Linguagens de Programação

Tratamento de Exceções

Andrei Rimsa Álvares
andrei@cefetmg.br

Sumário

- Introdução
- Motivação
- Exceções
- Tratamento de exceções
- Tipos de exceções
- Tratar ou propagar?
- Lançamento de exceções
- Exceções próprias
- Bloco `finally`
- Resumo

INTRODUÇÃO

Linguagens de Programação

Introdução

- Em um mundo perfeito, usuários nunca entram com dados inválidos e o código nunca tem bugs

- Agora é preciso lidar com problemas do mundo real, de forma que dados podem ser malformados e códigos podem ter erros

E se o programa abortar e você perder tudo?

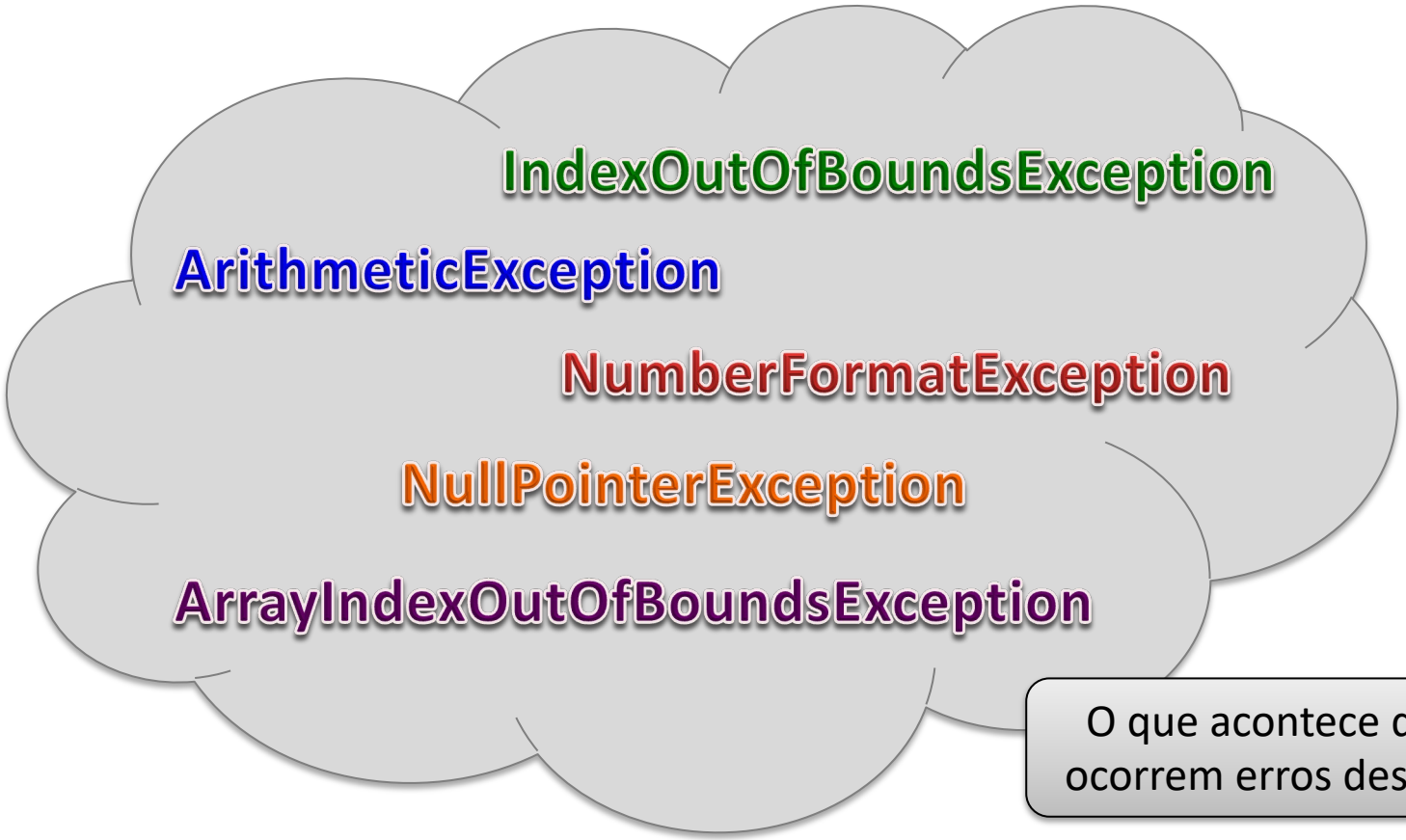
Introdução

- Por mais que você seja um bom programador, não se consegue controlar tudo

- Coisas inesperadas ainda podem ocorrer
  - O arquivo pode não estar lá
  - O servidor pode estar fora do ar
  - A impressora pode estar sem papel
  - O disco pode estar cheio
  - Pode acabar a memória do sistema
  - Entre outros

Introdução

- Em Java, é comum se deparar com algumas situações inesperadas em tempo de execução



Introdução

- É preciso se resguardar ao chamar um **método de risco**
  - Como lidar com a situação caso alguma coisa dê errado?

MOTIVAÇÃO

Contrato

- Em programação, um método é uma espécie de contrato

  - Se as entradas estiverem corretas, ele se comporta como esperado
  - Mas como proceder se as entradas forem inválidas?

Exemplo

- Considere o método **sacar** de uma conta bancária com saldo inicial de R$0,00 e limite de R$100,00

```
Conta minhaConta = new Conta();
minhaConta.setLimite(100);
minhaConta.depositar(100);
minhaConta.sacar(1000);
```

Qual o valor na conta após o saque?
-900, -800, 0, 100?

Caso não seja possível fazer a operação, como sinalizar o erro?

Exemplo

- Em programas reais é mais comum que quem saiba tratar o erro seja quem chamou o método e não onde o erro foi gerado

- **Solução inicial:** retornar um valor lógico (*boolean*) indicando sucesso ou falha na operação

```
boolean sacar(double quantidade) {
   // verificar se não posso sacar até saldo+limite
   if (quantidade > (this.saldo + this.limite))
       return false;

   this.saldo = this.saldo - quantidade;
   return true;
}
```

Como usar a operação **sacar** agora?

Exemplo

- Para verificar se a operação teve sucesso, basta verificar o valor de seu retorno e contornar a situação de acordo

```
Conta minhaConta = new Conta();
minhaConta.setLimite(100);
minhaConta.deposita(100);
if (!minhaConta.sacar(1000)) {
   System.out.println("Não foi possível sacar");
}
```

Qual o problema dessa abordagem?

Exemplo

- Para o exemplo anterior foi preciso saber/lembrar que o método sacar retorna um valor indicando sucesso ou não; mas não existe **obrigatoriedade**

- Esquecer de testar poderia trazer consequências drásticas, como uma máquina de autoatendimento liberar dinheiro sem que a operação tivesse êxito

```
Conta minhaConta = new Conta();
minhaConta.deposita(100);

// ...
double valor = 5000;
// vai retornar false, mas ninguém verifica!
minhaConta.sacar(valor);
caixaEletronico.emite(valor);
```

Existe outro problema com essa abordagem?

