fusberti@ic.unicamp.br

Tratamento de Exceções

prof. Fábio Luiz Usberti

MC322 - Programação Orientada a Objetos

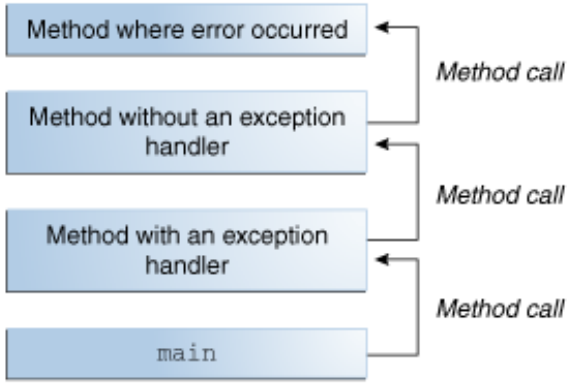
Instituto de Computação - UNICAMP - 2014

Sumário

Exceções

- Uma exceção ocorre durante a execução de um programa diante de um evento excepcional, interrompendo o fluxo normal de instruções do programa.
- Quando uma exceção ocorre dentro de um método, esse método lança um objeto de exceção, contendo informações sobre o problema ocorrido, como o tipo de exceção e o estado do programa quando o problema ocorreu.
- Quando um método lança uma exceção, a JVM tenta encontrar um tratador apropriado para essa exceção.
- Esse tratador pode se encontrar em qualquer um dos métodos que se encontram na pilha de execução.

Exceções

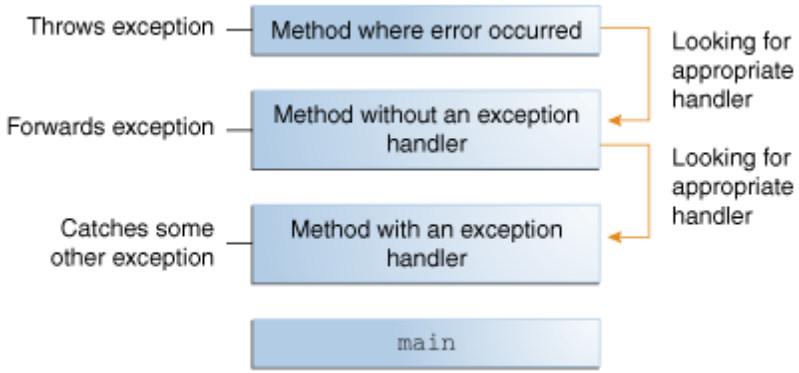


Fonte: http://docs.oracle.com/javase/tutorial/essential/exceptions/

Exceções

- A JVM busca na pilha de execução por um método que contenha um bloco `catch` capaz de tratar o tipo da exceção que foi lançada.
- A busca se inicia no próprio método onde a exceção ocorreu e retrocede na pilha de execução até encontrar um método capaz de tratar a exceção.
- Ao encontrar um bloco `catch` correspondente, a exceção é passada para esse tratador, processo denominado captura da exceção.
- Se a busca na pilha de execução por um tratador de exceção não tiver sucesso, a exceção é capturada por um tratador default que imprime o tipo da exceção, sua descrição e o estado da pilha de execução quando a exceção foi lançada.

