

TRATAMENTO DE ERROS COM EXCEÇÕES

ACH 2003 — COMPUTAÇÃO ORIENTADA A OBJETOS

Daniel Cordeiro

9 de março de 2016

Escola de Artes, Ciências e Humanidades | EACH | USP

```
Exception in thread "main" java.lang.NullPointerException at  
    java.io.Writer.write(Writer.java:157) at  
    java.io.PrintStream.write(PrintStream.java:462) at  
    java.io.PrintStream.print(PrintStream.java:584) at  
    java.io.PrintStream.println(PrintStream.java:700) at  
    Main.main(Main.java:21)
```

- Filosofia do Java: “Se o código não estiver bem formado, não o execute”
- Qual o melhor momento para detectar um erro?

- Filosofia do Java: “Se o código não estiver bem formado, não o execute”
- Qual o melhor momento para detectar um erro?
- **Em tempo de compilação!** Mas nem sempre isso é possível
- Como fazer isso em tempo de execução?

- Filosofia do Java: “Se o código não estiver bem formado, não o execute”
- Qual o melhor momento para detectar um erro?
- Em tempo de compilação! Mas nem sempre isso é possível
- Como fazer isso em tempo de execução?
 - Devolução de valor especial que representa erro
 - Alterar uma *flag* para alertar um erro
 - Problema: programadores tendem a se esquecer de verificar por erros (ou se acham invencíveis)

Quantos de vocês já usaram o `errno.h` em C?

- `E2BIG` Argument list too long (POSIX.1)
- `EACCES` Permission denied (POSIX.1)
- `EADDRINUSE` Address already in use (POSIX.1)
- `EADDRNOTAVAIL` Address not available (POSIX.1)
- `EAFNOSUPPORT` Address family not supported (POSIX.1)
- `EAGAIN` Resource temporarily unavailable (may be the same value as `EWOULDBLOCK`) (POSIX.1)
- `EALREADY` Connection already in progress (POSIX.1)
- `EBADF` Invalid exchange
- `EBUSY` Device or resource busy (POSIX.1)
- `ECANCELED` Operation canceled (POSIX.1)
- `ECHILD` No child processes (POSIX.1)
- `ECHRNG` Channel number out of range
- `ECOMM` Communication error on send
- `ECONNABORTED` Connection aborted (POSIX.1)
- `ECONNREFUSED` Connection refused (POSIX.1)
- `ECONNRESET` Connection reset (POSIX.1)
- `EDEADLK` Resource deadlock avoided (POSIX.1)
- `EEXIST` File exists (POSIX.1)
- `EFAULT` Bad address (POSIX.1)
- `EFBIG` File too large (POSIX.1)
- `EHOSTDOWN` Host is down
- `EHOSTUNREACH` Host is unreachable (POSIX.1)
- `EIDRM` Identifier removed (POSIX.1)
- `EILSEQ` Illegal byte sequence (POSIX.1, C99)
- `EINPROGRESS` Operation in progress (POSIX.1)
- `EINTR` Interrupted function call (POSIX.1);
- `EINVAL` Invalid argument (POSIX.1)
- `EIO` Input/output error (POSIX.1)
- `EMULTIHOP` Multihop attempted (POSIX.1)
- `ENAMETOOLONG` Filename too long (POSIX.1)
- `ENETDOWN` Network is down (POSIX.1)
- `ENETRESET` Connection aborted by network (POSIX.1)
- `ENETUNREACH` Network unreachable (POSIX.1)
- `ENODEV` No such device (POSIX.1)
- `ENOENT` No such file or directory (POSIX.1)
- `ENOEXEC` Exec format error (POSIX.1)
- `ENOKEY` Required key not available
- `ENOLCK` No locks available (POSIX.1)
- `ENOLINK` Link has

O QUE É UMA EXCEÇÃO?

Definição

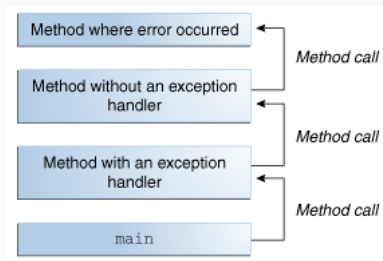
Uma exceção é um evento que ocorre durante a execução de um programa e que interrompe o seu fluxo normal execução.