Exemplo

- Uma outra solução (mas que também não resolve o problema anterior) é retornar um código de erro que indica o tipo de falha ocorrida e zero para sucesso

```
static int SUCESSO = 0;
static int VALOR_INVALIDO = -1;
static int SALDO_INSUFICIENTE = -2;

int sacar(double quantidade) {
   // quantidade inválida
   if (quantidade <= 0)
       return VALOR_INVALIDO;
   // verificar se não posso sacar até saldo+limite
   else if (quantidade > this.saldo + this.limite)
       return SALDO_INSUFICIENTE;

   this.saldo = this.saldo - quantidade;
   return SUCESSO;
}
```

Que outro(s) problema(s) essa estratégia apresenta?

Exemplo

- Como cada método pode criar seus próprios códigos, o usuário precisa conhecer os possíveis valores de retorno (normalmente através de extensa documentação)
  - Tarefa nada prática!
- Além disso, como retornar um código de erro se o método já possui um outro tipo de retorno?

```
Cliente procuraCliente(int id) {
   if (isIdInvalido(id)) {
       // Como avisar o método chamador que ocorreu um erro?
   } else {
       Cliente cliente = new Cliente();
       cliente.setId(id);
       // ...
       return cliente;
   }
}
```

EXCEÇÕES


Windows
An exception 06 has occured at 0028:C11B3ADC in VxD DiskTSD(03) +
00001660, This was called from 0028:C11B40C8 in VxD voltrack(04) +
00000000, It may be possible to continue normally.
* Press any key to attempt to continue.
* Press CTRL+ALT+RESET to restart your computer. You will
lose any unsaved information in all applications.
Press any key to continue

Exceções

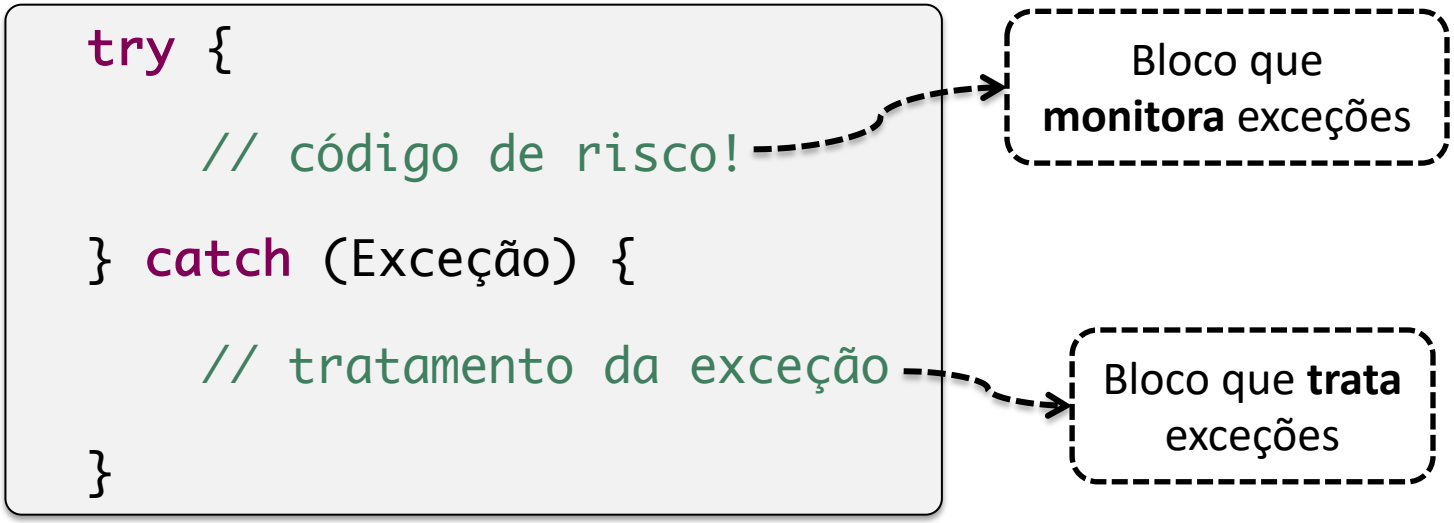
- Uma exceção é um evento, normalmente associado a um erro, que ao ser disparado interrompe o fluxo normal de execução da aplicação
  - Ex.: divisões por zero, acesso a posições inválidas em listas ou arranjos, falta de memória, disco cheio, ...

- Como o próprio nome indica, exceções são situações que não ocorrem com frequência, mas aplicações devem estar preparadas para tratar essas exceções sem abortar o programa

Tratamento de Exceções

- O processamento especial que pode ser exigido pela detecção de uma exceção é chamado **tratador** (**ou manipulador**) **de exceção**

- O **tratador de exceção** é o bloco de código para onde a execução do programa é desviada quando ocorre uma exceção



Suporte

- Muitas linguagens de programação possuem suporte a exceções e tratamento de exceções

1) Actionscript
2) Ada
3) C++
4) C#
5) Delphi
6) ECMAScript
7) Eiffel
8) Java
9) Lisp
10) ML
11) Objective-C
12) OCaml
13) PHP (>= 5.0)
14) PL/1
15) PL/SQL
16) Prolog
17) Python
18) Ruby
19) Scala
20) ...

Onde começou toda essa história?♣

Vamos ver a implementação em Java e C++

♣: http://en.wikipedia.org/wiki/Exception_handling#History

Exceções Geradas

- Exceções podem ser geradas
  - Pelo ambiente de execução (Em Java, pela JVM)
    - Normalmente erros fundamentais causados por erros que violam a natureza da linguagem
    - Restrições do próprio ambiente
  - Manualmente pelo seu código
    - Normalmente utilizadas para notificar o método chamador de alguma condição de erro

Exemplo

Qual o resultado da execução desse programa?

- Exemplo de exceções em Java

```
 1. class DivisaoPorZero {
 2.     static final int MULTIPLICACAO = 3;
 3.     static final int DIVISAO = 4;
 4.
 5.     static int multiplicacao(int a, int b) {
 6.         return a * b;
 7.     }
 8.     static int divisao(int a, int b) {
 9.         return a / b;
10.     }
11.     static int operacao(int operacao, int a, int b) {
12.         switch (operacao) {
13.             case MULTIPLICACAO: return multiplicacao(a, b);
14.             case DIVISAO: return divisao(a, b);
15.             default: return 0;
16.         }
17.     }
18.     public static void main(String[] args) {
19.         int d = operacao(DIVISAO, 10, 0);
20.         System.out.println(d);
21.     }
22.}
```

Rastro da Pilha

- O programa foi abortado por causa de uma exceção não tratada

```
bin — bash — 70×6
$ java DivisaoPorZero
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at DivisaoPorZero.divisao(Teste.java:10)
        at DivisaoPorZero.operacao(Teste.java:15)
        at DivisaoPorZero.main(Teste.java:20)
$
```