Exceções



Fonte: http://docs.oracle.com/javase/tutorial/essential/exceptions/

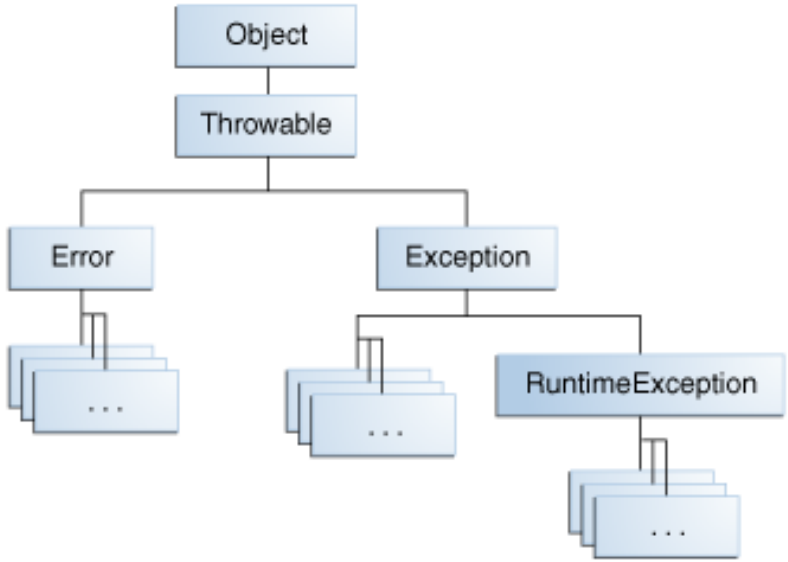
Hierarquia de Exceções

Classe `Exception`

- Exceções em Java herdam direta ou indiretamente da classe `Throwable`, formando uma hierarquia de exceções.
- Qualquer objeto `Throwable` pode ser utilizado pelo mecanismo de tratamento de exceções em Java.
- É possível estender a hierarquia com classes de exceções próprias.
- As duas subclasses diretas de Throwable são `Exception` e `Error`.

fusberti@ic.unicamp.br

Hierarquia de Exceções



Fonte: http://docs.oracle.com/javase/tutorial/essential/exceptions/

Hierarquia de Exceções

Classe `Exception`

- A classe `Exception` e suas subclasses representam situações anormais em um programa Java e que podem ser tratadas pela aplicação.
- A classe `Error` e suas subclasses representam situações anormais mais graves, que normalmente acontecem na JVM. A frequencia de exceções do tipo `Error`, ou simplesmente erros, em um aplicativo é baixa.
- Erros não devem ser capturados pela aplicação, dado que na maioria das vezes um programa não consegue se recuperar de um erro.
- Leia a documentação API sobre a classe `Throwable` para saber mais sobre os diversos tipos de exceções e erros.

Política de capturar ou declarar

Exceções verificadas e não verificadas

- A linguagem Java estabelece uma política de captura ou declaração para instruções que lançam certos tipos de exceção, denominadas exceções verificadas (pelo compilador).
- Instruções que lançam exceções verificadas devem respeitar uma das seguintes condições:
  1. Captura: A instrução deve estar contida em um bloco `try..catch` que captura o tipo correspondente da exceção verificada.
  2. Declaração: A instrução deve estar contida em um método que declara (com a palavra-chave `throws`) o tipo correspondente da exceção verificada.
- Um código que não respeita a política de captura ou declaração para exceções verificadas resulta em um erro de compilação.

Política de capturar ou declarar

Exceções verificadas e não verificadas

- A linguagem Java distingue duas categorias de exceções: verificadas (*checked*) e não verificadas (*unchecked*).
- Determinar se uma exceção é verificada ou não depende exclusivamente de sua classe:
  - Exceções do tipo `RuntimeException (java.lang)` e suas subclasses são exceções não verificadas.
  - Exceções do tipo `Error`, ou simplesmente erros, são consideradas exceções não verificadas.
  - Quaisquer outras classes de exceções correspondem a exceções verificadas.

Política de capturar ou declarar

```
// CatchOrDeclare.java
// A classe CatchOrDeclare exemplifica a política de captura ou declaração de exceções verificadas.
public class CatchOrDeclare {

    // method1 realiza a captura da exceção verificada Exception
    public static void method1() {
        try {
            method2();
        } catch (Exception e) {
            System.err.println("Exceção capturada: "+e);
        }
    } // fim method1

    // method2 realiza a declaração da exceção verificada Exception
    public static void method2() throws Exception {
        method3();
    } // fim method2

    // method3 realiza a declaração da exceção verificada Exception
    public static void method3() throws Exception {
        throw new Exception();
    } // fim method3

    public static void main(String args[]) {
        method1();
    } // fim main

} // fim classe CatchOrDeclare
```

Política de capturar ou declarar

Exceções verificadas e não verificadas

- As exceções não verificadas têm esse nome por não serem checadas pelo compilador, portanto elas não participam da política de capturar ou declarar. Em outras palavras, esses tipos de exceção não requerem estar declaradas na assinatura do método e também não precisam ser capturadas pela aplicação.
- Exceções do tipo `RuntimeException` são tipicamente lançadas devido a defeitos do próprio código. Em outras palavras, essas exceções resultam de erros lógicos de programação e usos indevidos de APIs.

