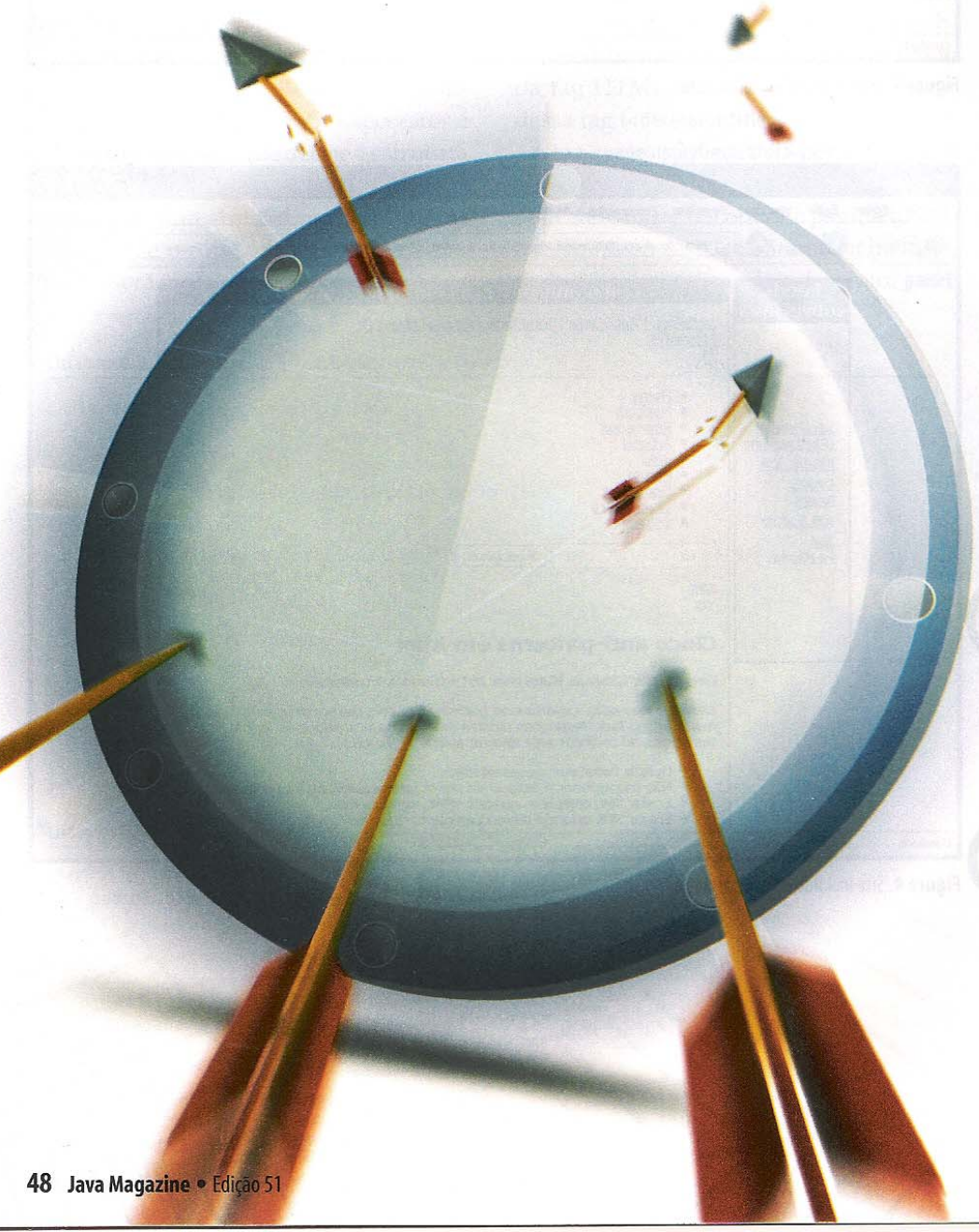
Tratamento de Exceções

Conceitos básicos, armadilhas e boas práticas

Neste artigo, vamos explorar desde os conceitos básicos o tratamento de exceções em aplicações Java, traçando um paralelo entre as situações encontradas no dia-a-dia e o impacto em suas aplicações. Falamos também um pouco sobre questões levantadas sobre exceções checadas na linguagem Java, e apresentamos dicas e boas práticas para o uso deste recurso.