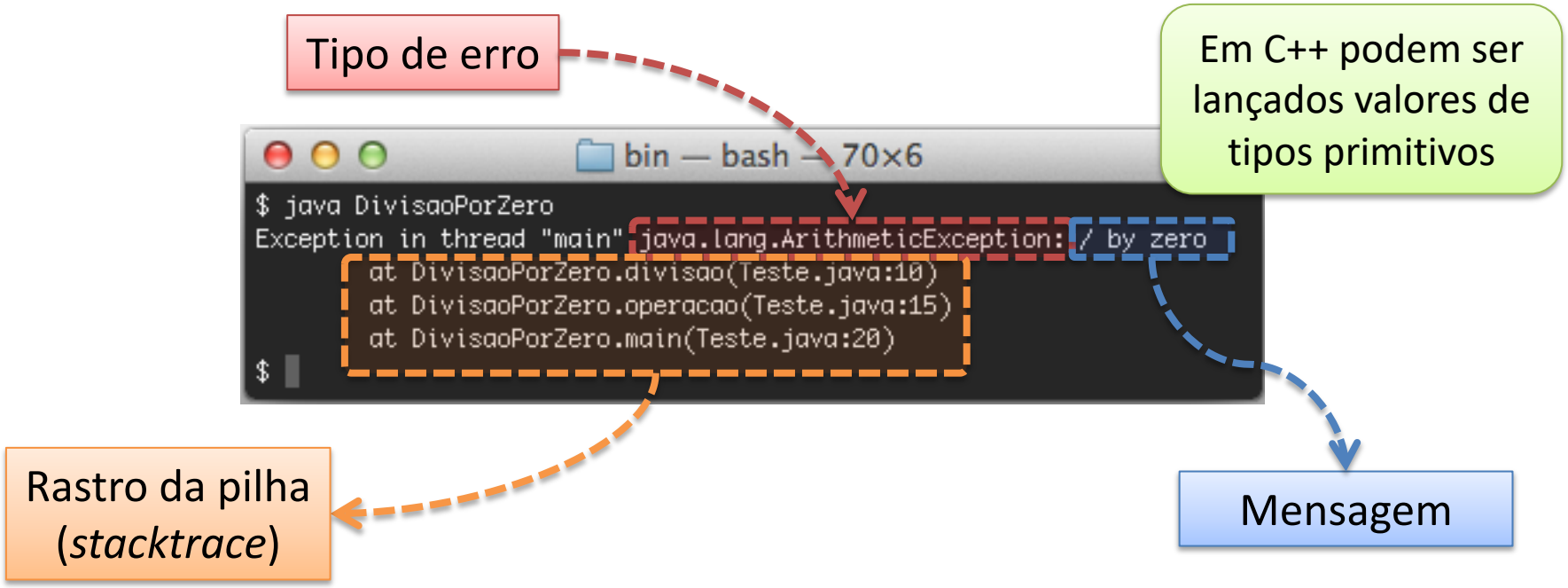
Sequência de chamadas

Rastro da pilha (stacktrace)

Condição da Exceção

- Uma exceção em Java é um **objeto** que descreve uma condição de exceção que ocorreu em algum fragmento de código

- Quando surge uma condição excepcional, um objeto do tipo **Exception** é criado e **lançado** naquele momento



TRATAMENTO DE EXCEÇÕES

Linguagens de Programação

Tratamento de Exceções

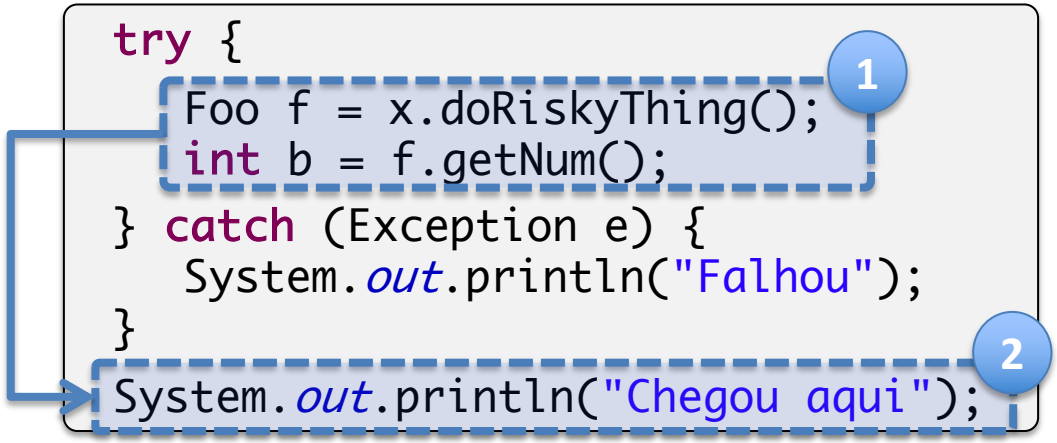
- Para tratar uma exceção, basta envolver o código que possui risco associado em um bloco **try** (*tentar*) com a exceção que se deseja capturar no bloco **catch** (*capturar*)

Como é o fluxo de execução **com sucesso**? E **com exceção**?

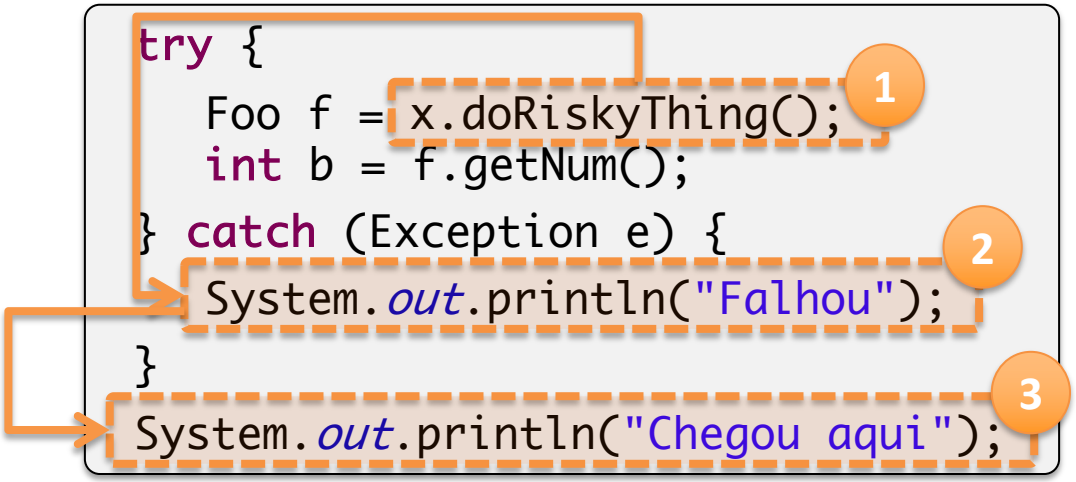
```
try {
   // fazer coisas arriscadas aqui
} catch (Exception ex) {
   // tentar recuperar da condição excepcional
}
```

Fluxo de Execução

- Em caso de sucesso



- Em caso de exceção



Exemplo

- Voltando o exemplo da divisão por zero

1) Qual exceção deve ser capturada?
2) Onde deve ser feito o tratamento?

```
 1. class DivisaoPorZero {
 2.     static final int MULTIPLICACAO = 3;
 3.     static final int DIVISAO = 4;
 4.
 5.     static int multiplicacao(int a, int b) {
 6.         return a * b;
 7.     }
 8.     static int divisao(int a, int b) {
 9.         return a / b;
10.     }
11.     static int operacao(int operacao, int a, int b) {
12.         switch (operacao) {
13.             case MULTIPLICACAO: return multiplicacao(a, b);
14.             case DIVISAO: return divisao(a, b);
15.             default: return 0;
16.         }
17.     }
18.     public static void main(String[] args) {
19.         int d = operacao(DIVISAO, 10, 0);
20.         System.out.println(d);
21.     }
22.}
```

Exemplo

- **Primeira opção:** no método `divisao`, quando é feita a operação

```
class DivisaoPorZero {

        ...

    static int divisao(int a, int b) {
        try {
            return a / b;
        } catch (ArithmeticException ex) {
            return 0;
        }
    }

        ...

}
```

Tipo da exceção que se deseja capturar

Exemplo

- **Segunda opção:** no método operacao, na chamada divisao

```
class DivisaoPorZero {

        ...

    static int operacao(int operacao, int a, int b) {
        switch (operacao) {
            case MULTIPLICACAO:
                return multiplicacao(a, b);
            case DIVISAO:
                try {
                    return divisao(a, b);
                } catch (ArithmeticException e) {
                    return 0;
                }
            default: return 0;
        }
    }

        ...

}
```

Exemplo

- **Terceira opção:** no método `main`, na chamada `operacao`

```
class DivisaoPorZero {

        ...

    public static void main(String[] args) {
        try {
            int d = operacao(DIVISAO, 10, 0);
            System.out.println(d);
        } catch (ArithmeticException e) {
            System.out.println("Divisão por zero");
        }
    }

        ...

}
```

Onde é melhor fazer o tratamento da exceção?