- Transforma o erro em um elemento formal da linguagem
- Obriga o tratamento do erro na hora que ele acontece
- Mesmo que você não saiba o que fazer com o erro naquele momento, você para a execução. Alguém, em algum lugar, deve saber o que fazer — *cadeia de comando*
- Exceções permitem escrever todo o código supondo que nada deu errado e tratar os casos de erro separadamente

- Um erro é sinalizado pro interpretador pelo código que detectou o erro
- O programa cria um objeto, chamado **objeto de exceção** (*exception object*), que contém informações sobre o erro:
 - tipo do erro;
 - estado do programa quando o erro aconteceu;
 - etc.
- O ato de criar um objeto de exceção e passá-lo para o sistema de execução é chamado de **lançar uma exceção** (*throwing an exception*)

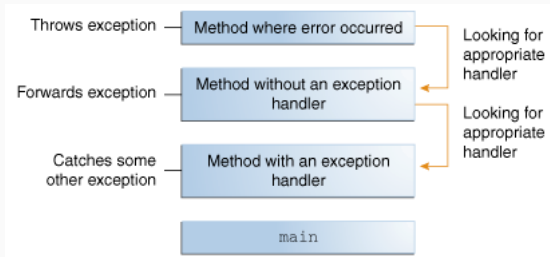
TRATAMENTO DE EXCEÇÕES

- Após o lançamento da exceção, o sistema de execução tenta encontrar “algo” para tratá-la
- A lista dos candidatos a tratadores de exceção é a lista ordenada dos métodos que foram sendo chamados até que o erro ocorreu — a pilha de execução



TRATAMENTO DE EXCEÇÕES

- O sistema de execução procura na pilha de execução pelo código que pode tratar a exceção
- Esse bloco de dados é chamado de **tratador de exceção** (*exception handler*)
- A busca começa no método que gerou o erro e segue pela pilha de execução em ordem reversa até que o tratador de exceção apropriado (aquele que sabe tratar exceções de um tipo de objeto de exceção) seja encontrado
- Dizemos que o tratador **captura a exceção**



Um código em Java válido precisa honrar o requisito de *Capturar ou Especificar* exceções:

- Ou o método captura a exceção, usando uma expressão **try**
- Ou diz explicitamente que ele pode gerar uma exceção, com a palavra chama **throws**

TRÊS TIPOS DE EXCEÇÕES

Exceção verificada (*checked exception*) são exceções relacionadas à aplicação que um código bem escrito sabe antecipar e tratar. Ex: `java.io.FileNotFoundException`

Erro indicam situações excepcionais externas às aplicações, que normalmente não são antecipadas e nem podem ser corrigidas. Ex: `java.io.IOException`

Exceção de execução (*runtime exception*) são relacionadas à aplicação, mas que não podem ser tratadas. Normalmente indicam um *bug* ou uso impróprio de uma API. Ex: `java.lang.NullPointerException`

Observação:

Exceções verificadas sempre devem ser capturadas ou especificadas.

TRÊS TIPOS DE EXCEÇÕES

Os tipos de exceções são especificados pelas suas classes pai.

Exceção verificada : `java.lang.Exception`

Erro : `java.lang.Error`

Exceção de execução : `java.lang.RuntimeException`

Observação:

Exceções verificadas sempre devem ser capturadas ou especificadas.

CAPTURA E TRATAMENTO DE EXCEÇÃO

```
// Nota: Essa classe não compila
import java.io.*; import java.util.List; import java.util.ArrayList;

public class ListOfNumbers {
    private List<Integer> list;
    private static final int SIZE = 10;

    public ListOfNumbers () {
        list = new ArrayList<Integer>(SIZE);
        for (int i = 0; i < SIZE; i++) {
            list.add(new Integer(i));
        }
    }

    public void writelist() {
        // O construtor FileWriter pode lançar um IOException,
        // que deve ser capturado
        PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));

        for (int i = 0; i < SIZE; i++) {
            // get(int) pode lançar um IndexOutOfBoundsException
            out.println("Valor em: " + i + " = " + list.get(i));
        }
        out.close();
    }
}
```

```
try {  
    // código que pode gerar exceção  
}  
blocos catch e finally . . .
```

- O primeiro passo para tratar uma exceção é colocar o código que pode gerar a exceção dentro de um bloco **try**
- Você pode colocar um **try** para cada linha de código que pode gerar uma exceção e escrever o tratador de exceção para cada um deles
- Ou você pode colocar várias linhas em um único **try** e associar múltiplos tratadores

O BLOCO TRY

```
private List<Integer> list;
private static final int SIZE = 10;

public void writeList() {
    PrintWriter out = null;
    try {
        System.out.println("Entrou no bloco try");
        out = new PrintWriter(new FileWriter("OutFile.txt"));
        for (int i = 0; i < SIZE; i++) {
            out.println("Valor em: " + i + " = " + list.get(i));
        }
    }
    blocos catch e finally . . .
}
```



```
try {  
    // código que pode gerar vários tipos de exceções  
} catch (TipoDaExceção nome) {  
    // código que trata o primeiro tipo de exceção  
} catch (TipoDaExceção nome) {  
    // código que trata o outro tipo de exceção  
}
```

