Tratamento de Exceções

Programação Orientada a Objetos — QXD0007

UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS QUIXADÁ

Prof. Atílio Gomes Luiz

gomes.atilio@ufc.br

Universidade Federal do Ceará

1º semestre/2022

Leitura para esta aula

- **Capítulo 11** (Tratamento de Exceção) do livro Java Como Programar, Décima Edição.

Introdução

Exceção (*exception*)

Na programação orientada a objetos, uma exceção é um evento que ocorre durante a execução de um programa e quebra o fluxo normal de execução das instruções.

É uma condição anormal, possivelmente um erro, que deve ser tratada pelo programa.

Exceção (*exception*)

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Na programação orientada a objetos, uma exceção é um evento que ocorre durante a execução de um programa e quebra o fluxo normal de execução das instruções.

É uma condição anormal, possivelmente um erro, que deve ser tratada pelo programa.

Tratar uma exceção significa identificar quando uma exceção pode ocorrer e possibilitar uma correção do problema ou, então, finalizar a execução do programa de forma elegante e segura.

Exceção (*exception*)

- Linguagens orientadas a objetos, como C++ e Java, fornecem suporte para o tratamento sistemático de exceções.

Exceção (*exception*)

- Linguagens orientadas a objetos, como C++ e Java, fornecem suporte para o tratamento sistemático de exceções.

- O tratamento de exceções permite separar o código relativo ao tratamento de exceções do resto do código da aplicação.
  - Misturar a lógica do programa com a lógica do tratamento de erros pode tornar os programas difíceis de ler, modificar, manter e depurar.

Exceção (*exception*)

- Linguagens orientadas a objetos, como C++ e Java, fornecem suporte para o tratamento sistemático de exceções.

- O tratamento de exceções permite separar o código relativo ao tratamento de exceções do resto do código da aplicação.
  - Misturar a lógica do programa com a lógica do tratamento de erros pode tornar os programas difíceis de ler, modificar, manter e depurar.

- O tratamento de exceções permite que os programadores criem programas mais **robustos** e **tolerantes a falhas**.

Exemplos de situações anormais

- **Erros de lógica de programação:**
  - Devem ser corrigidos pelo programador;
  - Exemplos: (a) limite do vetor ultrapassado; (b) divisão por zero;

Exemplos de situações anormais

- **Erros de lógica de programação:**
  - Devem ser corrigidos pelo programador;
  - Exemplos: (a) limite do vetor ultrapassado; (b) divisão por zero;

- **Erros devido a condições do ambiente de execução:**
  - Fogem ao controle do programador, mas podem ser contornados;
  - Exemplos: (a) arquivo não encontrado; (b) conexão não estabelecida;

Exemplos de situações anormais

- **Erros de lógica de programação:**
  - Devem ser corrigidos pelo programador;
  - Exemplos: (a) limite do vetor ultrapassado; (b) divisão por zero;

- **Erros devido a condições do ambiente de execução:**
  - Fogem ao controle do programador, mas podem ser contornados;
  - Exemplos: (a) arquivo não encontrado; (b) conexão não estabelecida;

- **Erros graves, onde não há recuperação:**
  - Fogem ao controle do programador e não podem ser contornados;
  - Exemplos: (a) falta de memória; (b) erro interno da JVM.

Exemplos de Exceções Comuns

- `NullPointerException` — ocorre quando uma referência `null` é utilizada onde um objeto é esperado.

```java
public class NullPointer {
    static void doSomething(Integer number) {
        if(number > 0) {
            System.out.println(x: "positive number");
        }
    }
    Run | Debug
    public static void main(String[] args) {
        doSomething(number: null);
    }
}
```

Exemplos de Exceções Comuns

- `IndexOutOfBoundsException` — ocorre quando é feita uma tentativa de acessar um elemento fora dos intervalos de índice válidos de uma lista.

```java
import java.util.ArrayList;
import java.util.List;

public class IndexOutOfBounds {
    Run | Debug
    public static void main(String[] args) {
        List<String> nomes = new ArrayList<>();
        nomes.add(e: "Mara");
        nomes.add(e: "Junior");
        nomes.add(e: "Zeca");

        System.out.println(nomes.get(index: 5));
    }
}
```

Exemplos de Exceções Comuns

- `ClassCastException` — ocorre quando você tenta converter um objeto em outro objeto que não é da mesma hierarquia de herança.

```java
public class ClassCast {
    Run | Debug
    public static void main(String[] args) {
        Object valor = 1987;
        String nome = (String) valor;
        System.out.println(nome);
    }
}
```