TIPOS DE EXCEÇÕES

Linguagens de Programação

Tipos de Exceções

- Em C++, exceções podem ser de
  - Tipos primitivos

```
try {
    // código monitorado
} catch (int i) {
    // tratamento de exceção
}
```

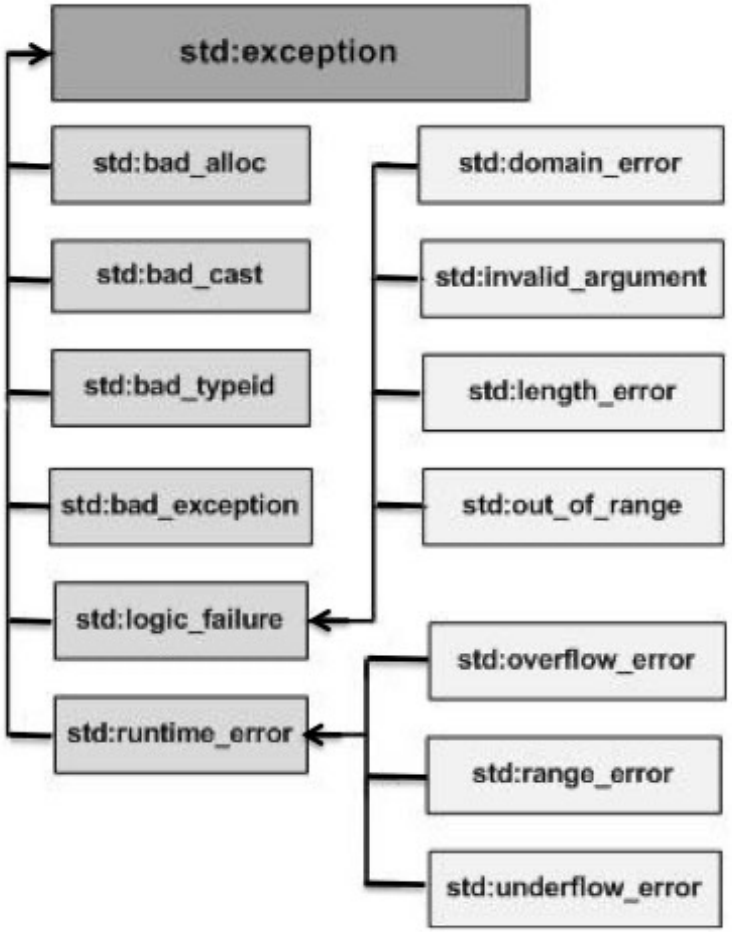
  - Objetos

```
try {
    // código monitorado
} catch (std::string& str) {
    // tratamento de exceção
}
```

Dica: é comum passar objetos por referência em blocos **catch** por questões de eficiência

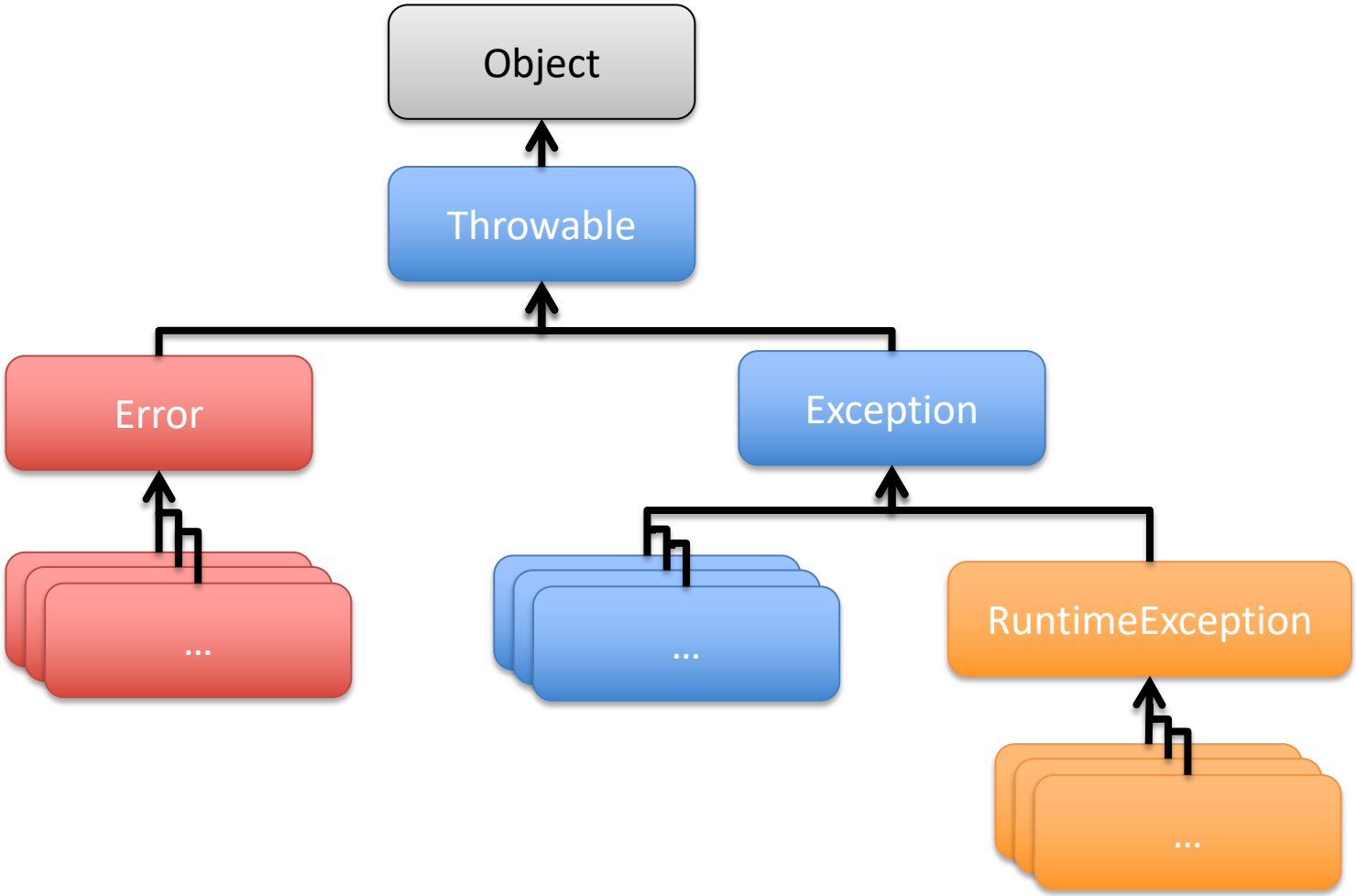
Exceções Padrões C++

- Lista de exceções padrões de C++



Tipos de Exceções

- Em Java, exceções são do tipo `Throwable` ou subclasses dela



Classificação de Exceções

- Em Java, exceções podem ser classificadas como
  - Exceções **não verificadas**
    - **Error** e subclasses: normalmente não devem ser tratadas, já que são erros não contornáveis