- O bloco `catch` implementa o código de tratamento da exceção
- `TipoDaExceção` (qualquer subclasse de `Throwable`) declara o tipo de exceção que aquele bloco sabe tratar
- O tratador tem acesso ao objeto de exceção pela variável `nome`

```
try {  
    // método writeList() do exemplo anterior  
} catch (IndexOutOfBoundsException e) {  
    System.err.println("IndexOutOfBoundsException: "  
        + e.getMessage());  
} catch (IOException e) {  
    System.err.println("Capturou um IOException: "  
        + e.getMessage());  
}
```

Em Java ≥ 7 , se você tiver um único tratador para vários tipos de exceções, você pode listar os múltiplos tipos de exceções separados por “|” para reduzir código duplicado.

```
catch (IOException|SQLException ex) {  
    logger.log(ex);  
    throw ex;  
}
```

- Um bloco **finally** **sempre** é executado depois que o bloco **try** termina
- Ele serve pra garantir que o código seja executado mesmo que uma exceção inesperada ocorra
- **finally** pode ser usado em outras situações, para garantir que o programador não se esqueça de executar “código de limpeza” após um **return**, **continue** ou **break**

FINALLY E O EXEMPLO

- O exemplo abre um arquivo usando a classe `PrintWriter`
- O programa deve sempre fechar o arquivo antes de sair do método `writeList`
- Porém o método pode terminar de três modos diferentes:
 1. se o `new FileWriter` lançar uma `IOException`
 2. se `list.get(i)` falhar e lançar um `IndexOutOfBoundsException`
 3. ou se tudo funcionar e o bloco `try` terminar normalmente

```
try {  
    // método writeList() do exemplo anterior  
} catch (IndexOutOfBoundsException e) {  
    System.err.println("IndexOutOfBoundsException: "  
        + e.getMessage());  
} catch (IOException e) {  
    System.err.println("Capturou um IOException: "  
        + e.getMessage());  
}
```

FINALLY E O EXEMPLO

- O exemplo abre um arquivo usando a classe `PrintWriter`
- O programa deve sempre fechar o arquivo antes de sair do método `writeList`
- Porém o método pode terminar de três modos diferentes:
 1. se o `new FileWriter` lançar uma `IOException`
 2. se `list.get(i)` falhar e lançar um `IndexOutOfBoundsException`
 3. ou se tudo funcionar e o bloco `try` terminar normalmente

Solução:

```
finally {  
    if (out != null) {  
        System.out.println("Fechando PrintWriter");  
        out.close();  
    } else {  
        System.out.println("PrintWriter não foi aberto");  
    }  
}
```

- Expressões do tipo *try-com-recursos* são usadas para declarar recursos que devem ser fechados depois de serem usados
- A expressão garante que todos os recursos serão fechados após serem utilizados
- Qualquer objeto que implemente a interface `java.lang.AutoCloseable` pode ser usado como recurso

Se Java \geq 7:

```
static String readFirstLineFromFile(String path)
    throws IOException {
    try (BufferedReader br =
        new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
}
```

- Expressões do tipo *try-com-recursos* são usadas para declarar recursos que devem ser fechados depois de serem usados
- A expressão garante que todos os recursos serão fechados após serem utilizados
- Qualquer objeto que implemente a interface `java.lang.AutoCloseable` pode ser usado como recurso

Se Java < 7:

```
static String rflffWithFinallyBlock(String path)
    throws IOException {
    BufferedReader br = new BufferedReader(
        new FileReader(path));
    try { return br.readLine(); }
    finally {
        if (br != null) br.close();
    }
}
```


O QUE É EXECUTADO SE OCORRER UMA EXCEÇÃO?

```
public void writeList() {
    PrintWriter out = null;

    try {
        System.out.println("Entrou no bloco try");

        out = new PrintWriter(new FileWriter("OutFile.txt"));
        for (int i = 0; i < SIZE; i++) {
            out.println("Valor em: " + i + " = " + list.get(i));
        }
    } catch (IndexOutOfBoundsException e) {
        System.err.println("Capturou um IndexOutOfBoundsException: "
            + e.getMessage());
    } catch (IOException e) {
        System.err.println("Capturou um IOException: " + e.getMessage());
    } finally {
        if (out != null) {
            System.out.println("Fechando PrintWriter");
            out.close();
        }
        else { System.out.println("PrintWriter não estava aberto"); }
    }
}
```

COMO INDICAR QUE UM MÉTODO PODE LANÇAR EXCEÇÃO?

