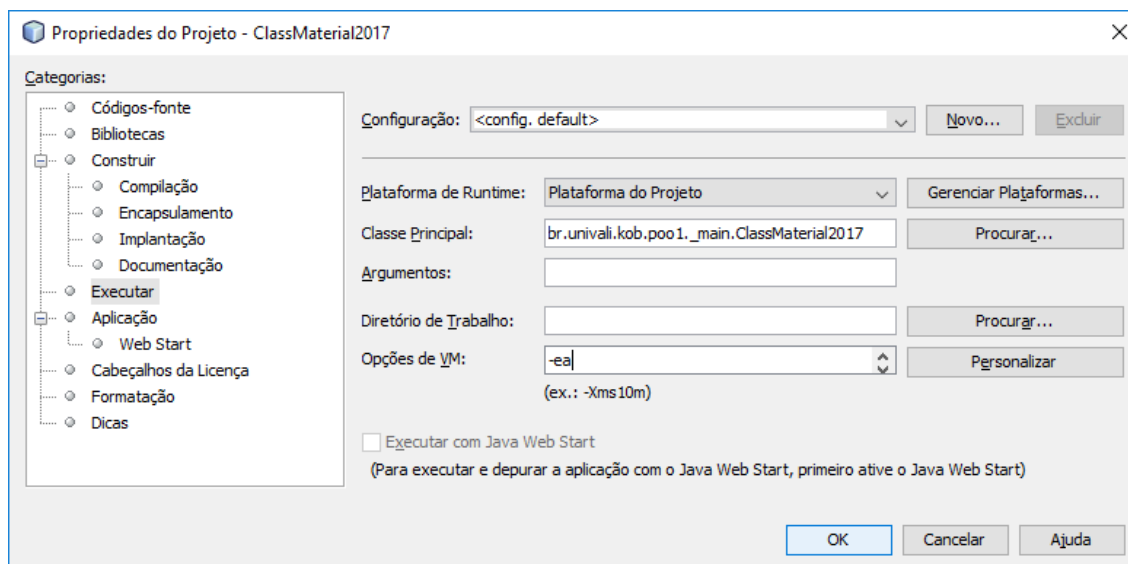


Embora exista discussões sobre vantagens e desvantagens sobre declarar **@throws** no comentário **javadoc** do método, nós iremos adotar esta declaração como boa prática. Esta prática será aplicada tanto para exceções checadas quanto para não checadas. O objetivo aqui é didático, pois forçará a confirmação das exceções utilizadas. Para evitar documentação excessiva, se todos os argumentos com valor `null` passados para os métodos de uma classe disparem uma exceção `NullPointerException`, você não precisa documentar a exceção a cada método. Neste caso, basta informar no javadoc da classe uma única vez. Eventualmente, em sua vida profissional, você pode avaliar quais orientações se aplicam melhor. Também observe quais são as regras definidas pela sua organização. Se elas não forem bem definidas, é recomendado que a equipe defina um guia de estilo de código que deverá ser adotado por todos.

Métodos privados não precisam ser validados. Por serem privados, o programador responsável pela classe deve garantir que valores corretos estão sendo passados e que o método não quebrará um invariante da classe. O desenvolvimento e aplicação de testes (chamados de unidade, por tratarem a classe como uma unidade) para garantir que o funcionamento interno está atendendo à especificação é de responsabilidade do programador. Note que, em nosso exemplo, a validação do argumento é feita em um método público (`setHoursPerWorkWeek`).

Uma forma de testar condições em um método privado é por meio do uso de assertivas (*asserts*). Uma assertiva permite que você teste suposições sobre seu programa. Embora este recurso seja interessante para testar condições como invariantes, ele é desabilitado em tempo de execução por default. **Deste modo, assertivas devem ser utilizadas SOMENTE para validações em métodos privados ou para depuração.**

Antes de apresentar um exemplo, você precisa configurar o modo que o Netbeans executa programas em Java. Com o botão direito do mouse sobre seu projeto, selecione a opção [ Propriedades | Executar ] e no campos “Opções de VM”, digite **-ea** (na linha de comando, seria “java -ea minhaAplicacao”).



A cláusula `assert` possui dois formatos de uso:

```
assert <expressão1>;
```

```
assert <expressão1> : <expressão2>;
```

A `expressão1` é sempre uma expressão booleana. No primeiro caso, se a `expressão1` for falsa, a exceção não checada `AssertionError` é disparada sem fornecer uma mensagem específica. No segundo caso, o sistema passa o valor da `expressão2` para o construtor da exceção `AssertionError`, o qual utiliza a representação *string* do



## Roteiro 04 – Atividade 03/06

valor passado como a mensagem de detalhamento do erro. Procure utilizar mensagens que tenham um significado consistente no contexto do método chamado e que ajudem durante a depuração do programa.