      Ex.: falta de memória virtual ou exaustão de pilha
    - **RuntimeException** e subclasses: o tratamento da exceção é opcional e usualmente pode ser contornado

      Ex.: índice fora do arranjo ou referência nula
  - Exceções **verificadas**
    - **Throwable** e subclasses, excluindo aquelas de **Error** e **RuntimeException**: devem ser obrigatoriamente tratadas ou propagadas

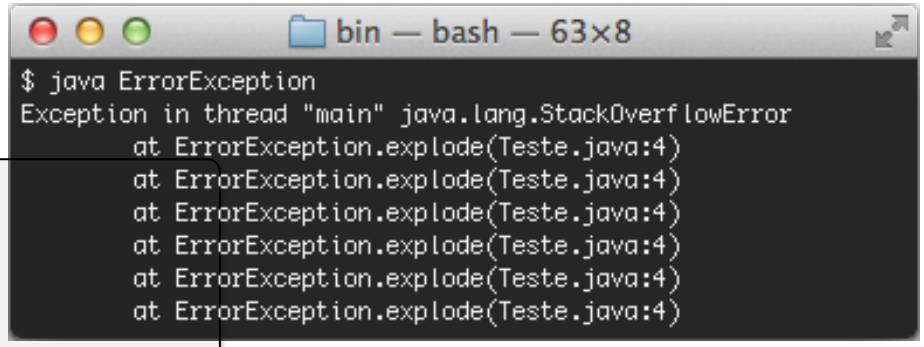
      Ex.: arquivo não encontrado ou fim de arquivo

Exceções Não Verificadas

- Exemplos de exceções **não verificadas** em Java

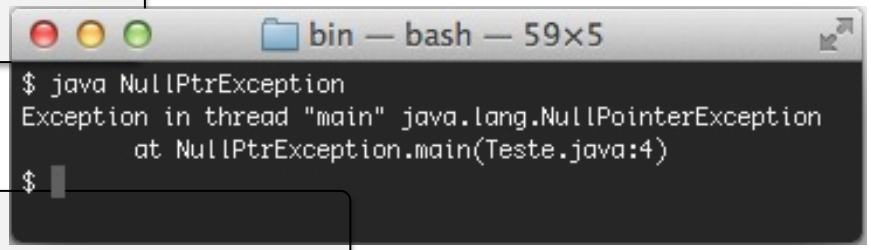
StackOverflowError

```
class ErrorException {
    static int explode(int n) {
        return explode(n + 1);
    }
    public static void main(String args[]) {
        System.out.println(explode(0));
    }
}
```



NullPointerException

```
class NullPtrException {
    public static void main(String args[]) {
        Object o = null;
        System.out.println(o.hashCode());
    }
}
```



Exceções Verificadas

- Exemplos de exceções **verificadas** em Java
  - Devem ser capturadas ou propagadas, caso contrário há um erro de **compilação**

FileNotFoundException

```
import java.io.FileInputStream;
import java.io.InputStream;

class ArquivoException {
    public InputStream abrirArquivo(String arquivo) {
        return new FileInputStream(arquivo);
    }
}
```

```
src — bash — 76×7
$ javac ArquivoException.java
ArquivoException.java:6: error: unreported exception FileNotFoundException;
must be caught or declared to be thrown
                return new FileInputStream(arquivo);
                       ^
1 error
$
```

TRATAR OU PROPAGAR?

Tratar ou Propagar?

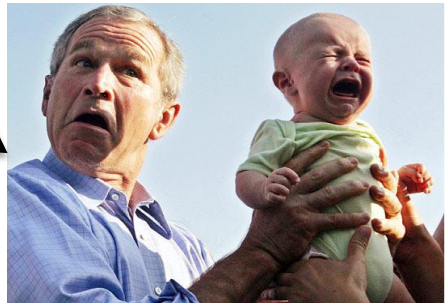
- Quando uma exceção é ativada, existem apenas duas ações possíveis: **tratar** ou **propagar**

- Exemplo: `PitiException`

Problema

Tratar?

Propagar?



Tratamento Único

- Um tratamento de exceção simples

```
class Exc2 {
    public static void main(String args[]) {
        int d, a;
        try {
            d = 0;
            a = 42 / d;
            System.out.println("Isso não será impresso");
        } catch (ArithmeticException e) {
            System.out.println("Divisão por zero.");
        }
        System.out.println("Após o catch.");
    }
}
```

Tratamento Múltiplo

- Pode-se tratar múltiplas exceções em um bloco de código monitorado, utilizando várias cláusulas catch

```
class MultiCatch {
    public static void main(String args[]) {
        try {
            int a = args.length;
            System.out.println("a = " + a);
            int b = 42 / a;
            int c[] = { 1 };
            c[42] = 99;
        } catch (ArithmeticException e) {
            System.out.println("Divisao por 0: " + e);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Indice fora da faixa: " + e);
        }
        System.out.println("Após bloco try/catch");
    }
}
```

As cláusulas catch são inspecionadas na ordem em que foram declaradas

Tratamento Múltiplo

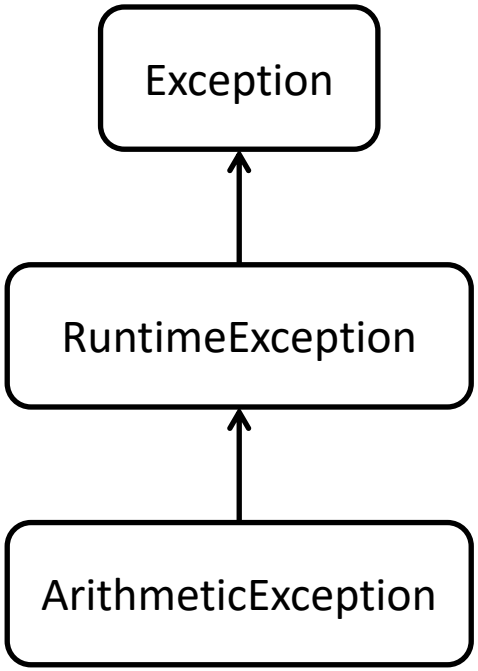
- Em Java, a partir da versão 1.7, pode-se tratar mais de uma exceção em um mesmo bloco catch através do operador |

```
class MultiCatch {
    public static void main(String args[]) {
        try {
            int a = args.length;
            System.out.println("a = " + a);
            int b = 42 / a;
            int c[] = { 1 };
            c[42] = 99;
        } catch (ArithmeticException e1 | ArrayIndexOutOfBoundsException e2) {
            System.out.println("Mesmo tratamento");
        }
        System.out.println("Após bloco try/catch");
    }
}
```

C++ não possui tal facilidade, como resolver?