- No exemplo anterior nós sabíamos como tratar as possíveis exceções
- Mas em alguns casos é melhor deixar um outro método na pilha de execução trata a exceção
- A assinatura de um método pode indicar quais são as exceções que devem ser verificadas pelos outros métodos que o chamarem

No exemplo faríamos¹:

```
public void writeList() throws IOException,  
                                IndexOutOfBoundsException  
{  
    // código do método sem o bloco try  
}
```

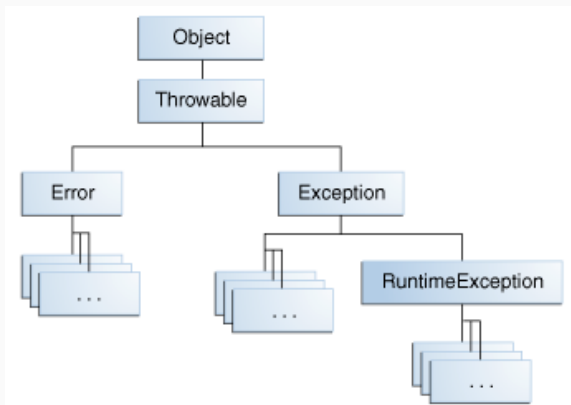
¹IndexOutOfBoundsException não é uma exceção verificada e pode ser omitida da declaração do método.

COMO LANÇAR EXCEÇÕES?

- Antes de poder tratar uma exceção, você precisa poder criar e lançar a exceção
- Um objeto de exceção é lançado com a expressão **throw**
- Você pode criar seus próprios tipos de exceção, criando subclasses de **Throwable**

```
public Object pop() throws EmptyStackException {  
    Object obj;  
  
    if (size == 0) {  
        throw new EmptyStackException();  
    }  
  
    obj = objectAt(size - 1);  
    setObjectAt(size - 1, null);  
    size--;  
    return obj;  
}
```

A CLASSE THROWABLE E SUAS SUBCLASSES



- Programas capturam subclasses de **Exception**; elas indicam que um problema ocorreu, mas que ele não é muito sério (maioria das exceções de Java)
- Já um **Error** indica um problema sério que uma aplicação não deveria tentar capturar

EXCEÇÕES ENCADEADAS

- É comum que o tratamento de uma exceção seja o lançamento de uma nova exceção
- A primeira exceção *causa* a segunda
- Para saber qual a causa de uma exceção, Java permite encadear as exceções com a ajuda dos seguintes métodos:

```
Throwable getCause()  
Throwable initCause(Throwable)  
Throwable(String, Throwable)  
Throwable(Throwable)
```

Exemplo:

```
try {  
    // ...  
} catch (IOException e) {  
    throw new SampleException("Outra IOException", e);  
}
```

CRIAÇÃO DE NOVOS TIPOS DE EXCEÇÕES

- Quando decidir que seu código deve lançar uma exceção, você pode lançar uma das exceções definidas pela plataforma Java ou criar uma nova
- Você deveria criar uma nova se responder “sim” a alguma dessas perguntas:
 - eu preciso de um tipo de exceção que não é representada pelas exceções de Java?
 - meus usuários gostariam de poder diferenciar as suas exceções das exceções dos outros?
 - o seu código lança 2 ou mais exceções relacionadas?
 - se você usar as exceções criadas por outra pessoa, seus usuários terão acesso a elas? Equivalente a: meu pacote deve ser independente e autocontido?

VANTAGENS DO USO DE EXCEÇÕES

1: SEPARAR O TRATAMENTO DE ERRO DO RESTO DO CÓDIGO

lêArquivo

```
abre o arquivo;  
calcula seu tamanho;  
aloca memória suficiente pro arquivo;  
copia o conteúdo do arquivo na memória;  
fecha o arquivo;
```


1: SEPARAR O TRATAMENTO DE ERRO DO RESTO DO CÓDIGO

O que acontece:

- se o arquivo não puder ser aberto?
- se o tamanho do arquivo não puder ser determinado?
- se não houver memória suficiente disponível?
- se ocorrer um erro na leitura?
- se o arquivo não puder ser fechado?