Exemplos de Exceções Comuns

- `ArithmeticException` — é lançada quando ocorre um erro aritmético, como por exemplo, divisão inteira por zero.

```
return numerator / 0;
```

Exemplo


```
1  import java.util.Scanner;
2
3  public class DivideByZeroNoExceptionHandling {
4     // demonstrates throwing an exception
5     // when a divide-by-zero occurs
6     public static int quotient(int num, int den) {
7        return num / den; // possible division by zero
8     }
9
10    public static void main(String[] args) {
11       Scanner scanner = new Scanner(System.in);
12
13       System.out.print("Enter an integer numerator: ");
14       int numerator = scanner.nextInt();
15       System.out.print("Enter an integer denominator: ");
16       int denominator = scanner.nextInt();
17
18       int result = quotient(numerator, denominator);
19       System.out.printf("%nResult: %d / %d = %d%n",
20          numerator, denominator, result);
21    }
```

Quais erros podem acontecer durante a execução?

Stack trace (rastreamento de pilha)

Tentativa de Divisão de inteiro por zero:

```
Enter an integer numerator: 100
Enter an integer denominator: 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at DivideByZeroNoExceptionHandling.quotient(DivideByZeroNoExceptionHandling.java:9)
        at DivideByZeroNoExceptionHandling.main(DivideByZeroNoExceptionHandling.java:20)
```

Stack trace (rastreamento de pilha)

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Tentativa de Divisão de inteiro por zero:

```
Enter an integer numerator: 100
Enter an integer denominator: 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at DivideByZeroNoExceptionHandling.quotient(DivideByZeroNoExceptionHandling.java:9)
        at DivideByZeroNoExceptionHandling.main(DivideByZeroNoExceptionHandling.java:20)
```

- A figura acima mostra uma execução do programa anterior quando tentamos dividir 100 por 0. No Java, a divisão de um inteiro por zero não é permitida. A figura mostra o stack trace que é gerado quando a exceção é lançada.

- Um stack trace mostra uma lista de chamadas de método que levam ao lançamento da exceção, junto com os nomes de arquivo e números de linha em que as chamadas ocorreram.

Stack trace (rastreamento de pilha)

Entrada não casa com o tipo de dado esperado:

```
Enter an integer numerator: 100
Enter an integer denominator: oi
Exception in thread "main" java.util.InputMismatchException
        at java.base/java.util.Scanner.throwFor(Scanner.java:939)
        at java.base/java.util.Scanner.next(Scanner.java:1594)
        at java.base/java.util.Scanner.nextInt(Scanner.java:2258)
        at java.base/java.util.Scanner.nextInt(Scanner.java:2212)
        at DivideByZeroNoExceptionHandling.main(DivideByZeroNoExceptionHandling.java:18)
```

- Neste exemplo, foi lançado um objeto da classe `InputMismatchException`, que pertence ao pacote `java.util`
- A exceção `InputMismatchException` é lançada quando o método `Scanner.nextInt()` recebe uma String que não representa um inteiro válido.

Tratando exceções

Tratamento de Exceção - Geração e Captura

As duas atividades associadas ao tratamento de uma exceção são:

- **Geração:** a sinalização de que uma situação excepcional ocorreu.
- **Captura:** o tratamento da situação excepcional, onde as ações necessárias para a recuperação da situação de erro são definidas.

Blocos try-catch-finally

A captura e o tratamento de exceções em Java se dá através da especificação de blocos `try`, `catch` e `finally`, definidos através destas mesmas palavras reservadas da linguagem.

Blocos try-catch-finally

Um comando try/catch/finally obedece à seguinte sintaxe:

```
1  try {
2
3  }
4  catch (XException exception) {
5
6  }
7  catch (YException exception) {
8
9  }
10 finally {
11
12 }
```

Bloco try

- Bloco `try` contém o código que pode lançar (throw) uma exceção.
- Consiste na palavra-chave `try` seguida por um bloco de código entre chaves.
- Se ocorrer uma exceção em algum ponto, o restante do código contido no bloco `try` não será executado.

Bloco catch

- Um bloco `catch` captura (recebe) e trata uma exceção.
- Esse bloco recebe um parâmetro de exceção, que identifica o tipo de exceção e permite que o bloco catch interaja com o objeto da exceção capturada.
- Se o tipo do parâmetro de exceção de um bloco catch corresponder exatamente ao tipo de exceção lançada ou for uma superclasse dele, então esse bloco catch executará. A isso chamamos captura da exceção.