Tratamento pela Hierarquia de Classe

- Pode-se utilizar uma superclasse para tratar vários tipos de exceções (classes de exceções) em um mesmo bloco **catch**




```
class SuperSubCatch {
    public static void main(String args[]) {
        try {
            int a = 0;
            int b = 42 / a;
        } catch (Exception e) {
            System.out.println(
                "Tratamento de exceção genérico");
        }
    }
}
```

Cuidado: uma exceção muito genérica pode acabar pegando mais erros do que aqueles intencionados

Tratamento pela Hierarquia de Classe

- **Cuidado:** subclasses devem ser declaradas sempre antes das superclasses no tratamento de exceções (erro de compilação)

```
class SuperSubCatch {
    public static void main(String args[]) {
        try {
            int a = 0;
            int b = 42 / a;
        } catch (Exception e) {
            System.out.println("Tratamento de exceção genérico.");
        } catch (ArithmeticException e) {
            System.out.println("Nunca será alcançado.");
        }
    }
}
```

Tratamento em Blocos try Aninhados

- Blocos try podem ser aninhados

```
class NestTry {
    public static void main(String args[]) {
        try {
            int a = args.length;
            int b = 42 / a;
            System.out.println("a = " + a);
            try {
                if (a == 1)
                    a = a / (a - a); // division by zero
                else if (a == 2) {
                    int c[] = { 1 };
                    c[42] = 99; // generate an out-of-bounds exception
                }
            } catch (ArrayIndexOutOfBoundsException e) {
                System.out.println("Índice fora do arranjo: " + e);
            }
        } catch (ArithmeticException e) {
            System.out.println("Divisão por 0: " + e);
        }
    }
}
```

As cláusulas catch mais internas são inspecionadas antes das mais externas

Tratamento em Blocos try Aninhados

- ... até de formas menos óbvias

```
class MethNestTry {
    static void nesttry(int a) {
        try {
            if (a == 1) a = a / (a - a);
            else if (a == 2) {
                int c[] = { 1 };
                c[42] = 99;
            }
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Índice fora do arranjo: " + e);
        }
    }
    public static void main(String args[]) {
        try {
            int a = args.length;
            int b = 42 / a;
            nesttry(a);
        } catch (ArithmeticException e) {
            System.out.println("Divisão por 0: " + e);
        }
    }
}
```

Tratamento de Qualquer Exceção

- Em Java, pode-se tratar **todas** as exceções capturando exceções do tipo `Throwable`, mas não deve-se capturar exceções do tipo `Error`, portanto use `Exception`

```
try {
  // ...
} catch (Exception e) {
  // ...
}
```

- Em C++, pode-se capturar todas as exceções usando ... (três pontos)

```
try {
  // ...
} catch (...) {
  // ...
}
```

Propagação

- Propagar é delegar para o método chamador a responsabilidade de tratar a exceção (que por sua vez pode tratar ou propagar)

- Em C++, se uma exceção não é tratada, ela é automaticamente propagada

- Em Java depende do tipo das exceções
  - Se **não verificadas**: são automaticamente propagadas
  - Se **verificadas**: devem ser explicitamente anotadas no método através da palavra reservada `throws`

```
tipo nome-metodo(lista-parâmetros) throws lista-exceções {
   // Corpo do método
}
```

Propagação de Exceções Verificadas

- Exemplo de propagação de exceção verificada (FileNotFoundException)

O que acontece se não tratar no método main?

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.InputStream;
class ArquivoException {
    static InputStream abrirArquivo(String arquivo)
                throws FileNotFoundException {
        return new FileInputStream(arquivo);
    }
    public static void main(String[] args) {
        try {
            InputStream i = abrirArquivo("arquivo.txt");
            // ...
        } catch (FileNotFoundException e) {
            System.out.println("Arquivo não encontrado");
        }
    }
}
```

LANÇAMENTO DE EXCEÇÕES

Linguagens de Programação

Lançamento de Exceções

- Até agora foram capturadas apenas exceções lançadas pelo próprio ambiente de execução, mas é possível lançar suas próprias exceções

- Tanto C++ quanto Java utilizam a palavra reservada `throw` para lançar uma exceção
  - C++

```
throw valor;
```

    - tal que `valor` pode ser um tipo primitivo, objetos alocados na pilha ou na heap
  - Java

```
throw instânciaThrowable;
```

    - tal que `instânciaThrowable` é um objeto do tipo `Throwable` ou subclasse (tipos primitivos e outros objetos não são permitidos)

throw

- Assim como no comando **`return`** o fluxo de execução é interrompido no comando **`throw`**, ou seja, nenhum comando subsequente será executado

- O fluxo então segue como se fosse uma exceção lançada pelo ambiente
  - o bloco `try` mais próximo será inspecionado, se não o próximo bloco `try` até que algum trate a exceção ou chegue no tratador padrão

Exemplo

- Reescrevendo o método sacar de Conta para lançar uma exceção

```
boolean sacar(double valor) {
    if (this.saldo < valor) {
        return false;
    } else {
        this.saldo -= valor;
        return true;
    }
}
```

Repare que não é mais **boolean**

```
void sacar(double valor) {
    if (this.saldo < valor)
        throw new RuntimeException();

    this.saldo -= valor;
}
```

Qual a desvantagem de usar exceção do tipo RuntimeException?

Exemplo

- Reescrevendo o lançador usando uma exceção mais específica, como **IllegalArgumentException**

```
void sacar(double valor) {
   if (this.saldo < valor)
       throw new IllegalArgumentException();

   this.saldo -= valor;
}
```

- ... e tratando ...

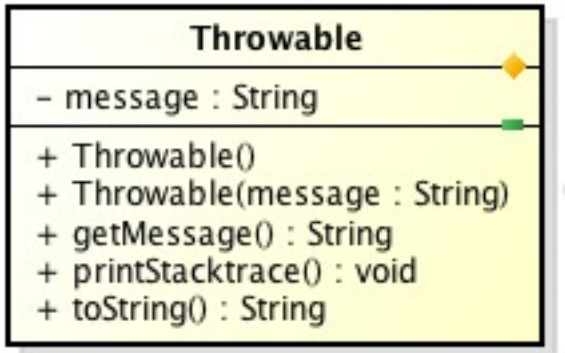
```
Conta cc = new ContaCorrente();
cc.depositar(100);

try {
   cc.sacar(100);
} catch (IllegalArgumentException e) {
   System.out.println("Saldo Insuficiente");
}
```

Como padronizar a mensagem de erro?

Mensagem de Erro

- Em Java, a classe `Throwable` mantém um atributo `message` que pode ser utilizado para armazenar uma mensagem de erro



Possui mais métodos!