Conceitos iniciais

Uma exceção em programação pode ser vista como um evento que é disparado sempre que alguma coisa errada ou não prevista ocorre durante a execução. Esse evento altera o fluxo normal do programa, geralmente cancelando a execução de ações esperadas de um ou mais métodos. A linguagem Java e o Java SE fornecem recursos sofisticados para manipulação de exceções, através das cláusulas **try**, **catch** e **finally**. Há também todo um mecanismo de runtime para lidar com exceções.

Veremos que adotar uma boa estratégia para manipular exceções facilita na detecção de erros e torna as aplicações mais robustas (ou seja, mais tolerantes a erros). Veja na **Listagem 1** um exemplo de estrutura para o tratamento de exceções em Java.

Vamos comentar esse código, ao mesmo temo que apresentamos conceitos fundamentais. Uma cláusula **try** vem acompanhada de zero ou mais cláusulas **catch**, e uma ou nenhuma cláusula **finally**. Pelo menos uma destas cláusulas é obrigatória, sendo ilegal um bloco **try** sem nenhum **catch** nem **finally**.

Observe que a ordem das cláusulas **catch** é significativa. As classes de exceção **StringIndexOutOfBoundsException** e **ArrayIndexOutOfBoundsException** são subclasses de **IndexOutOfBoundsException**, e por serem mais especializadas, seus **catch** devem ser declarados antes. Já a ordem relativa dos **catch** de **StringIndexOutOfBoundsException** e de **ArrayIndexOutOfBoundsException** não é forçada pelo compilador, pois elas estão no mesmo nível hierárquico. Note ainda que tratamos a classe **Exception** por último, porque ela é a mais geral entre as utilizadas no exemplo.

Caso a ordenação da exceção mais específica para a mais geral não seja obedecida, o compilador apresentará um erro de compilação indicando que a exceção já foi tratada anteriormente. O código da **Listagem 2**, por exemplo, não compila porque a classe **Exception** é mais geral do que a classe **IndexOutOfBoundsException**.

A cláusula finally

O código contido no bloco **finally** sempre será executado, mesmo se uma exceção for lançada ou se um **return** for encontrado

na Prática

Entenda o mecanismo de controle e tratamento de exceções do Java e evite problemas comuns

André Diniz

dentro de um **catch** do mesmo bloco **try**. O bloco **finally** deve ser incluído quando desejamos garantir a execução de algum código de "limpeza" do método, por exemplo, o fechamento de arquivos, liberação de conexões de rede ou de banco de dados etc.

Vale ressaltar que existem situações excepcionais (!):

- O bloco **finally** não será executado se a JVM for encerrada explicitamente dentro de um bloco **try/catch**, através de uma instrução **System.exit()**, por falhas de hardware ou até mesmo por restrições de acesso a recursos do computador. Também existem situações em que o código escrito dentro do bloco **finally** também pode provocar uma exceção.
- Se um bloco **finally** executar um **return** ou lançar uma nova exceção, é dessa forma que o método terminará. Se algum bloco **catch** do mesmo método tiver sido executado (lembre que isso acontece *antes* do **finally**) e tiver retornado outro valor, ou gerado outra exceção, *este valor ou exceção serão simplesmente ignorados*. Isso pode ser bastante confuso; portanto recomenda-se evitar instruções **return** ou **throw** em blocos **finally**.

Hierarquia de exceções

A **Figura 1** mostra algumas classes importantes na hierarquia de erros e exceções de Java. A classe **Throwable** é a superclasse para todas as classes de erros e exceções em Java. A partir dela, temos as classes **Error** e **Exception**, e cada uma possui uma ramificação (veja nos links um site onde você pode ver uma árvore extensa de exceções do Java).

Categorias de exceções

Em Java, podemos separar as exceções em duas categorias: *checked* (checadas) e *unchecked* (não-checadas).