Política de capturar ou declarar

Exceções verificadas e não verificadas

- Exceções não verificadas, por se tratarem de defeitos do código, devem ser evitadas pela correção dos defeitos presentes no código, portanto em geral elas não precisam ser tratadas durante execução. Ainda assim, o tratamento desses tipos de exceção pode ser conveniente em alguns casos:
- Exemplo: O método `parseInt` da classe `Integer` lança uma exceção não verificada `NumberFormatException` caso um número não inteiro seja passado como argumento. Desse modo, apesar do compilador não exigir a captura ou declaração de exceções não verificadas, o programa torna-se mais robusto se mesmo esses tipos de exceções forem tratadas.
- Exemplos de exceções não verificadas incluem `NullPointerException`, `ArrayIndexOutOfBoundsExceptions` e `ArithmeticExceptions`.

Política de capturar ou declarar

Exceções verificadas e não verificadas

- Todas as exceções que não são do tipo `RuntimeException` ou `Error` são exceções verificadas.
- Exceções verificadas são tipicamente aquelas provocadas por situações fora do controle do programa, mas que podem ser antecipadas e para as quais o aplicativo ainda pode se recuperar.
- Exemplo: um aplicativo solicita ao usuário o nome de um arquivo para leitura e escrita. Eventualmente, é possível que o usuário passe o nome de um arquivo não existente, o que poderá provocar uma exceção do tipo `FileNotFoundException`. Um bom programa deve antecipar essa exceção, notificando o usuário do engano e permitindo uma nova entrada de nome de arquivo.

Política de capturar ou declarar

Exceções verificadas e não verificadas

- Se um método chama um segundo método que declara exceções verificadas, essa chamada deve respeitar a política de captura ou declaração.
- Se uma exceção já pode ser tratada adequadamente em um método, esse método deve capturar a exceção ao invés de declará-la.
- Na sobreposição de um método, o método da subclasse pode declarar qualquer subconjunto de exceções declaradas pelo método da superclasse.

Política de capturar ou declarar

Capturando exceções em subclasses

- Um tratador de exceção que captura um objeto de exceção de uma superclasse também captura objetos de exceções das subclasses correspondentes.
- Isso permite realizar processamento polimórfico de exceções.
- Também é possível tratar cada subclasse de exceção de forma individual caso as exceções requisitarem um processamento específico.

Bloco `finally`

Vazamento de recursos

- Programas que utilizam certos tipos de recursos devem retorná-los ao sistema explicitamente para evitar o chamado vazamento de recursos.
- Em linguagens como C é comum ocorrer vazamentos de memória, o que é menos comum em Java graças ao coletor de lixo.
- Vazamentos de memória ainda podem ocorrer mesmo em aplicativos Java pois o coletor de lixo não libera a memória de objetos que ainda possuem referências associadas.
- Se o programador erroneamente mantiver referências de objetos não utilizados, haverá um vazamento de memória. Para evitar essas situações, o programador deve lembrar de atribuir `null` a variáveis que não serão mais utilizadas.

Bloco `finally`

Vazamento de recursos

- Outros possíveis tipos de vazamentos ocorrem com referências a arquivos, banco de dados e conexões de redes que não são apropriadamente fechadas.
- Recursos não fechados pelo aplicativo podem ficar indisponíveis para uso por outras aplicações.
- É possível evitar muitos tipos de vazamentos de recursos através do bloco `finally`, que se trata de um bloco opcional que pode ser anexado a um bloco `try..catch`.
- O bloco `finally` deve ser colocado após o último bloco `catch` ou após o bloco `try` se não houver nenhum `catch`.