Bloco catch

- Um bloco `catch` captura (recebe) e trata uma exceção.
- Esse bloco recebe um parâmetro de exceção, que identifica o tipo de exceção e permite que o bloco catch interaja com o objeto da exceção capturada.
- Se o tipo do parâmetro de exceção de um bloco catch corresponder exatamente ao tipo de exceção lançada ou for uma superclasse dele, então esse bloco catch executará. A isso chamamos captura da exceção.
- Exceção não-capturada — uma exceção que ocorre para a qual não há nenhum bloco catch correspondente.
  - Faz com que o programa termine se o programa tiver somente um thread; do contrário apenas o thread atual é terminado e pode haver efeitos adversos no restante do programa.

Bloco finally

- O bloco `finally` é sempre executado. Em geral, ele inclui comandos que liberam recursos que eventualmente possam ter sido alocados durante o processamento do bloco try e que podem ser liberados, independentemente de a execução ter encerrado com sucesso ou ter sido interrompida por uma condição de exceção.
  - A presença desse bloco é opcional.

Exemplo 1


```
1  public class TesteException1 {
2      public static void main(String[] args) {
3          int[] vet = {1, 2, 3, 4, 5};
4          try {
5              // Esse laço tentará acessar uma região fora do
6              // intervalo do vetor: uma exceção será lançada
7              for(int i = 0; i <= 9; i++)
8                  System.out.println( vet[i] );
9          }
10         catch(ArrayIndexOutOfBoundsException excp) {
11             System.out.println(excp); // toString implícito
12         }
13         catch(Exception e) {
14             System.out.println("Execução do catch Exception");
15             System.out.println(e); // toString implícito
16         }
17         finally {
18             System.out.println("finally sempre executado");
19         }
20     }
21 }
```

Neste exemplo, um objeto da classe `ArrayIndexOutOfBoundsException` é lançado. Esta classe pertence ao pacote `java.lang`

Exemplo 2

- No exemplo a seguir, adicionamos o tratamento de exceções ao programa da divisão inteira por zero.

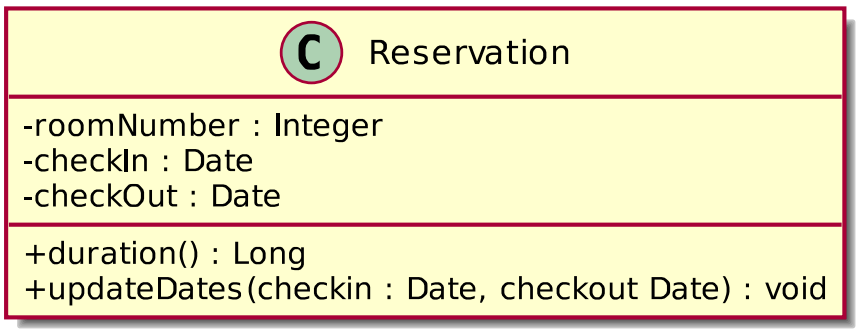
- Analisar o arquivo `DivideByZeroWithExceptionHandling.java`

Problema Exemplo

Fazer um programa para ler os dados de uma reserva de hotel (número do quarto, data de entrada e data de saída) e mostrar os dados da reserva, inclusive sua duração.

Em seguida, ler novas datas de entrada e saída, atualizar a reserva, e mostrar novamente a reserva com os dados atualizados. O programa não deve aceitar dados inválidos para a reserva, conforme as seguintes regras:

- alterações de reservas só podem ocorrer para datas futuras
- a data de saída deve ser maior que a data de entrada



Exemplos


```
Room number: 8021
Check-in date (dd/MM/yyyy): 23/09/2019
Check-out date (dd/MM/yyyy): 26/09/2019
Reservation: Room 8021, check-in: 23/09/2019, check-out: 26/09/2019, 3 nights

Enter data to update the reservation:
Check-in date (dd/MM/yyyy): 24/09/2019
Check-out date (dd/MM/yyyy): 29/09/2019
Reservation: Room 8021, check-in: 24/09/2019, check-out: 29/09/2019, 5 nights
```

```
Room number: 8021
Check-in date (dd/MM/yyyy): 23/09/2019
Check-out date (dd/MM/yyyy): 21/09/2019
Error in reservation: Check-out date must be after check-in date
```

Exemplos

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
Room number: 8021
Check-in date (dd/MM/yyyy): 23/09/2019
Check-out date (dd/MM/yyyy): 26/09/2019
Reservation: Room 8021, check-in: 23/09/2019, check-out: 26/09/2019, 3 nights

Enter data to update the reservation:
Check-in date (dd/MM/yyyy): 24/09/2015
Check-out date (dd/MM/yyyy): 29/09/2015
Error in reservation: Reservation dates for update must be future dates
```