Checked exceptions

As classes de exceções checadas são derivadas de **Exception**, mas não de **RuntimeException**. O código a seguir mostra um método declarando uma exceção checada (**NumberFormatException** é uma subclasse direta de **Exception**):

```
public int stringToInt(String value)
  throws NumberFormatException{..};
```

Ao chamar o método **stringToInt()** do exemplo, o desenvolvedor é obrigado a tratar a exceção **NumberFormatException**. Caso contrário, um erro de compilação será apontado. Isso acontece devido à presença da cláusula **throws** declarada no método, e porque **NumberFormatException** é checada.

Note que, com isso, a responsabilidade de tratar a exceção passa a ser de quem vai utilizar o método e não de quem o codificou.

Em suma, as exceções checadas sempre obrigam ao desenvolvedor estabelecer uma política para o tratamento de exceções, seja utilizando **try/catch**, seja "propagando-a" através da cláusula **throws**.

Unchecked exceptions

A categoria das exceções não-checadas inclui as exceções que *não* derivam de **Exception** (nem mesmo indiretamente), e as que derivam de **RuntimeException** (veja a **Figura 1**). Exceções unchecked são "imprevistas" num sentido mais amplo: não devem ocorrer nem mesmo em cenários anormais previsíveis pelo desenvolvedor, tais como a entrada de dados inválidos. Dito de outra forma, a ocorrência de uma exceção unchecked quase sempre indica um bug da aplicação.

Por exemplo, imagine um método que re-

Listagem 1. Estrutura básica para o tratamento de exceções

```
try{
  /* código que pode lançar uma exceção. */
}
catch (StringIndexOutOfBoundsException e){
  /* tratamento da exceção mais especializada */
}
catch (ArrayIndexOutOfBoundsException e){
  /* tratamento da exceção mais especializada */
}
catch (IndexOutOfBoundsException e){
  /* tratamento da exceção geral */
}
catch (Exception e){
  /* tratamento da exceção mais geral */
}
finally{
  /* trecho de código que sempre vai ser executado
  (exceto se a JVM for encerrada antes) */
}
```

Listagem 2. Tratamento de exceções errado porque não obedeceu a hierarquia de exceções

```
try{
  /* código que pode lançar uma exceção. */
}
catch (Exception e){
  /* tratamento da exceção mais geral */
}
/* Erro de compilação neste ponto porque Exception já "engloba" a IndexOutOfBoundsException */
catch (IndexOutOfBoundsException e){
  /* tratamento da exceção mais específica */
}
```

cebe uma **String** como argumento e retorna os três primeiros caracteres dessa **String**:

```
public String getPrefixo(String texto){
  return value.substring(0,3);
}
```

Aparentemente, não há nada de errado neste método, mas o fato é que o método **getPrefixo()** falha em não validar o valor recebido. Se este método receber um **null** para **texto**, ele irá lançar uma **NullPointerException**; se receber uma string vazia, lançará **StringIndexOutOfBounds Exception**. Ambos os casos são bugs do método **getPrefixo()**, e não deveriam ocorrer em nenhuma circunstância.

Alguém até poderia argumentar que o bug está no código que invocou **getPrefixo()**, por ter passado um parâmetro ilegal. Mas seguindo boas práticas da orientação a objetos, cada método deve assumir a responsabilidade pela validação dos dados que recebe. Veja então uma revisão do método, lançando uma exceção não-checada:

```
/**
 * Obtém o prefixo do valor.
 * @param value String não-nula, com pelo menos 3 caracteres.
*/
public String getPrefixo(String value) {
   if (value == null)
     throw new IllegalArgumentException
        ("Argumento nulo");
   if (value.length() < 3)
     throw new IllegalArgumentException
        ("Tamanho do argumento é menor que 3");
   return value.substring(0,3);
}
```

