11 Erros, exceções e afirmações

Helder da Rocha
www.argonavis.com.br

Controle de erros com Exceções

- Exceções são
  - Erros de tempo de execução
  - Objetos criados a partir de classes especiais que são "lançados" quando ocorrem condições excepcionais
- Métodos podem capturar ou deixar passar exceções que ocorrerem em seu corpo
  - É obrigatório, para a maior parte das exceções, que o método declare quaisquer exceções que ele não capturar
- Mecanismo try-catch é usado para tentar capturar exceções enquanto elas passam por métodos

Três tipos de erros de tempo de execução

- 1. Erros de lógica de programação
  - ex: limites do vetor ultrapassados, divisão por zero
  - devem ser corrigidos pelo programador
- 2. Erros devido a condições do ambiente de execução
  - ex: arquivo não encontrado, rede fora do ar, etc.
  - fogem do controle do programador mas podem ser contornados em tempo de execução
- 3. Erros graves onde não adianta tentar recuperação
  - ex: falta de memória, erro interno do JVM
  - fogem do controle do programador e não podem ser contornados

Como causar uma exceção?

- *Uma exceção é um tipo de objeto que sinaliza que uma condição excepcional ocorreu*
  - *A identificação (nome da classe) é sua parte mais importante*
- *Precisa ser criada com new e depois lançada com* ***throw***
  - ```
    IllegalArgumentException e = new
            IllegalArgumentException("Erro!");
    throw e;  // exceção foi lançada!
    ```
- *A referência é desnecessária. A sintaxe abaixo é mais usual:*
  - ```
    throw new IllegalArgumentException("Erro!");
    ```

Exceções e métodos

- *Uma declaração throws é obrigatória em métodos e construtores que deixam de capturar uma ou mais exceções que ocorrem em seu interior*

```
public void m() throws Excecao1, Excecao2 {...}
public Circulo() throws ExcecaoDeLimite {...}
```

- ***throws*** *declara que o método pode provocar exceções do tipo declarado (ou de qualquer subtipo)*
  - *A declaração abaixo declara que o método pode provocar qualquer exceção (não faça isto)*

```
public void m() throws Exception {...}
```

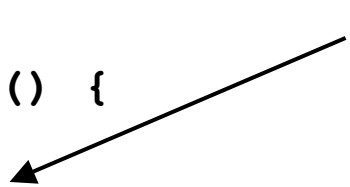
- *Métodos sobrepostos não podem provocar mais exceções que os métodos originais*

O que acontece?

- Uma exceção lançada interrompe o fluxo normal do programa
  - O fluxo do programa segue a exceção
  - Se o método onde ela ocorrer não a capturar, ela será propagada para o método que chamar esse método e assim por diante
  - Se ninguém capturar a exceção, ela irá causar o término da aplicação
  - Se em algum lugar ela for capturada, o controle pode ser recuperado

Captura e declaração de exceções

```
public class RelatorioFinanceiro {
    public void metodoMau() throws ExcecaoContabil {
        if (!dadosCorretos) {
            throw new ExcecaoContabil("Dados Incorretos");
        }
    }
    public void metodoBom() {
        try {
            ... instruções ...
            metodoMau();
            ... instruções ...
        } catch (ExcecaoContabil ex) {
            System.out.println("Erro: " + ex.getMessage());
        }
        ... instruções ...
    }
}
```