```
Room number: 8021
Check-in date (dd/MM/yyyy): 23/09/2019
Check-out date (dd/MM/yyyy): 26/09/2019
Reservation: Room 8021, check-in: 23/09/2019, check-out: 26/09/2019, 3 nights

Enter data to update the reservation:
Check-in date (dd/MM/yyyy): 24/09/2020
Check-out date (dd/MM/yyyy): 22/09/2020
Error in reservation: Check-out date must be after check-in date
```

Date

- Representa um instante
- Pacote java.util
- `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Date.html`
- Um objeto Date internamente armazena:
  - O número de milissegundos desde a meia noite do dia 1 de janeiro de 1970 GMT
    - GMT: Greenwich Mean Time (time zone)

Simple Date Format

- Pacote java.text
- `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/text/SimpleDateFormat.html`
- Define formatos para conversão entre Date e String.
  - dd/MM/yyyy → 23/07/2018
  - dd/MM/yyyy HH:mm:ss → 23/07/2018 15:42:07

Exemplo – Date


```java
import java.text.SimpleDateFormat;
import java.util.Date;

public class DateTest {
    public static void main(String[] args) throws Exception {
        SimpleDateFormat sdf1 =
            new SimpleDateFormat("dd/MM/yyyy");
        SimpleDateFormat sdf2 =
            new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");

        Date now = new Date(); // agora

        System.out.println(now);
        System.out.println(sdf1.format(now));
        System.out.println(sdf2.format(now));