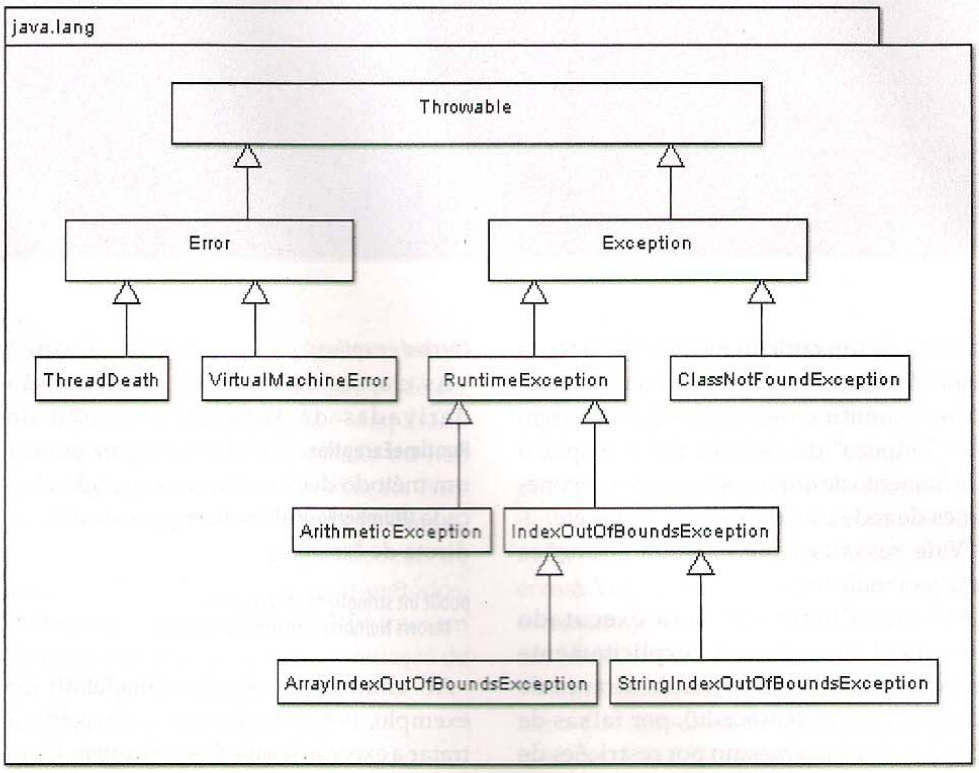


Figura 1. Árvore parcial de exceções em Java.

É importante entender que, nos dois casos, a chamada do método **getPrefixo()** passando argumentos inválidos continuará dando erro. Não há como evitar isso. Porém o novo código é melhor, porque gera uma exceção mais específica e com uma mensagem de erro mais clara. Além disso, também é importante documentar os valores válidos para o parâmetro.

Como as exceções não-checadas costumam indicar bugs, é péssimo hábito tratá-las com **catch**. Não existe motivo defensável para se tolerar, por exemplo, um método que gera uma **NullPointerException**. Mas nem todas as exceções checadas são iguais. No exemplo, tratamos exceções de "baixo nível" como **NullPointerException**, mas emitimos uma **IllegalArgumentException** *que também é unchecked*. Esta exceção pelo menos é de mais alto nível: não só indica a causa do problema, mas também o localiza melhor, pois deixa claro que o bug está no método que passou o parâmetro ilegal para **getPrefixo()** – e não no método que recebeu estes parâmetros e emitiu a exceção. Ainda assim, é uma exceção que nunca deve ocorrer em um programa sem bugs.

Polêmica sobre as checked exceptions

Atualmente, existe certa polêmica entre os desenvolvedores sobre a real necessidade das checked exceptions. Algumas linguagens de programação, tais como com C++, C# e Python, só possuem as unchecked exceptions; Java foi a primeira a adotar as do tipo checked.

Apesar de a Sun recomendar as checked exceptions sempre que criarmos exceções customizadas (especialmente de "nível de aplicação" – coisas como **ChequeSemFundosException** etc.), alguns autores afirmam que as checked exceptions atrapalham mais do que ajudam. O problema é que as elas forçam o tratamento da exceção e certos desenvolvedores, apenas para evitar um erro de compilação, terminam mascarando o problema, por exemplo escrevendo cláusulas **catch** vazias. Mas não é justo julgar um recurso da linguagem a partir de código que o usa de forma errada! É como dizer que a obrigatoriedade do uso de cintos de segurança é ruim porque alguém pode sair dirigindo bêbado

a 200 km/h e se julgar seguro porque está com o cinto.