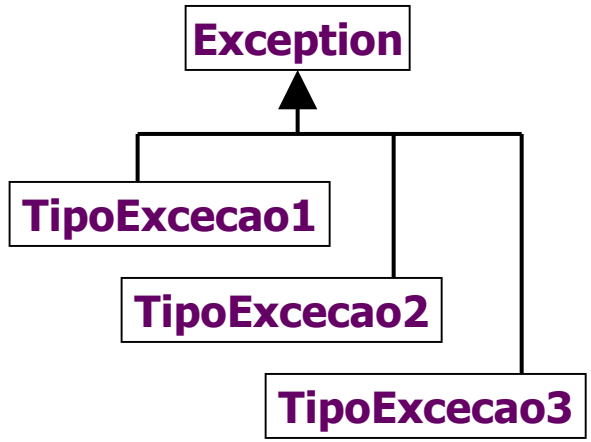
instruções que sempre serão executadas

instruções serão executadas se exceção não ocorrer

instruções serão executadas se exceção não ocorrer ou se ocorrer e for capturada

try e catch

- *O bloco try "tenta" executar um bloco de código que pode causar exceção*
- *Deve ser seguido por*
  - *Um ou mais blocos catch(TipoDeExcecao ref)*
  - *E/ou um bloco finally*
- *Blocos catch recebem tipo de exceção como argumento*
  - *Se ocorrer uma exceção no try, ela irá descer pelos catch até encontrar um que declare capturar exceções de uma classe ou superclasse da exceção*
  - *Apenas um dos blocos catch é executado*




```
try {
    ... instruções ...
} catch (TipoExcecao1 ex) {
    ... faz alguma coisa ...
} catch (TipoExcecao2 ex) {
    ... faz alguma coisa ...
} catch (Exception ex) {
    ... faz alguma coisa ...
}

... continuação ...
```

finally

- O bloco try não pode aparecer sozinho
  - deve ser seguido por pelo menos um catch ou por um finally
- O bloco finally contém instruções que devem se executadas independentemente da ocorrência ou não de exceções

```
try {
  // instruções: executa até linha onde ocorrer exceção
} catch (TipoExcecao1 ex) {
   // executa somente se ocorrer TipoExcecao1

} catch (TipoExcecao2 ex) {
   // executa somente se ocorrer TipoExcecao2
} finally {
   // executa sempre ...
}

// executa se exceção for capturada ou se não ocorrer
```

Como criar uma exceção

- *A não ser que você esteja construindo uma API de baixo-nível ou uma ferramenta de desenvolvimento, você so usará exceções do tipo (2) (veja página 3)*
- *Para criar uma classe que represente sua exceção, basta estender java.lang.Exception:*

```
class NovaExcecao extends Exception {}
```

- *Não precisa de mais nada. O mais importante é herdar de Exception e fornecer uma identificação diferente*
  - *Bloco catch usa nome da classe para identificar exceções.*

Como criar uma exceção (2)

- *Você também pode acrescentar métodos, campos de dados e construtores como em qualquer classe.*
- *É comum é criar a classe com dois construtores*

```
class NovaExcecao extends Exception {
    public NovaExcecao () {}
    public NovaExcecao (String mensagem) {
        super(mensagem);
    }
}
```

- *Esta implementação permite passar mensagem que será lida através de toString() e getMessage()*

Principais métodos

- Construtores de Exception
  - Exception (String message)
  - Exception ()
- Métodos de Exception
  - String getMessage()
    - Retorna mensagem passada pelo construtor
  - String toString()
    - Retorna nome da exceção e mensagem
  - void printStackTrace()
    - imprime detalhes sobre exceção

Como pegar qualquer exceção

- Se, entre os blocos catch, houver exceções da mesma hierarquia de classes, as classes mais específicas (que estão mais abaixo na hierarquia) devem aparecer primeiro
  - Se uma classe genérica (ex: Exception) aparecer antes de uma mais específica, uma exceção do tipo da específica jamais será capturado
  - O compilador detecta a situação acima e não compila o código
- Para pegar qualquer exceção, faça um catch que pegue a classe Exception