        Date date = sdf2.parse("05/06/2022 23:04:00");
        System.out.println(date);
    }
}
```

Blocos try-catch-finally (Regras)

- O bloco **try** deve ser seguido por pelo menos um bloco **catch**, ou por um bloco **finally**;
- Os blocos não podem ser separados por outros comandos – um erro de sintaxe seria detectado pelo compilador Java neste caso.
- Cada bloco `try` pode ser seguido por zero ou mais blocos `catch`.

Blocos try-catch-finally (Regras)

- O bloco **try** deve ser seguido por pelo menos um bloco **catch**, ou por um bloco **finally**;
- Os blocos não podem ser separados por outros comandos – um erro de sintaxe seria detectado pelo compilador Java neste caso.
- Cada bloco `try` pode ser seguido por zero ou mais blocos `catch`.
- Geralmente, cada bloco catch trata de apenas uma exceção. Porém, se os corpos de vários blocos `catch` forem iguais, você pode usar um multi-catch para capturar todas essas exceções em um único bloco catch e realizar a mesma tarefa para cada uma delas.
  - Sintaxe: `catch (Type1 | Type2 | Type3 exception)`

Blocos try-catch-finally (Regras)

- Caso um **catch** mais genérico (por exemplo, catch(Exception e)) apareça antes de um mais específico (por exemplo, catch(ArithmeticException e)), o bloco mais específico jamais será executado.
  - O compilador detecta a situação acima e sinaliza o erro.
  - Portanto, as exceções listadas em sucessivos blocos catch devem ser dispostas da classe mais específica para a classe mais geral.

Blocos try-catch-finally (Regras)

- Caso um **catch** mais genérico (por exemplo, catch(Exception e)) apareça antes de um mais específico (por exemplo, catch(ArithmeticException e)), o bloco mais específico jamais será executado.
  - O compilador detecta a situação acima e sinaliza o erro.
  - Portanto, as exceções listadas em sucessivos blocos catch devem ser dispostas da classe mais específica para a classe mais geral.
- Quando um bloco try termina, as variáveis locais declaradas no bloco saem do escopo e não são mais acessíveis fora do bloco try.

Utilizando a cláusula `throws`

Utilizando a cláusula throws

- Cláusula `throws` — especifica as exceções que um método pode lançar.
- Aparece depois da lista de parâmetros do método e antes do corpo do método.
- Contém uma lista de exceções separadas por vírgulas.
- As exceções podem ser lançadas pelas instruções no corpo do método ou pelos métodos chamados no corpo do método.
- As exceções podem ser dos tipos listados na cláusula throws ou subclasses delas.

Utilizando a cláusula throws

- Se souber que um método pode lançar uma exceção, inclua o código de tratamento de exceções apropriado no programa para torná-lo mais robusto.

- Leia a documentação on-line da API para obter informações sobre um método antes de utilizar esse método em um programa.
  - A documentação especifica as exceções lançadas pelo método (se houver alguma) e indica as razões pelas quais tais exceções podem ocorrer.

Modelo de terminação de tratamento de exceções

Modelo de terminação × Modelo de retomada

- Modelo de terminação do tratamento de exceções — o controle do programa não retorna ao ponto de lançamento porque o bloco catch terminou; o fluxo de controle prossegue para a primeira instrução depois do último bloco catch. **Usado pelo Java.**

- Modelo de retomada do tratamento de exceções — o controle do programa é retomado logo depois do ponto de lançamento.

Modelo de Terminação

1. O código contido em um bloco **try** lança uma exceção;
2. Um objeto contendo informações sobre a exceção é instanciado;
3. O fluxo de execução desce pelos blocos **catch** até encontrar um que declare tratar a exceção específica que foi lançada;
4. O bloco **catch** correspondente é executado;
5. O bloco **finally** é executado, caso exista um. Na verdade, esse bloco é sempre executado, independente de ter sido ou não lançada uma exceção;
6. O código que segue o mecanismo **try-catch-finally** é executado, caso não exista um **return** no bloco que tratou a exceção.

O bloco `finally`

- O bloco **finally** contém instruções que devem ser executadas independentemente da ocorrência ou não de exceções.