```
void sacar(double valor) {
   if (this.saldo < valor)
       throw new IllegalArgumentException("Saldo Insuficiente");

   this.saldo -= valor;
}
```

```
Conta cc = new ContaCorrente();
cc.depositar(100);

try {
   cc.saca(100);
} catch (IllegalArgumentException e) {
   System.out.println(e.getMessage());
}
```

Em C++, pode-se criar suas próprias classes base

EXCEÇÕES PRÓPRIAS

Linguagens de Programação

Exceções Próprias

- É recomendado que você aproveite as exceções já disponíveis no seu ambiente sempre que possível

- Algumas exceções de Java comumente reaproveitadas
  - `IllegalArgumentException`: quando um argumento de um método é inválido, como um número negativo
  - `IllegalStateException`: quando se pede uma ação em uma classe mas ela se encontra em um estado tal que não é possível realizá-la
  - `IllegalAccessException`: quando não se tem permissão para acessar determinado recurso

Exceções Próprias

- Contudo, é muito comum criar exceções próprias para melhor retratar o domínio de sua aplicação
  - Com isso, é possível adicionar mais informações de interesse que podem ser carregadas nas exceções

- Para criar sua própria exceção, deve-se
  - Em C++
    - Usar tipos primitivos ou quaisquer objetos
  - Em Java
    - Herdar de **`Exception`** para criar **exceções verificadas**
    - Herdar de **`RuntimeException`** para criar exceções **não verificadas**

Por que não herdar de Throwable?

Exceções Não Verificadas

- Declaração de exceção **não verificadas**

```
class SaldoInsuficienteException extends RuntimeException {

    private double valor;
    private double saldo;

    SaldoInsuficienteException(double valor, double saldo) {
        super("Saldo de " + saldo + " é insuficiente para " + valor);

        this.valor = valor;
        this.saldo = saldo;
    }
    double getValor() {
        return valor;
    }
    double getSaldo() {
        return saldo;
    }
}
```

Chamada construtor superclasse com mensagem de erro

Recomendação: terminar nome com **Exception**

Exemplo

- Exemplo de uso da nova exceção **não verificada**

```
void sacar(double valor) {
   if (this.saldo < valor)
       throw new SaldoInsuficienteException(valor, this.saldo);

   this.saldo -= valor;
}
```

```
Conta cc = new ContaCorrente();
cc.deposita(10);

try {
   cc.sacar(100);
} catch (SaldoInsuficienteException e) {
   System.out.println(e.getMessage());
}
```

Exceções Verificadas

- Tornando a declaração da exceção anterior **verificada** passando somente mensagem

```
class SaldoInsuficienteException extends Exception {

    SaldoInsuficienteException() {
    }

    SaldoInsuficienteException(String msg) {
        super(msg);
    }
}
```

Exemplo

- O método que lança a exceção deve ser agora anotado com **throws**

```
void sacar(double valor) throws SaldoInsuficienteException  {
   if (this.saldo < valor)
       throw new SaldoInsuficienteException("Saldo insuficiente");

   this.saldo -= valor;
}
```

Qual tipo de exceção utilizar?
Verificada ou **não verificada**?

Verificadas vs. Não Verificadas

> Não lance uma exceção **RuntimeException** ou crie uma subclasse dela simplesmente porque você não quer ser incomodado ao especificar as exceções que os métodos podem lançar.

Tutorial de Java♣

> Eu vejo dois grandes problemas com exceções verificadas: escalabilidade e versionamento.

Anders Hejlsberg, criador do C#♠

C# não suporta exceções verificadas

♣: http://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html
♠: http://www.artima.com/intv/handcuffs.html

Verificadas vs. Não Verificadas

- Em Java tem-se as duas opções de exceções: quando usar cada uma delas e em quais casos?

> “Se é esperado que um cliente possa razoavelmente recuperar de uma exceção, faça uma **exceção verificada**. Se um cliente não pode fazer nada para recuperar da situação, faça uma exceção **não verificada.**”
>
> Tutorial de Java♣

♣ : http://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html

BLOCO FINALLY

Bloco `finally`

- Quando exceções são lançadas, o código muda de direção de forma abrupta tomando um caminho não linear de execução; o método pode até retornar prematuramente

- Isso pode ser um problema...
  - Considere o caso onde um método abre um arquivo e o fecha no final: é desejável que, mesmo que haja uma exceção, o código que fecha o arquivo seja executado

O bloco **`finally`** resolve esse problema

C++ não possui bloco **`finally`**, por quê?

Declaração com `finally`

- **`finally`** cria um bloco de código que será executado após um **`try/catch`** ter completado e antes da próxima instrução subsequente
- Será executado independentemente de ter lançado exceção ou não
  - Até com **`return`**, mas não com **`System.exit()`**
  - Se uma exceção for lançada e não houver tratador no **`try/catch`**, ainda sim o bloco de código contido na cláusula **`finally`** será executado

```
try {
   // bloco try
} catch (IOException ex) {
   // bloco catch 1
} catch (SQLException sqlex) {
   // bloco catch 2
} finally {
   // bloco que será sempre executado, independente
   // se houve ou não exception e se ela foi tratada ou não
}
```

`finally` sem `catch`

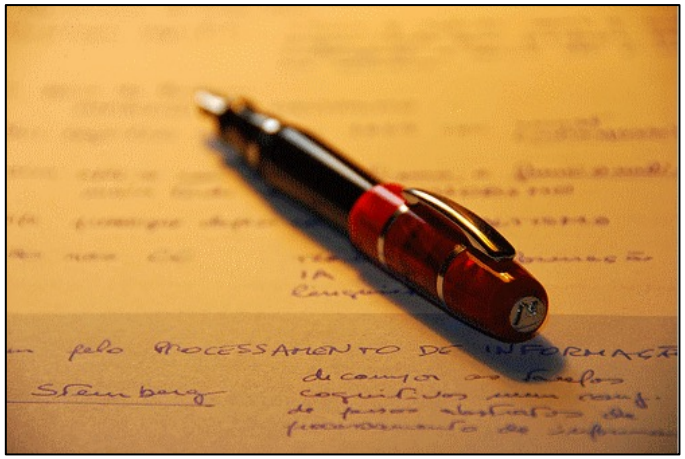
- **`finally`** é opcional, mas precisa de um bloco **`try`**; porém não necessariamente de um bloco **`catch`**

```
try {
   // bloco try
} finally {
   // bloco finally
}
```

Exemplo

```
class FinallyDemo {
    static void procA() {
        try {
            System.out.println("inside procA");
            throw new RuntimeException("demo");
        } finally {
            System.out.println("procA's finally");
        }
    }
    static void procB() {
        try {
            System.out.println("inside procB");
            return;
        } finally {
            System.out.println("procB's finally");
        }
    }
    static void procC() {
        try {
            System.out.println("inside procC");
        } finally {
            System.out.println("procC's finally");
        }
    }
}
```

RESUMO



Exceções

- Tratar erros é uma tarefa que demanda bastante codificação, o que acaba tornando o código difícil de entender

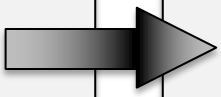
```
function register()
{
    if (!empty($_POST)) {
        $msg = '';
        if ($_POST['user_name']) {
            if ($_POST['user_password_new']) {
                if ($_POST['user_password_new'] === $_POST['user_password_repeat']) {
                    if (strlen($_POST['user_password_new']) > 5) {
                        if (strlen($_POST['user_name']) < 65 && strlen($_POST['user_name']) > 1) {
                            if (preg_match('/^[a-z\d]{2,64}$/i', $_POST['user_name'])) {
                                $user = read_user($_POST['user_name']);
                                if (!isset($user['user_name'])) {
                                    if ($_POST['user_email']) {
                                        if (strlen($_POST['user_email']) < 65) {
                                            if (filter_var($_POST['user_email'], FILTER_VALIDATE_EMAIL)) {
                                                create_user();
                                                $_SESSION['msg'] = 'You are now registered so please login';
                                                header('Location: ' . $_SERVER['PHP_SELF']);
                                                exit();
                                            } else $msg = 'You must provide a valid email address';
                                        } else $msg = 'Email must be less than 64 characters';
                                    } else $msg = 'Email cannot be empty';
                                } else $msg = 'Username already exists';
                            } else $msg = 'Username must be only a-z, A-Z, 0-9';
                        } else $msg = 'Username must be between 2 and 64 characters';
                    } else $msg = 'Password must be at least 6 characters';
                } else $msg = 'Passwords do not match';
            } else $msg = 'Empty Password';
        } else $msg = 'Empty Username';
        $_SESSION['msg'] = $msg;
    }
    return register_form();
}
```

Exemplo de cadastro de usuário em PHP

Vantagem 1: Separar Código de Erro de Código Regular

- O tratamento de exceções permite aos programadores remover código de tratamento de erro da 'linha principal'