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

Relançar uma exceção

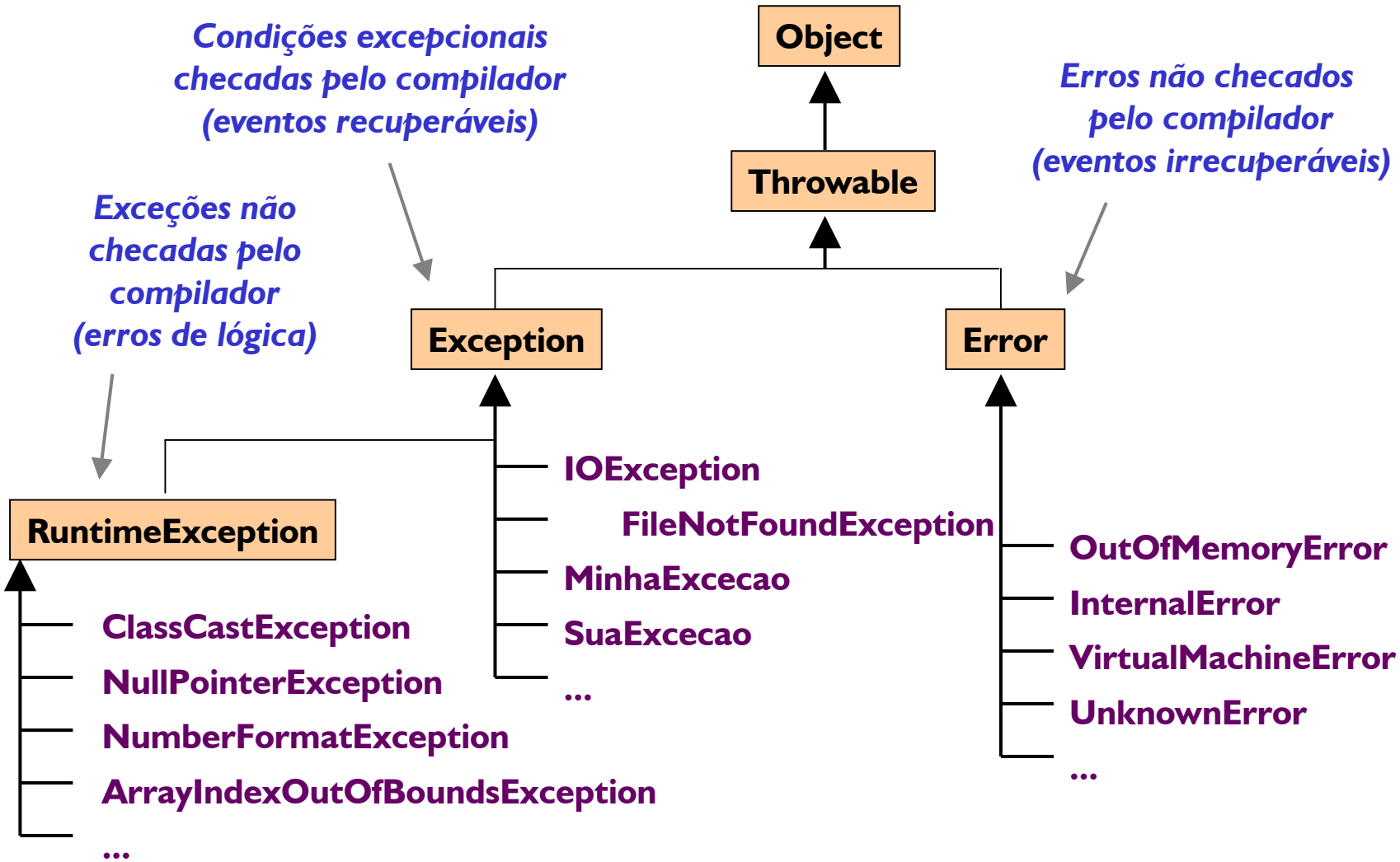
- *Às vezes, após a captura de uma exceção, é desejável relançá-la para que outros métodos lidem com ela*
  - *Isto pode ser feito da seguinte forma*

```
public void metodo() throws ExcecaoSimples {
   try {
     // instruções
   } catch (ExcecaoSimples ex) {
     // faz alguma coisa para lidar com a exceção
     throw ex;  // relança exceção
   }
}
```

Classes base da API

- *RuntimeException e Error*
  - *Exceções não verificadas em tempo de compilação*
  - *Subclasses de Error não devem ser capturadas (são situações graves em que a recuperação é impossível ou indesejável)*
  - *Subclasses de RuntimeException representam erros de lógica de programação que devem ser corrigidos (podem, mas não devem ser capturadas: erros devem ser corrigidos)*
- *Exception*
  - *Exceções verificadas em tempo de compilação (exceção à regra são as subclasses de RuntimeException)*
  - *Compilador exige que sejam ou capturadas ou declaradas pelo método que potencialmente as provoca*

Hierarquia


Condições excepcionais
checadas pelo compilador
(eventos recuperáveis)
Exceções não
checadas pelo
compilador
(erros de lógica)
Erros não checados
pelo compilador
(eventos irrecuperáveis)
Object
Throwable
Exception
Error
RuntimeException
IOException
FileNotFoundException
MinhaExcecao
SuaExcecao
...
ClassCastException
NullPointerException
NumberFormatException
ArrayIndexOutOfBoundsException
...
OutOfMemoryError
InternalError
VirtualMachineError
UnknownError
...