Bloco `finally`

Estrutura de um bloco `try..catch..finally`

```
public class ClassName {
    public void methodName() {
        try {
            // instruções
            // instruções para aquisição de recursos
        } catch (TypeOfException e) {
            // instruções para o tratamento da exceção
        } catch (AnotherTypeOfException e) {
            // instruções para o tratamento da exceção
        } finally {
            // instruções
            // instruções de liberação de recursos
        }
    }
}
```

Bloco `finally`

Vazamento de recursos

- O bloco `finally` é (quase) sempre executado, mesmo nas seguintes situações ocorridas no bloco `try` correspondente:
  - O bloco `try` foi executado até o fim sem lançar nenhuma exceção.
  - O bloco `try` foi interrompido por uma exceção lançada.
  - O bloco `try` foi interrompido por um `return`.
  - O bloco `try` foi interrompido por um `break`.
  - O bloco `try` foi interrompido por um `continue`.
- O bloco `finally` não é executado se ocorrer uma chamada do método `System.exit` no bloco `try` correspondente, pois isso resulta no término da execução do aplicativo.

Bloco `finally`

Vazamento de recursos

- Se nenhuma exceção ocorrer em um bloco `try`, a linha de execução pula os blocos `catch` e passa diretamente para as instruções do bloco `finally`.
- Se uma exceção ocorre no bloco `try` e a exceção é capturada por um dos blocos `catch`, então a exceção é tratada e a linha de execução passa para as instruções do bloco `finally`.
- Se uma exceção ocorre no bloco `try` e a exceção não é capturada por nenhum dos blocos `catch`, então a linha de execução passa para as instruções do bloco `finally` e em seguida a exceção é relançada para o método que fez a chamada.

Bloco `finally`

Vazamento de recursos

- Considerando que o bloco `finally` é executado na ocorrência ou não de uma exceção, ele normalmente contém códigos para a liberação de recursos que foram alocados em um bloco `try`.
- Se uma exceção que ocorre em um bloco `try` não pode ser capturada por nenhum de seus tratadores, então o programa pula todo o restante do bloco `try` e passa diretamente para o bloco `finally`. Em seguida, o programa passa a exceção para o próximo bloco `try` externo, que pode se encontrar no método seguinte da pilha de execução. Esse processo pode ocorrer através de múltiplos níveis de blocos `try`.

Palavra-chave `throw`

Lançando exceções

- Métodos podem lançar uma exceção através da instrução `throw`.
- Uma instrução `throw` requer um único argumento, um objeto da classe (ou subclasse de) `Throwable`.
- Sintaxe da instrução `throw`:

```
throw someThrowableObject;
```

- No exemplo a seguir é fornecida uma classe que define uma pilha simples de objetos genéricos. Essa classe contém os métodos `pop` e `push` que lançam exceções não verificadas `EmptyStackException` e `FullStackException`.

Palavra-chave `throw`

```
import java. util .EmptyStackException;
class FullStackException extends RuntimeException { }

// MyStackClass.java
// A classe MyStackClass exemplifica o uso da instrução throw, para lançar exceções
// Nesta classe é definida uma pilha simples para armazenar objetos genéricos.
public class MyStackClass {

    private static final int MAX_SIZE = 100; // tamanho máximo da pilha
    private Object stack[] = new Object[MAX_SIZE]; // vetor que armazena os objetos
    private int size = 0; // quantidade de elementos na pilha

    // remove um elemento da pilha
    public Object pop() {
        // pilha está vazia
        if (size == 0)
            throw new EmptyStackException(); // exceção do pacote java.util

        Object obj = stack[size - 1];
        stack[size - 1] = null;
        size--;
        return obj;
    }

    /* continua na próxima página */
```

Palavra-chave `throw`

```
    /* continua da página anterior */

    // insere um novo elemento na pilha
    public void push(Object obj) {
        // pilha está cheia
        if (size == MAX_SIZE)
            throw new FullStackException(); // exceção não existente, está declarada neste arquivo

        stack[size] = obj;
        size++;
    }
}
```

Palavra-chave `throw`

Lançando exceções

- A palavra-chave `throw` é utilizada para lançar exceções, ou seja, para indicar que alguma situação anormal ocorreu.
- A instrução `throw` deve especificar algum objeto derivado da classe `Throwable`.
- Um objeto pode ser lançado com uma descrição do problema ocorrido, bastando passar como argumento uma mensagem para o construtor da exceção.
- Quando o método `toString` é chamado a partir de um objeto `Throwable`, a string resultante inclui a descrição passada no construtor da exceção, ou simplesmente o nome da classe caso nenhuma descrição tenha sido fornecida.
- Exceções podem ser lançadas em construtores. Quando uma situação anormal é detectada em um construtor, uma exceção deve ser lançada para evitar a criação de um objeto mal-formado.