Embora `expressão1` possa ser o resultado da chamada de um método, esta não é uma boa prática. Assertivas não devem fazer parte da lógica do seu método. A ideia é que, ao desabilitá-las, o programa funcionará normalmente.

Experimente criar um método de teste e inserir o seguinte código:

```
int value = 41;  
assert value > 0 && value <= 40 : "Valor deve estar no intervalo [1..40].";
```

Execute o programa e note que a saída será similar a:

```
Exception in thread "main" java.lang.AssertionError: Valor deve estar no intervalo [1..40].  
    at br.univali.kob.pool.p04a_exception.InvariantTest.executeBundle(InvariantTest.java:56)  
    at br.univali.kob.pool.p01_simple_class.TestBundle.run(TestBundle.java:20)  
    at br.univali.kob.pool._main.ClassMaterial2017.main(ClassMaterial2017.java:56)  
C:\Users\marce\AppData\Local\NetBeans\Cache\8.1\executor-snippets\run.xml:53: Java returned: 1  
BUILD FAILED (total time: 0 seconds)
```

Outra possibilidade seria passar apenas o valor inválido para a exceção (experimente):

```
assert value > 0 && value <= 40 : value;
```

Você pode aprofundar um pouco mais seus conhecimentos sobre assertivas na documentação da Oracle, disponível em <https://docs.oracle.com/javase/8/docs/technotes/guides/language/assert.html>.

## Roteiro 04 – Atividade 04/06

1. Uma vez que exceções são classes em Java, então podemos implementar nossas próprias classes de exceção utilizando herança. Desta forma, podemos criar exceções específicas para a nossa aplicação. **Entretanto, antes de criar uma nova exceção, verifique cuidadosamente se a linguagem já não oferece uma exceção adequada para sua necessidade** (na maior parte dos casos, você utilizará uma exceção já existente). Entretanto, há situações em que criar uma exceção que possa aumentar a legibilidade do código, trazendo maior significado ao erro ocorrido. Considere o exemplo abaixo no contexto da nossa validação feita anteriormente.
2. Implemente a exceção `OutOfRangeException` abaixo e ajuste a validação implementada anteriormente (atualize o `javadoc` também).

```
public class OutOfRangeException extends IllegalArgumentException {

    /** Valor avaliado que está fora da faixa esperada ...3 lines */
    private final int value;
    /** Texto que descreve o valor ...3 lines */
    private final String valueLabel;
    /** Valor mínimo definido pela faixa ...3 lines */
    private final int min;
    /** Valor máximo definido pela faixa ...3 lines */
    private final int max;

    /**
     * A mensagem é montada com o rótulo passado no parâmetro.
     *
     * @param value o valor avaliado que está fora da faixa esperada
     * @param valueLabel o texto que descreve o valor
     * @param min o valor mínimo aceitável para a faixa
     * @param max o valor máximo aceitável para a faixa
     */
    public OutOfRangeException(int value, String valueLabel, int min, int max) {
        super("Value " + value + " for " + valueLabel + " is out of range [" +
            min + ".." + max + "]");
        this.value = value;
        this.valueLabel = valueLabel;
        this.min = min;
        this.max = max;
    }

    /**
     * @return o valor avaliado que está fora da faixa esperada
     */
    public int getValue() {
        return value;
    }

    /**
     * @return o valor máximo aceitável para a faixa
     */
    public String getValueLabel() {
        return valueLabel;
    }

    /**
     * @return o valor mínimo aceitável para a faixa
     */
    public int getMin() {
        return min;
    }

    /**
     * @return o valor máximo aceitável para a faixa
     */
    public int getMax() {
        return max;
    }
}
```

O nome de uma exceção deve sempre terminar com "Exception" (convenção plenamente adotada).

Os atributos devem ser sempre privados e modificados com final. Um objeto de exceção deve ser imutável.

Evite exceções que utilizam apenas uma mensagem String. Avalie quais informações ajudarão o programador a entender melhor o que aconteceu. Uma exceção própria deve agregar valor além de apenas ter um nome diferente.

## Roteiro 04 – Atividade 05/06

### Boas práticas para manipular exceções

Existe muita discussão e controvérsia sobre como tratar exceções em Java. Nesta seção, são apresentadas algumas das orientações mais comuns e que iremos adotar para a disciplina. Sempre leia estas dicas antes de utilizar exceções.