```
errorCodeType readFile {
  initialize errorCode = 0;
  open the file;
  if (theFileIsOpen) {
    determine the length of the file;
    if (gotTheFileLength) {
      allocate that much memory;
      if (gotEnoughMemory) {
        read the file into memory;
        if (readFailed) errorCode = -1;
      } else errorCode = -2;
    } else errorCode = -3;
      close the file;
      if (theFileDidntClose && errorCode == 0)
        errorCode = -4;
      else
        errorCode = errorCode and -4;
    } else errorCode = -5;
    return errorCode;
}
```

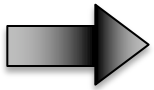


```
readFile {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    } catch (fileOpenFailed) {
        doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
    } catch (memoryAllocationFailed) {
        doSomething;
    } catch (readFailed) {
        doSomething;
    } catch (fileCloseFailed) {
        doSomething;
    }
}
```

Vantagem 2: Propagar Erros na Pilha de Chamadas

- Habilidade de propagar relatório de erro na pilha de chamadas

```
method1 {
    errorCodeType error;
    error = call method2;
    if (error)
        doErrorProcessing;
    else
        proceed;
}
errorCodeType method2 {
    errorCodeType error;
    error = call method3;
    if (error)
        return error;
    else
        proceed;
}
errorCodeType method3 {
    errorCodeType error;
    error = call readFile;
    if (error)
        return error;
    else
        proceed;
}
```



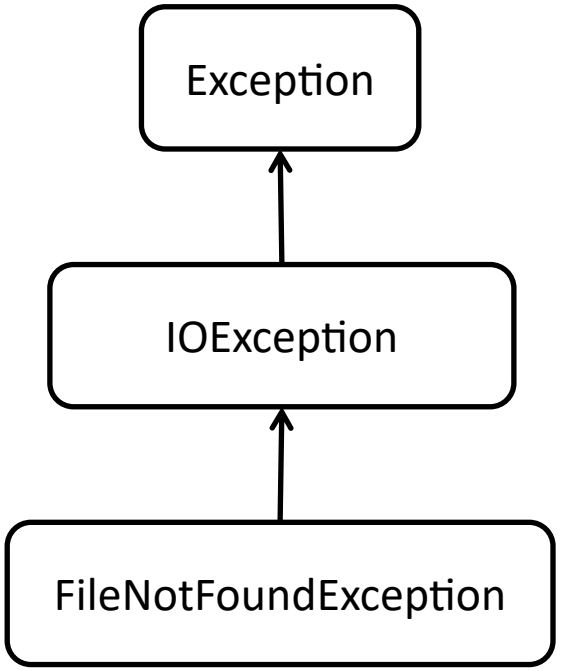
```
method1 {
    try {
        call method2;
    } catch (exception e) {
        doErrorProcessing;
    }
}

method2 throws exception {
    call method3;
}

method3 throws exception {
    call readFile;
}
```

Vantagem 3: Agrupar e Diferenciar Tipos de Erros

- Como todas as exceções lançadas são objetos, o agrupamento ou categorização vem naturalmente com hierarquia de classes




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

ou

```
} catch (IOException e) {
```

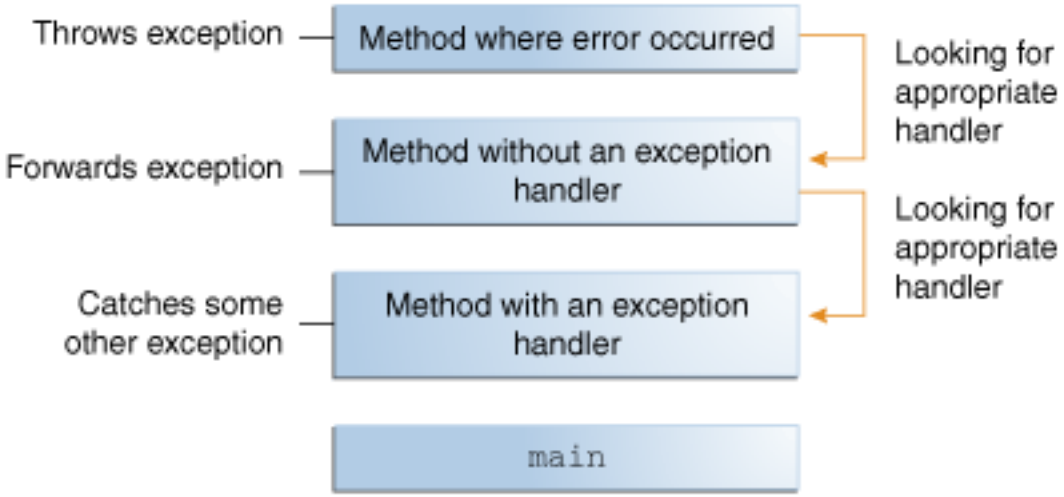
ou

```
} catch (Exception e) {
```

Desvantagens

- Não existe suporte nativo nos processadores atuais para tratamento de exceção de alto nível

- O compilador deve manter uma estrutura de dados e adicionar código para gerenciar o lançamento e tratamento de exceções em todos os métodos envolvidos com exceções
  - O código é naturalmente mais lento



Evitar Ativar Exceções

- Não se pode controlar quando algumas exceções irão ocorrer
  - Ex.: `SocketException`, `SQLException`, ...
- Contudo, para algumas exceções (principalmente aquelas do tipo RuntimeException), é possível evitar que elas sejam ativadas; ou seja, não precisam ser tratadas
  - Ex.: `ArithmeticException`, `NullPointerException`, ...

Exemplos

- Exemplo: `ArithmeticException`

```
try {
    c = a / b;
} catch (ArithmeticException e) {
    // ...
}
```

```
if (b != 0) {
    c = a / b;
}
```

- Exemplo: `NullPointerException`

```
for (Conta c : contas) {
    if (c != null) {
        lista.add(c.getNumero());
    }
}
```

```
for (Conta c : contas) {
    try {
        lista.add(c.getNumero());
    } catch (NullPointerException e) {
        // ...
    }
}
```

- Exemplo: `ArrayIndexOutOfBoundsException`

```
try {
    v[i] = 50;
} catch (ArrayIndexOutOfBoundsException e) {
    // ...
}
```

```
if (i < v.length) {
    v[i] = 50;
}
```

Resumo

- Resumo das palavras-chave envolvendo exceções
  - **`try`:** bloco de código capaz de identificar onde podem ocorrer exceções
  - **`catch`:** bloco de código que pode lidar com um tipo de exceção particular (*exception handler*)
  - **`finally`:** bloco de código onde é garantido que será executado; lugar certo para fechar arquivos, recuperar recursos, entre outros
  - **`throws`:** utilizado na assinatura da função para indicar quais exceções o método pode lançar
  - **`throw`:** ativar uma exceção

ISSO É TUDO, PESSOAL!