Palavra-chave `throw`

Relançando exceções

- Uma exceção é relançada quando um bloco `catch`, ao receber uma exceção, decide que não pode processá-la ou pode processá-la somente parcialmente.
- Nesse caso, a exceção pode ser relançada pela instrução `throw` seguido da referência ao objeto de exceção que acabou de ser capturado.
- Desse modo, relançar a exceção posterga o tratamento completo da exceção para outro bloco `catch` associado com um bloco `try` externo.
- No caso geral, exceções não podem ser relançadas a partir de um bloco `finally`, dado que a referência ao objeto de exceção saiu de seu escopo.

Palavra-chave `throw`

Relançando exceções

- Quando uma exceção é relançada para um bloco `try` externo, primeiramente o bloco `finally` corrente é executado.
- Recomenda-se evitar lançar novas exceções em um bloco `finally` e, se não for possível evitar, deve-se procurar envolver as novas exceções por um bloco `try..catch`. Isso porque se uma exceção não é capturada ao chegar em um bloco `finally` e nele uma nova exceção é lançada, mas não capturada dentro desse bloco, então a primeira exceção é perdida.
- Uma alternativa que permite lançar uma nova exceção sem perder a exceção original consiste no processo de exceções em cadeia.

Bloco `finally`

```
// UsingExceptions1.java
// Demonstração do tratamento de exceções com o mecanismo try...catch...finally

public class UsingExceptions1 {

    public static void main(String args[]) {
        try {
            throwException();
        } catch (Exception exception) { // captura exceção lançada pelo método throwException
            System.err.println("Exceção tratada em main");
        } // fim try

        doesNotThrowException();
    } // fim main

    /* continua na próxima página */
```

Bloco `finally`

```
    /* continua da página anterior */

    // Exemplifica o bloco finally quando uma exceção é lançada
    public static void throwException() throws Exception {

        try { // lança uma exceção e faz sua captura
            System.out.println("Método throwException");
            throw new Exception(); // lança exceção
            // qualquer código nesta parte não é executado
        } catch (Exception exception) { // captura exceção lançada em try
            System.err.println("Exceção tratada no método throwException");
            throw exception; // relança a exceção para posterior tratamento
            // qualquer código nesta parte não é executado
        } finally { // executa independentemente do que ocorre no bloco try...catch
            System.err.println("Bloco finally executado em throwException");
        } // fim try

        // qualquer código nesta parte não é executado
        // System.out.println("Fim do método throwException");

    } // fim método throwException

    // Exemplifica o bloco finally quando nenhuma exceção é lançada
    public static void doesNotThrowException() {
        try {
            System.out.println("Método doesNotThrowException");
        } catch (Exception exception) { // não é executado
            System.err.println(exception);
        } finally { // executa independentemente do que ocorre no bloco try...catch
            System.err.println("Bloco finally executado em doesNotThrowException");
        } // fim try

        System.out.println("Fim do método doesNotThrowException");
    } // fim método doesNotThrowException
} // fim classe UsingExceptions1
```

Retrocesso da pilha de execução

- Quando uma exceção é lançada mas não capturada em um dado escopo, é feita a tentativa de capturar a exceção em um bloco `try` externo.
- Esse processo é denominado de retrocesso da pilha de execução, o que significa que o método cuja exceção não foi capturada irá terminar, todas suas variáveis locais saem de escopo, e o controle retorna ao método que originalmente fez a chamada (próximo método da pilha de execução).
- Se um bloco `try` envolve a chamada, então é feita a tentativa de capturar a exceção. Se essa captura falhar ou se a chamada do método não está contida por um bloco `try`, então mais um retrocesso de pilha é efetuado.