- Sempre que você capturar uma exceção, faça alguma ação. Evite apenas escondê-la e não fazer nada, pois embora o programa não seja abortado, a exceção terá ocorrido e o programa poderá não ter funcionado como esperado.
- Evite capturar uma exceção somente para propagá-la. Novamente, se você não tem nada para fazer em relação à exceção, por que você a capturou? Lembre-se que a propagação já é realizada automaticamente pelo sistema de execução JVM.
- Se você não quer capturar uma exceção, mas quer garantir que algo seja executado sempre (ex: algum tipo de faxina antes de terminar o método), utilize `try-finally`, sem o `catch`.
- Evite disparar `Exception`: `... throws Exception`. Esta exceção é genérica demais e apenas informa que algo deu errado (o que não ajuda muito). Declare as exceções checadas específicas que seu método irá disparar. Se existem várias que podem ser disparadas, uma alternativa é criar uma exceção para encapsulá-las (evitando a perda de legibilidade para uma lista longa de exceções). A mesma recomendação vale para a classe `Throwable` (ainda mais genérica).
- Evite capturar `Exception`: `... catch(Exception ex)`. Você deve capturar a exceção checada específica que deve ser tratada, evitando um tratamento buraco negro (pega tudo que passar). Se alguma exceção checada for adicionada futuramente no método que você estiver chamando, você nunca saberá. A mesma recomendação vale para a classe `Throwable` (ainda mais genérica).
- Evite disparar exceções dentro de um `finally`. Garanta que nunca ocorrerá uma exceção dentro de um `finally` ou, caso seja necessário, que ela seja sempre tratada de forma apropriada (ela não deveria ser propagada). O objetivo do código contido em um `finally` é fazer uma faxina geral (ex: liberar recursos alocados no `try`).
- Você deve registrar a ocorrência (log de ações realizadas ou ocorridas no sistema) de uma exceção ou propagá-la, mas nunca ambos. Se você fizer as duas ações simultaneamente, haverá duplicação nas mensagens de log para um mesmo erro, o que dificultará a depuração efetiva.
- Adote o princípio **“dispare mais cedo, capture mais tarde”**. Você deve disparar uma exceção o mais cedo que você puder e capturá-la o mais tarde possível no programa. Aguarde até que você tenha toda a informação necessária para tratar a exceção de modo apropriado. Na prática, você dispara uma exceção em métodos com baixo nível de abstração, onde é tipicamente verificada a validade de argumentos. A exceção então subirá o **stack trace** até alcançar um nível suficiente de abstração para tratá-la.
- Dispare exceções (`throw`) apropriadas para a abstração. Exceções disparadas por um método devem ser definidas em um nível de abstração consistente com aquilo que o método faz e não com detalhes de como ele é implementado (baixo nível). Por exemplo, um método que carrega recursos de arquivos ou de banco de dados deveria disparar algo como `ResourceNotFound` quando um recurso não é encontrado e não exceções de baixo nível como `IOException` ou `SQLException`.
- Sempre dispare exceções relevantes de um método. Por exemplo, se você está lendo um arquivo, disparar uma exceção `NullPointerException` não ajudará na interpretação do usuário.
- Nunca utilize exceções para controlar fluxo do programa. Exceções são para condições excepcionais e não devem ser utilizadas como parte da lógica de negócio. Por exemplo, se você está percorrendo um vetor, utilize os métodos disponíveis para verificar se ainda há mais algum elemento a ser lido. Não utilize a exceção `ArrayIndexOutOfBoundsException` como controle de fim do seu laço.

## Roteiro 04 – Atividade 05/06

- Documente todas as exceções utilizando o javadoc. Ao especificar a **annotation** `@throws nome_da_exceção`, inicie a explicação com um “se” e complemente com a situação que levará ao disparo da exceção.
- Utilize exceções checadas para condições em que o programa pode se recuperar e exceções não checadas (`RuntimeException`) para verificar defeitos de programação (por exemplo, violações de invariantes). Evite o uso desnecessário de exceções checadas (por exemplo, se a única possível resposta é terminar o programa).
- Um programa deveria raramente se recuperar de uma exceção. Programas devem ser robustos (não quebrar facilmente) em relação às entradas que ele recebe, mas não deveriam esconder suas falhas internas. Caso contrário, os programadores terão dificuldade em identificar e corrigir os defeitos. Utilize exceções checadas quando for esperado que método chamador capture e trate a mensagem (o programa deve poder se recuperar do erro ocorrido). Por exemplo, um método que irá gravar informações em um banco de dados. Neste caso, quem chamar este método deseja saber se a gravação foi bem-sucedida ou se algum erro aconteceu, pois será necessário apresentar alguma resposta para o usuário. Note que erros podem ocorrer, não por defeito de programação, mas por falha de comunicação com o servidor de banco de dados. Além disso, o programa pode tentar novamente (recuperação). Outra situação similar é quando o usuário solicita retirar um determinado valor de uma conta. A falta de dinheiro na conta não é um defeito de programação, mas uma tentativa do usuário de violar uma restrição (mesmo que ele não tenha tido a intenção).