Movimento de exceções

- *Exceções são provocadas dentro de métodos, blocos de inicialização ou construtores*
  - *métodos e construtores podem capturá-las ou deixá-las acontecer (neste caso devem declarar quais exceções provocam)*
  - *blocos de inicialização static devem capturar todas as exceções*
- *Repasse de exceções*
  - *métodos que se declaram possíveis causadores de uma exceção, repassam o objeto para o método que o chamou que, por sua vez, pode capturá-la ou repassá-la*
  - *a declaração é feita com a palavra throws (não confunda com throw)*
- *Captura de exceções*
  - *trechos de código que podem provocar exceção (que chamam métodos que causam exceções ou que possuem instruções throw) devem ser declarados dentro de um bloco try*
  - *Um ou mais blocos catch seguem o bloco try e capturam exceções selecionadas pelo nome da classe*

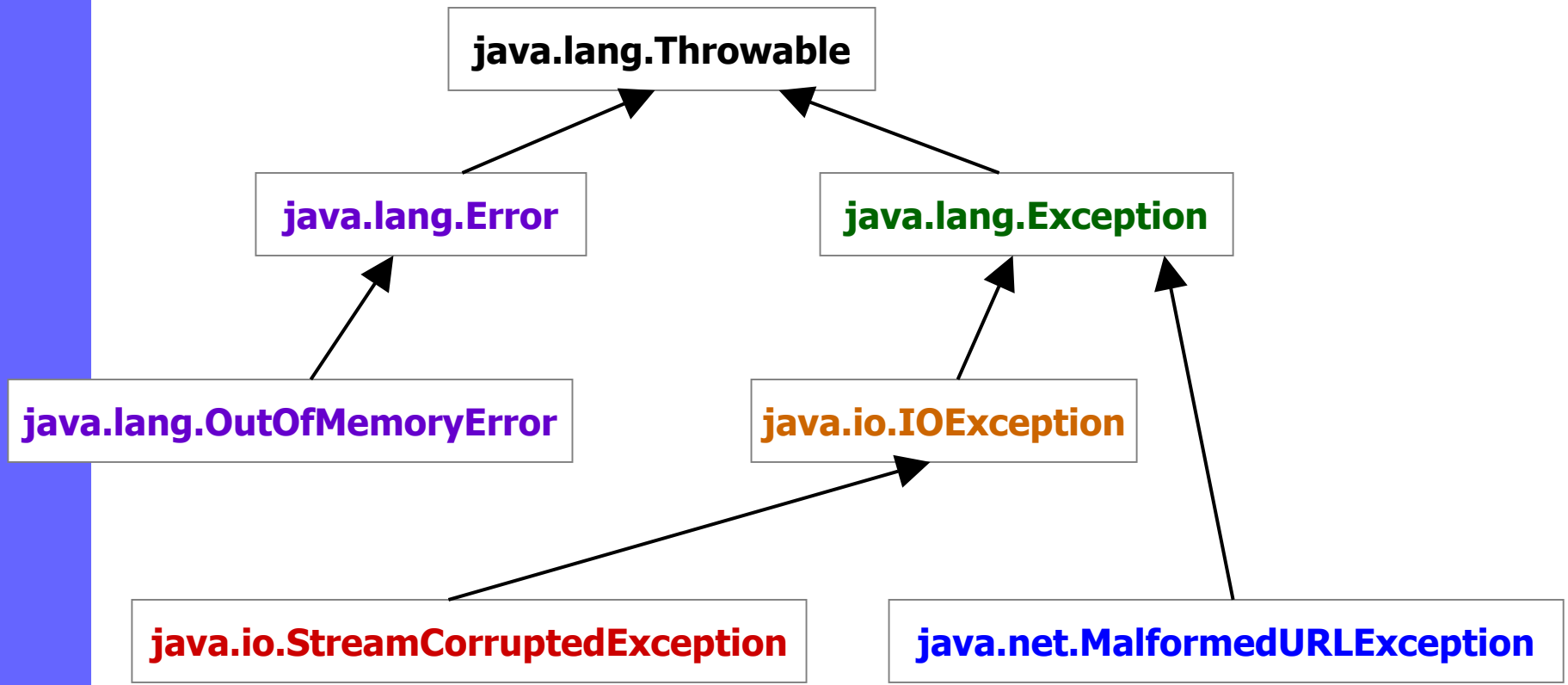
Testes (Enunciado parte 1)

- *Considere o seguinte código*

```
1.  try {
2.    URL u = new URL(s); // s is a previously defined String
3.    Object o = in.readObject(); // in is valid ObjectInputStream
4.    System.out.println("Success");
5.  }
6.  catch (MalformedURLException e) {
7.    System.out.println("Bad URL");
8.  }
9.  catch (IOException e) {
10.   System.out.println("Bad file contents");
11. }
12. catch (Exception e) {
13.   System.out.println("General exception");
14. }
15. finally {
16.   System.out.println("doing finally part");
17. }
18. System.out.println("Carrying on");
```