Retrocesso da pilha de execução

```
// UsingExceptions2.java
// Demonstração do retrocesso da pilha em tratamento de exceções
public class UsingExceptions2 {
    public static void main(String args[]) {
        try {
            throwException();
        } catch (Exception exception) { // exceção lançada em throwException
            System.err.println("Exceção tratada no método main");
        } // fim try
    } // fim main

    // throwException lança uma exceção que não é tratada neste método
    public static void throwException() throws Exception {
        try {
            System.out.println("Método throwException");
            throw new Exception();
        } catch (RuntimeException runtimeException) { // captura outro tipo de exceção
            System.err.println("Exceção tratada no método throwException");
        } finally {
            System.err.println("Bloco finally é sempre executado");
        } // fim try
    } // fim método throwException
} // fim classe UsingExceptions2
```

Informações de `Throwable`

Extraindo dados de um exceção

- A classe `Throwable` oferece métodos com informações úteis para teste e depuração do código:
  - O método `printStackTrace` imprime a pilha de execução na saida de erro.
  - O método `getStackTrace` retorna informações da pilha de execução para que o programador possa criar uma impressão personalizada.
  - O método `getMessage` retorna a descrição da exceção, caso ela tenha sido passada como argumento do construtor.
  - O método `toString` retorna o tipo da exceção e sua descrição.

Informações de `Throwable`

Extraindo dados de um exceção

- Quando uma exceção não é capturada, a JVM executa o tratador de exceções default. Esse tratador imprime o tipo da exceção, sua descrição e a pilha de execução completa.
- Recomenda-se nunca deixar um bloco `catch` vazio pois isso implicaria em ignorar completamente a exceção. No mínimo deve ser chamado o método `printStackTrace` para ao menos imprimir uma mensagem de que uma exceção ocorreu.

Informações de `Throwable`

```
// UsingExceptions3.java
// Demonstrando os métodos getMessage e printStackTrace da classe Exception.
public class UsingExceptions3 {
    public static void main(String args[]) {
        try {
            method1();
        } catch (Exception exception) { // captura exceção lançada pelo método method1
            System.err.printf("%s\n\n", exception.getMessage());
            exception.printStackTrace(); // print exception stack trace

            // obtém a informação da pilha de execução
            StackTraceElement[] traceElements = exception.getStackTrace();

            System.out.println("\nStack trace from getStackTrace:");
            System.out.println("Class\t\tFile\t\tLine\tMethod");

            // laço sobre os elementos de pilha de execução extraindo suas informações
            for (StackTraceElement element : traceElements) {
                System.out.printf("%s\t", element.getClassName());
                System.out.printf("%s\t", element.getFileName());
                System.out.printf("%s\t", element.getLineNumber());
                System.out.printf("%s\n", element.getMethodName());
            } // fim for
        } // fim try
    } // fim main

    /* continua na próxima página */
```

Informações de `Throwable`

```
    /* continua da página anterior */

    // method1 chama method2; exceção é lançada de volta para o main
    public static void method1() throws Exception {
        method2();
    } // fim method1

    // method2 chama method3; exceção é lançada de volta para method1
    public static void method2() throws Exception {
        method3();
    } // fim method2

    // method3 lança uma exceção para method2
    public static void method3() throws Exception {
        throw new Exception("Exception thrown in method3");
    } // fim method3
} // fim classe UsingExceptions3
```

Exceções em cadeia

- Algumas vezes um método responde a uma exceção lançando um outro tipo de exceção que é mais específico à aplicação corrente.
- Se um bloco `catch` lança uma nova exceção, as informações da exceção original são perdidas. Para evitar essa situação, a linguagem Java possibilita lançar exceções em cadeia.
- A linguagem Java permite lançar uma sequência de exceções, de tal modo que a informação completa da pilha de execução é mantida, desde a primeira exceção.
- O processo de lançar exceções em cadeia é realizado passando a última exceção como argumento para o construtor da nova exceção.