O bloco `finally`

- O bloco **finally** contém instruções que devem ser executadas independentemente da ocorrência ou não de exceções.

- O bloco **finally** executará se uma exceção for ou não lançada no bloco **try** correspondente. O bloco finally também será executado se o fluxo de execução normal do bloco **try** for interrompido usando uma instrução `return`, `break` ou `continue`.

O bloco `finally`

- O bloco **finally** contém instruções que devem ser executadas independentemente da ocorrência ou não de exceções.

- O bloco **finally** executará se uma exceção for ou não lançada no bloco **try** correspondente. O bloco finally também será executado se o fluxo de execução normal do bloco **try** for interrompido usando uma instrução `return`, `break` ou `continue`.

- O único caso em que o bloco **finally** não será executado é se o método **System.exit** for chamado dentro do bloco **try**. Esse método encerra o programa imediatamente.

O bloco `finally`

- O bloco **finally** contém instruções que devem ser executadas independentemente da ocorrência ou não de exceções.

- O bloco **finally** executará se uma exceção for ou não lançada no bloco **try** correspondente. O bloco finally também será executado se o fluxo de execução normal do bloco **try** for interrompido usando uma instrução `return`, `break` ou `continue`.

- O único caso em que o bloco **finally** não será executado é se o método **System.exit** for chamado dentro do bloco **try**. Esse método encerra o programa imediatamente.

- Resumindo: Um bloco **try** quase sempre executará.

O bloco `finally` — Quando usar?

- **Situação:** Programas que obtêm determinados recursos devem devolvê-los ao sistema para evitar *resource leaks*. Em linguagens de programação, como C e C++, o vazamento de recursos mais comum é um vazamento de memória.

- Java executa a coleta automática de lixo da memória que não é mais usada pelos programas, evitando assim a maioria dos vazamentos de memória.

- No entanto, outros tipos de vazamentos de recursos podem ocorrer. Por exemplo, arquivos, conexões de banco de dados e conexões de rede que não estão fechadas depois que eles não forem mais necessários, podem não estar disponíveis para uso em outros programas.

O bloco `finally` — Quando usar?

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- Como um bloco **finally** sempre é executado, ele geralmente contém código de liberação de recurso.

- Suponha que um recurso seja alocado em um bloco try. Se nenhuma exceção ocorrer, os blocos **catch** serão ignorados e o controle continuará no bloco **finally**, que libera o recurso.

- Se ocorrer uma exceção no bloco try, o bloco try será finalizado. Se o programa capturar a exceção em um dos blocos catch correspondentes, ele processará a exceção e, em seguida, o bloco finally liberará o recurso e o controle prossegue para a primeira declaração após o bloco finally.

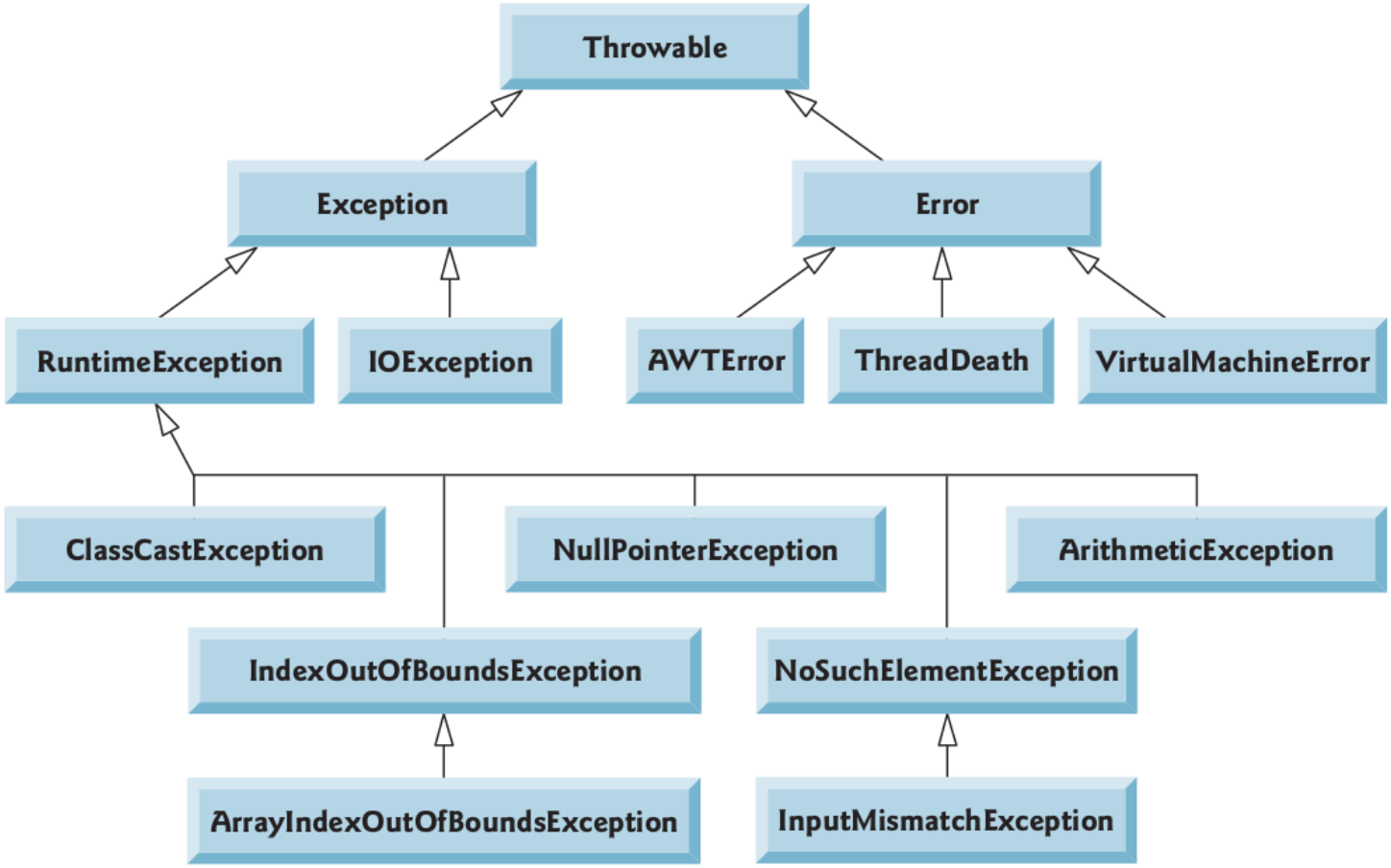
- Se o programa não capturar a exceção, o bloco finally ainda libera o recurso e uma tentativa é feita para capturar a exceção em um método de chamada.

Exemplo

- Mostrar projeto `ProjetoArquivo`.

Hierarquia de exceção no Java

Hierarquia de exceção no Java



Hierarquia de exceção no Java

- Todas as exceções que vimos até agora são herdadas direta ou indiretamente da classe `Exception`.

- `Exception` e suas descendentes formam uma hierarquia de herança que pode ser estendida.

Hierarquia de exceção no Java

- Todas as exceções que vimos até agora são herdadas direta ou indiretamente da classe `Exception`.

- `Exception` e suas descendentes formam uma hierarquia de herança que pode ser estendida.

- Classe `Throwable`, superclasse da `Exception`:
  - Somente objetos `Throwable` podem ser utilizados com o mecanismo de tratamento de exceções.

Hierarquia de exceção no Java

- Todas as exceções que vimos até agora são herdadas direta ou indiretamente da classe `Exception`.

- `Exception` e suas descendentes formam uma hierarquia de herança que pode ser estendida.

- Classe `Throwable`, superclasse da `Exception`:
  - Somente objetos `Throwable` podem ser utilizados com o mecanismo de tratamento de exceções.