Testes (Enunciado parte 2)

- *Considere a seguinte hierarquia*



Teste 1

- *1. Que linhas são impressas se os métodos das linhas 2 e 3 completarem com sucesso sem provocar exceções?*
  - *A. Success*
  - *B. Bad URL*
  - *C. Bad File Contents*
  - *D. General Exception*
  - *E. Doing finally part*
  - *F. Carrying on*

Teste 2

- *2. Que linhas são impressas se o método da linha 3 provocar um OutOfMemoryError?*
  - *A. Success*
  - *B. Bad URL*
  - *C. Bad File Contents*
  - *D. General Exception*
  - *E. Doing finally part*
  - *F. Carrying on*

Teste 3

- *3. Que linhas são impressas se o método da linha 2 provocar uma MalformedURLException?*
  - *A. Success*
  - *B. Bad URL*
  - *C. Bad File Contents*
  - *D. General Exception*
  - *E. Doing finally part*
  - *F. Carrying on*

Teste 4

- *4. Que linhas são impressas se o método da linha 3 provocar um StreamCorruptedException?*
  - *A. Success*
  - *B. Bad URL*
  - *C. Bad File Contents*
  - *D. General Exception*
  - *E. Doing finally part*
  - *F. Carrying on*

Exercícios

- 1. Crie a seguinte hierarquia de exceções
  - ExcecaoDePublicacao
  - AgenteInexistente
  - AgenteDuplicado
  - PublicacaoInexistente extends ExcecaoDePublicacao
  - PublicacaoDuplicada extends ExcecaoDePublicacao
- 2. Quais métodos das classes da aplicação Biblioteca (cap. 9) devem definir essas exceções?
  - Declare (throws) as excecoes nos métodos escolhidos
  - Inclua o teste e lançamento de exceções no seu código (RepositorioDadosMemoria)

Afirmações (assertions)

- São expressões booleanas que o programador define para afirmar uma condição que ele acredita ser verdade
  - Afirmações são usadas para validar código (ter a certeza que um vetor tem determinado tamanho, ter a certeza que o programa não passou por determinado lugar)
  - Melhoram a qualidade do código: tipo de teste
  - Devem ser usadas durante o desenvolvimento e desligadas na produção (afeta a performance)
  - Não devem ser usadas como parte da lógica do código
- Afirmações são um recurso novo do JSDK1.4.0
  - Nova palavra-chave: **assert**
  - É preciso compilar usando a opção -source 1.4:

```
> javac -source 1.4 Classe.java
```

  - Para executar, é preciso habilitar afirmações (enable assertions):

```
> java -ea Classe
```

Afirmações: sintaxe

- Afirmações testam uma condição. Se a condição for falsa, um AssertionError é lançado
- Sintaxe:
  - `assert expressão;`
  - `assert expressãoUm : expressãoDois;`
- Se primeira expressão for true, a segunda não é avaliada
  - Sendo falsa, um AssertionError é lançado e o valor da segunda expressão é passado no seu construtor.
- Exemplo

Em vez de usar comentário...

```
if (i%3 == 0) {
   ...
} else if (i%3 == 1) {
   ...
} else { // (i%3 == 2)
   ...
}
```

... use uma afirmação!.

```
if (i%3 == 0) {
   ...
} else if (i%3 == 1) {
   ...
} else {
   assert i%3 == 2;
   ...
}
```

Afirmações: exemplo

- *Trecho de código que afirma que controle nunca passará pelo default:*

```
switch(estacao) {
  case Estacao.PRIMAVERA:
    ...
    break;
  case Estacao.VERAO:
    ...
    break;
  case Estacao.OUTONO:
    ...
    break;
  case Estacao.INVERNO:
    ...
    break;
  default:
    assert false: "Controle nunca chegará aqui!";
}
```

Exercícios

- *1. Implemente um procedimento que tenha um switch para quatro estações (primavera: 1, verão: 2, outono: 3, inverno: 4)*
  - *a) Em cada case, coloque um System.out.println() que imprima a estação escolhida. Escreva um main que selecione algumas estações.*
  - *b) Coloque um assert false no default para afirmar que o código jamais deveria passar por lá*
  - *c) Invente uma quinta estação e veja se o AssertError acontece*
- *Não se esqueça de compilar com -source 1.4 e executar com -ea*
  - *(no Ant use o atributos source no <javac>)*