Exceções em cadeia

```
// UsingChainedExceptions.java
// Demonstrando exceções encadeadas.
public class UsingChainedExceptions {
    public static void main(String args[]) {
        try {
            method1();
        } catch (Exception exception) { // captura exceção lançada por method1
            exception.printStackTrace();
        } // fim try
    } // fim main

    // method1 chama method2; lança uma exceção de volta para o método main
    public static void method1() throws Exception {
        try {
            method2();
        } catch (Exception exception) { // captura exceção lançada por method2
            throw new Exception("Exceção lançada por method1", exception);
        } // fim try
    } // fim method1

    // method2 chama method3; lança uma exceção de volta para method1
    public static void method2() throws Exception {
        try {
            method3();
        } catch (Exception exception) { // captura exceção lançada por method3
            throw new Exception("Exception thrown in method2", exception);
        } // fim try
    } // fim method2

    // method3 lança uma exceção de volta para method2
    public static void method3() throws Exception {
        throw new Exception("Exception thrown in method3");
    } // fim method3
} // fim classe UsingChainedExceptions
```

Novos tipos de exceção

Declarando novas classes de exceção

- Quando possível, uma classe deve utilizar exceções de tipos já existentes, ao invés de criar novos tipos.
- A Java API possui uma variedade grande de tipos de exceções que podem se enquadrar ao tipo de exceção que se deseja tratar.
- Ao implementar novas classes, é razoável imaginar que haverão comportamentos anormais de execução que não estão cobertos por nenhuma das classes de exceções existentes.
- Às vezes torna-se conveniente declarar classes de exceção personalizadas para as diferentes anormalidades que podem ocorrer em um novo aplicativo.
- Uma nova classe de exceção precisa estender uma classe de exceção existente para assegurar que a nova classe possa ser utilizada pelo mecanismo de tratamento de exceções da linguagem Java.

Novos tipos de exceção

Declarando novas classes de exceção

- Como qualquer outra classe, uma classe de exceção pode conter atributos e métodos.
- Normalmente, uma nova classe de exceção deve conter pelo menos quatro construtores:
  1. Um construtor sem argumentos que passa uma mensagem de erro qualquer para o construtor da superclasse.
  2. Um construtor que recebe uma `String` (mensagem de erro) passada para o construtor da superclasse.
  3. Um construtor que recebe uma referência `Throwable` (para exceções em cadeia) passada para o construtor da superclasse.
  4. Um construtor que recebe uma `String` (mensagem de erro) e uma referência `Throwable` (para exceções em cadeia), ambas passadas para o construtor da superclasse.

Novos tipos de exceção

Declarando novas classes de exceção

- Ao definir um novo tipo de exceção, estude as classes existentes na Java API e procure estender de uma classe de exceção relacionada.
- Exemplo: se uma nova classe de exceção está sendo criada para representar uma operação aritmética inválida, é recomendável estender da classe `ArithmeticException`.
- Para criar uma nova classe de exceção verificada, basta que ela não estenda a classe `RuntimeException`, o que obrigará os clientes dessa classe a respeitarem a política de capturar ou declarar.
- A convenção de nomes em Java diz que classes do tipo exceção devem ter o sufixo `Exception`.

Pré-condições e pós-condições

- Para facilitar a manutenção e depuração do código, recomenda-se especificar os estados esperados antes e depois da execução de um método. Esses estados são denominados de pré-condições e pós-condições, respectivamente.
- Uma pré-condição deve ser verdadeira assim que um método é chamado.
- As pré-condições descrevem restrições sob os parâmetros de um método ou sob atributos do objeto ou da classe.
- Se essas restrições não forem respeitadas, o método apresentará um comportamento indefinido.
- Uma pós-condição deve ser verdadeira quando as pré-condições são respeitadas e um método termina seu serviço com sucesso.
- As pós-condições descrevem restrições sob o valor de retorno ou sob qualquer outro efeito produzido pelo método.

Pré-condições e pós-condições

- As pré-condições e pós-condições de um método devem ser parte de sua especificação. Portanto, ao implementar um método, em sua documentação devem estar listados todas as pré e pós-condições para que os clientes saibam o que esperar da execução do método.
- Exemplo: considere um método `charAt` que retorna o caractere posicionado em um certo índice de uma cadeia de caracteres.
- Uma pré-condição para o método `charAt` é de que o índice seja maior ou igual a zero e menor do que o tamanho da `String`.
- Uma pós-condição para o método `charAt` consiste no retorno de um caractere que corresponde ao elemento da posição especificada pelo índice.