  - Tem duas subclasses: `Exception` e `Error`.
    - A classe `Exception` e suas subclasses representam situações excepcionais que podem ocorrer em um programa Java e que podem ser capturadas pelo aplicativo.

Hierarquia de exceção no Java

- Todas as exceções que vimos até agora são herdadas direta ou indiretamente da classe `Exception`.

- `Exception` e suas descendentes formam uma hierarquia de herança que pode ser estendida.

- Classe `Throwable`, superclasse da `Exception`:
  - Somente objetos `Throwable` podem ser utilizados com o mecanismo de tratamento de exceções.
  - Tem duas subclasses: `Exception` e `Error`.
    - A classe `Exception` e suas subclasses representam situações excepcionais que podem ocorrer em um programa Java e que podem ser capturadas pelo aplicativo.
    - A classe `Error` e suas subclasses representam situações anormais que poderiam acontecer na JVM – normalmente não é possível que um programa se recupere de Errors.

java.lang.Throwable

- Classe `Throwable`: ancestral de todas as classes que recebem tratamento do mecanismo de exceções.

- Principais métodos:
  - `void printStackTrace()`: envia para a saída de erros padrão (System.err) a sequência de métodos chamados até o ponto onde a exceção foi lançada.
  - `String getMessage()`: retorna uma String contendo uma mensagem indicadora da exceção.
  - `String toString()`: retorna uma descrição sumária da exceção.
  - `getStackTrace`: recupera informações do rastreamento de pilha como um array de objetos `StackTraceElement`; permite processamento personalizado das informações sobre a exceção.

Exemplo

- Analisar o `ProjetoStackTrace`.

Exceções verificadas vs. Exceções não-verificadas

Exceções não-verificadas

Exceções não-verificadas (unchecked exceptions):

- Herdam da classe `RuntimeException` ou da classe `Error`.
- O compilador não verifica o código para ver se a exceção foi capturada ou declarada.
- Se uma exceção não-verificada ocorrer e não tiver sido capturada, o programa terminará ou executará com resultados inesperados.
- Em geral, podem ser evitadas com uma codificação adequada.

Exceções verificadas

Exceções verificadas (checked exceptions):

- Essas exceções que são herdadas da classe `Exception`, mas não de `RuntimeException`.
  - `ClassNotFoundException`, `IOException`, `SQLException` são exemplos de exceções verificadas.

Exceções verificadas

Exceções verificadas (checked exceptions):

- Essas exceções que são herdadas da classe `Exception`, mas não de `RuntimeException`.
  - `ClassNotFoundException`, `IOException`, `SQLException` são exemplos de exceções verificadas.

- Exceções verificadas pelo compilador em tempo de compilação. Essas exceções devem ser detectadas por um `try-catch` no código ou declaradas pelo método como lançáveis (cláusula `throws`).

Exceções verificadas

Exceções verificadas (checked exceptions):

- Essas exceções que são herdadas da classe `Exception`, mas não de `RuntimeException`.
  - `ClassNotFoundException`, `IOException`, `SQLException` são exemplos de exceções verificadas.

- Exceções verificadas pelo compilador em tempo de compilação. Essas exceções devem ser detectadas por um `try-catch` no código ou declaradas pelo método como lançáveis (cláusula `throws`).

- Por exemplo, se um programa tentar acessar um arquivo que não está disponível no momento, o método que tenta acessar o arquivo deve capturar ou declarar uma `FileNotFoundException`.

Exceções verificadas

Exceções verificadas (checked exceptions):

- Essas exceções que são herdadas da classe `Exception`, mas não de `RuntimeException`.
  - `ClassNotFoundException`, `IOException`, `SQLException` são exemplos de exceções verificadas.

- Exceções verificadas pelo compilador em tempo de compilação. Essas exceções devem ser detectadas por um `try-catch` no código ou declaradas pelo método como lançáveis (cláusula `throws`).

- Por exemplo, se um programa tentar acessar um arquivo que não está disponível no momento, o método que tenta acessar o arquivo deve capturar ou declarar uma `FileNotFoundException`.

Analisar `ProjetoArquivo2.java`

Declarando novas classes de Exceção