1: SEPARAR O TRATAMENTO DE ERRO DO RESTO DO CÓDIGO

```
errorCodeType lêArquivo {
    initialize errorCode = 0;
    abre o arquivo;
    if (theFileIsOpen) {
        calcula o tamanho do arquivo;
        if (gotTheFileLength) {
            aloca memória suficiente pro arquivo;
            if (gotEnoughMemory) {
                copia o conteúdo do arquivo na memória;
                if (readFailed) {
                    errorCode = -1;
                }
            } else {
                errorCode = -2;
            }
        } else {
            errorCode = -3;
        }
        fecha o arquivo;
        if (theFileDidntClose && errorCode == 0) {
            errorCode = -4;
        } else { errorCode = errorCode and -4;
        }
    } else { errorCode = -5; }
    return errorCode;
}
```

1: SEPARAR O TRATAMENTO DE ERRO DO RESTO DO CÓDIGO

```
lêArquivo {  
    try {  
        abre o arquivo;  
        calcula seu tamanho;  
        aloca memória suficiente pro arquivo;  
        copia o conteúdo do arquivo na memória;  
        fecha o arquivo;  
    } catch (aberturaArquivoFalhou) {  
        façaAlgo;  
    } catch (calcularTamanhoFalhou) {  
        façaAlgo;  
    } catch (alocarMemóriaFalhou) {  
        façaAlgo;  
    } catch (leituraFalhou) {  
        façaAlgo;  
    } catch (fechamentoFalhou) {  
        façaAlgo;  
    }  
}
```

2: PROPAGAÇÃO DE ERROS PELA PILHA DE EXECUÇÃO

Suponha o exemplo:

```
método1 {  
    chama método2;  
}
```

```
método2 {  
    chama método3;  
}
```

```
método3 {  
    chama lêArquivo;  
}
```

2: PROPAGAÇÃO DE ERROS PELA PILHA DE EXECUÇÃO

Suponha que o método1 seja o único interessado nos erros de lêArquivo

```
método1 {  
    TipoDoCódigoErro erro;  
    erro = chama método2;  
    if (erro)  
        tratamentoDoErro;  
    else  
        prossiga;  
}
```

```
TipoDoCódigoErro método2 {  
    TipoDoCódigoErro erro;  
    erro = chama método3;  
    if (erro)  
        return erro;  
    else  
        prossiga;  
}  
TipoDoCódigoErro método3 {  
    TipoDoCódigoErro erro;  
    erro = chama lêArquivo;  
    if (erro)  
        return erro;  
    else  
        prossiga;  
}
```

2: PROPAGAÇÃO DE ERROS PELA PILHA DE EXECUÇÃO

Em Java:

```
método1 {  
    try {  
        chama método2;  
    } catch (exception e) {  
        tratamentoDoErro;  
    }  
}
```

```
método2 throws exception {  
    chama método3;  
}
```

```
método3 throws exception {  
    chama lêArquivo;  
}
```

3: AGRUPAMENTO E DIFERENCIAÇÃO DE TIPOS DE ERROS

- Exceções são objetos. Categorização dos erros é uma consequência natural da hierarquia de classes
 - Um bom exemplo é o `java.io.IOException` e seus descendentes
 - Se eu quero tratar um erro específico, eu posso fazer:
- independentemente do tipo, e tratá-los do mesmo jeito:

```
catch (FileNotFoundException e) {  
    ...  
}
```

```
catch (IOException e) {  
    // A saída vai para System.err.  
    e.printStackTrace();  
    // Envia a saída para stdout.  
    e.printStackTrace(System.out);  
}
```

TESTES DE EXCEÇÕES

JUnit permite testar se exceções:

- foram lançadas como era esperado

```
@Test(expected = IndexOutOfBoundsException.class)
public void empty() {
    new ArrayList<Object>().get(0);
}
```

- foram lançadas com os atributos (ex: mensagem) esperados:

```
@Test
public void testExceptionMessage() {
    try {
        new ArrayList<Object>().get(0);
        fail("Expected an IndexOutOfBoundsException to be thrown");
    } catch (IndexOutOfBoundsException anIndexOutOfBoundsException) {
        assertThat(anIndexOutOfBoundsException.getMessage(),
            is("Index: 0, Size: 0"));
    }
}
```



```
Exception in thread "main" java.lang.NullPointerException at  
    java.io.Writer.write(Writer.java:157) at  
    java.io.PrintStream.write(PrintStream.java:462) at  
    java.io.PrintStream.print(PrintStream.java:584) at  
    java.io.PrintStream.println(PrintStream.java:700) at  
    Main.main(Main.java:21)
```

- The Java™ Tutorials – Exceptions: <http://docs.oracle.com/javase/tutorial/essential/exceptions/>
- Thinking in Java, 3rd Edition. Bruce Eckel. Disponível gratuitamente em <http://www.mindviewinc.com/Books/TIJ/>