Assertivas

- Ao implementar uma classe, pode ser útil estabelecer invariantes (condições sempre verdadeiras) em algum ponto de um método. Em Java, as invariantes podem ser checadas por meio de assertivas (*assertions*).
- Pré-condições e pós-condições são invariantes a respeito do estado do método antes e depois de sua execução, logo podem ser verificadas por meio de assertivas.
- As assertivas em Java correspondem em uma expressão booleana que, se forem falsas, lançam um erro do tipo `AssertionError`.

```
assert <boolean expression> : 'minha mensagem de erro';
```

- As assertivas são utilizadas principalmente para depuração e identificação de erros lógicos da aplicação.

Assertivas

- Como as assertivas não foram feitas para tratamento de erros, seu uso se restringe ao processo de depuração do código, logo não devem ser habilitadas em fase de uso da aplicação. Em outras palavras, usuários do aplicativo não devem se deparar com `AssertionErrors` na execução normal do programa.
- Para habilitar as assertivas, deve ser utilizada a opção `-ea` na execução do programa.
- Exceções provocadas por assertivas não devem ser capturadas pelo programa. O programador deve simplesmente deixar o programa terminar e corrigir o código para que o erro não volte a se repetir.

```
java  -ea ClassName
```

Assertivas

```
import java.util.Arrays;

/*
 * ThreeSortTest.java
 * Classe que exemplifica o uso de assertivas para uma aplicação que ordena um vetor de inteiros com três elementos.
 */
public class ThreeSortTest {

    /*
     * O método switchPlaces troca a posição de dois elementos (i e i+1) em um vetor de inteiros
     * Pré-condições:
     * 1. v != null
     * 2. 0 <= i < v.length-1
     * Pós-condições (v2 é o vetor original):
     * 1. v[i] == v2[i+1]
     * 2. v[i+1] == v2[i]
     */
    public static void switchPlaces(int i, int v[]) {

        // Assegurar pré-condições
        assert (v != null) : "Vetor nulo";
        assert ((i>=0) && (i<v.length-1)) : "Índice fora dos limites";

        int aux1 = v[i];
        int aux2 = v[i+1];
        v[i] = v[i+1];
        v[i+1] = aux1;

        // Assegurar pós-condições
        assert ((v[i] == aux2) && (v[i+1] == aux1)) : "Troca falhou.";

    }

    /* continua na próxima página */
```

Assertivas

```
    /* continua da página anterior */

    /*
     * O método threeSort ordena de modo crescente um vetor de inteiros com três elementos
     * Pré-condições:
     * 1. v != null
     * 2. v.length == 3
     * Pós-condições:
     * 1. v[0] <= v[1] <= v[2]
     */
    public static void threeSort(int v[]) {

        // Assegurar pré-condições
        assert (v != null) : "Vetor nulo";
        assert (v.length == 3) : "Vetor não possui três elementos.";

        if (v[0] > v[1]) switchPlaces(0,v);
        if (v[1] > v[2]) switchPlaces(1,v);
        if (v[0] > v[1]) switchPlaces(0,v);

        // Assegurar pós-condições
        assert ((v[0] <= v[1]) && (v[1] <= v[2])) : "Ordenação falhou.";

    }

    public static void main(String[] args) {
        int v[] = {3,2,1};
        threeSort(v);
        System.out.println(Arrays.toString(v));
    }
}
```

Referências

1. Java: Como Programar, Paul Deitel & Heivey Deitel; Pearson; 7a. Ed. (no. chamada IMECC – 05.133 D368j)
2. Data Structures and Algorithms with Object Oriented Design Patterns in Java, Bruno Preiss; (http://www.brpreiss.com/books/opus6/)
3. The Java Tutorials (Oracle) (http://docs.oracle.com/javase/tutorial/)
4. Guia do Usuário UML, Grady Booch et. al.; Campus(1999)
5. Java Pocket Guide - Robert Liguori & Patricia Liguori; O'Reilley, 2008.