Declarando novas classes de Exceção

- Você pode declarar suas próprias classes de exceção específicas dos problemas que podem ocorrer quando um outro programa utiliza suas classes reutilizáveis.

- A nova classe de exceção deve estender uma classe de exceção existente.

Declarando novas classes de Exceção

- Você pode declarar suas próprias classes de exceção específicas dos problemas que podem ocorrer quando um outro programa utiliza suas classes reutilizáveis.

- A nova classe de exceção deve estender uma classe de exceção existente.

- Por convenção, todos os nomes de classe de exceções devem terminar com a palavra `Exception`.

Declarando novas classes de Exceção

- Em geral, uma nova classe de exceção deve conter como membros somente quatro construtores:
  - Um construtor sem argumentos que passa uma String padrão para o construtor da superclasse.
  - Um construtor que recebe uma String como argumento e a repassa para o construtor da superclasse.
  - Um construtor que recebe uma String e um objeto `Throwable` como argumentos e os repassa para o construtor da superclasse.
  - Um construtor que recebe um objeto `Throwable` como argumento e o repassa para o construtor da superclasse.

Observações

- Se possível, indique as exceções provenientes de seus métodos utilizando classes de exceção existentes, em vez de criar novas classes de exceção. A API do Java contém muitas classes de exceção que podem ser adequadas ao tipo de problema que seu método precisa indicar.

Observações

- Ao definir seu próprio tipo de exceção, estude as classes de exceção existentes na API do Java e tente estender uma classe de exceção relacionada.

- Se as classes existentes não forem superclasses apropriadas para sua nova classe de exceção, decida se a nova classe deve ser uma classe de exceção verificada ou não-verificada.

Observações

- Ao definir seu próprio tipo de exceção, estude as classes de exceção existentes na API do Java e tente estender uma classe de exceção relacionada.

- Se as classes existentes não forem superclasses apropriadas para sua nova classe de exceção, decida se a nova classe deve ser uma classe de exceção verificada ou não-verificada.

- A nova classe de exceção deve ser uma exceção verificada (isto é, estender `Exception`, mas não `RuntimeException`) se possíveis clientes precisarem tratar a exceção. A aplicação cliente deve ser razoavelmente capaz de se recuperar de tal exceção.

Observações

- Ao definir seu próprio tipo de exceção, estude as classes de exceção existentes na API do Java e tente estender uma classe de exceção relacionada.

- Se as classes existentes não forem superclasses apropriadas para sua nova classe de exceção, decida se a nova classe deve ser uma classe de exceção verificada ou não-verificada.

- A nova classe de exceção deve ser uma exceção verificada (isto é, estender `Exception`, mas não `RuntimeException`) se possíveis clientes precisarem tratar a exceção. A aplicação cliente deve ser razoavelmente capaz de se recuperar de tal exceção.

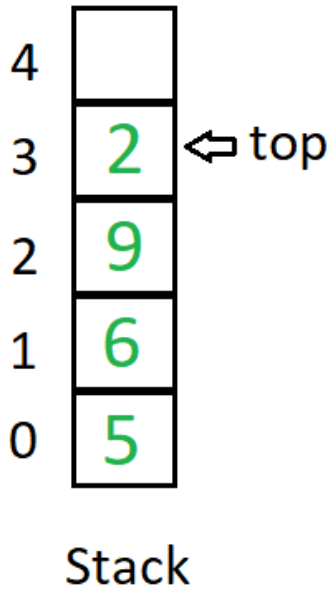
- A nova classe de exceção deve estender `RuntimeException` se o código de cliente for capaz de ignorar a exceção (isto é, se a exceção for uma exceção não-verificada).

Exemplo – Implementando uma Pilha

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

A estrutura de dados pilha pode ser implementada usando um array de tamanho fixo e possui duas operações básicas:

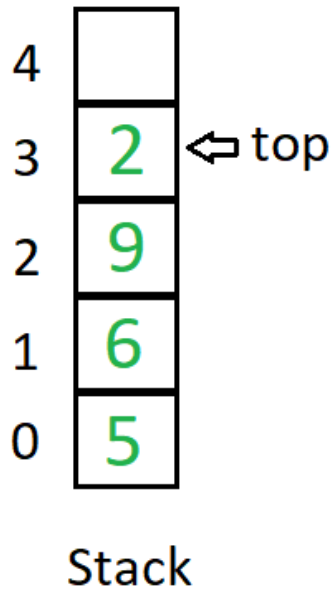
- `push(x)`: colocar o elemento `x` no topo da pilha
- `pop()` remover o elemento que está no topo da pilha e retornar o seu valor.



Exemplo – Implementando uma Pilha

A estrutura de dados pilha pode ser implementada usando um array de tamanho fixo e possui duas operações básicas:

- `push(x)`: colocar o elemento `x` no topo da pilha
- `pop()` remover o elemento que está no topo da pilha e retornar o seu valor.



- Ao tentar inserir mais elementos do que a pilha suporta, uma exceção deve ser lançada. Do mesmo modo, ao tentar remover elementos de uma pilha vazia, uma exceção deve ser lançada.
- Analisar o projeto `Stack`.

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

FIM