Uma crítica mais válida é que às vezes é difícil decidir se uma exceção deve ser checked ou unchecked, e esta decisão nem sempre é feita corretamente. Por exemplo, certas exceções das APIs do Java, como **RemoteException**, são muito criticadas pela sua tendência em poluir o código. Mas se você concorda com esta crítica, isso apenas quer dizer que os designers destas APIs cometeram um erro, e talvez estas exceções deveriam ter sido unchecked. De qualquer maneira, isso não significa que o recurso não seja correto e útil para muitos outros casos.

Evitando armadilhas

Vamos explorar alguns trechos de código comumente encontrados em aplicações Java e que devem ser evitados para que tenhamos um controle eficiente de exceções. Mostramos um pequeno trecho para cada situação e o comentamos em seguida. Nos exemplos, vamos considerar que o método **processar()** pode lançar uma exceção checada.

Bloco catch vazio

```
public void executar() {
   try{
      processar();
   }
   catch (Exception e){ }
}
```

Esse é o mais perigoso dos casos. O programador mascara a exceção apenas para se livrar das mensagens do compilador. Dessa forma, se o método **processar()** lançar uma exceção, nada vai acontecer, e o programa vai continuar "funcionando" como se nada de errado tivesse acontecido. Bugs da aplicação podem ser mascarados, e se acontecerem em produção, você não terá nem mesmo uma mensagem ou log de erro para ajudá-lo a diagnosticar o problema.

Bloco com printStackTrace()

```
public void executar() {
   try{
      processar();
   }
   catch (Exception e){
      e.printStackTrace();
   }
}
```

Neste caso muito comum, o programador, por ser obrigado a tratar a exceção, apenas registra no console o erro ocorrido, através do método **printStackTrace()** da classe **Exception**. Apesar de ser melhor que o caso anterior, pois a informação sobre o erro não é perdida, provavelmente o usuário não vai tomar conhecimento do que aconteceu (a não ser que o programa tenha interface de linha de comando, ou que a saída do console seja interceptada por um arquivo de log).

Transferência de responsabilidade

```
public void executar() throws Exception {
   processar();
}
```

Neste caso, o programador se livra das reclamações do compilador, passando a responsabilidade do tratamento da exceção para quem vai chamar o método **executar()**. Aparentemente não há nada de errado nisso, mas a pergunta que precisamos fazer é: será que essa responsabilidade não é mesmo desse método? Note que quem implementou **processar()** também passou a responsabilidade "para cima"!

Não é proibido usar cláusulas **throws** para passar adiante a responsabilidade pelo tratamento de uma exceção, mas isso não deve ser feito pelo motivo errados, como a preguiça de escrever um bloco **try/catch**.

Boas práticas

Podemos tornar nossas aplicações mais robustas e fáceis de manter quando evitamos as armadilhas impostas pelo mecanismo de exceções do Java. Vejamos algumas boas práticas em resumo.

- Não use exceções para controlar o fluxo do programa. Exceções devem ser usadas para tratar situações anormais.
- Para decidir quando usar exceções checadas ou não, responda às seguintes perguntas: a exceção pode acontecer numa execução normal do código (sem bugs)? O usuário precisa ser informado sobre o problema para que alguma ação seja tomada? Se responder sim a alguma dessas perguntas, utilize uma exceção checada.
- Nunca ignore as exceções. Quando um método provocar uma ou várias exceções, trate-as efetivamente, sem cair nas armadilhas citadas no tópico anterior.
- Faça o log das exceções que podem indicar bugs.

Conclusões

Apresentamos neste artigo alguns pontos importantes sobre a manipulação de exceções – conceitos e exemplos que nos ajudam a entender melhor o mecanismo fornecido pelo Java. O tratamento de exceções pode parecer à primeira vista uma tarefa cansativa e desinteressante, mas as situações que geram exceções são uma realidade inevitável para a maioria das aplicações. E você pode ter certeza que teria bem mais trabalho para lidar com erros e situações imprevistas, de forma robusta, numa linguagem que não possuísse uma facilidade de tratamento de erros sofisticada como a do Java. ●

java.sun.com/docs/books/tutorial/essential/exceptions
Tutorial sobre controle de exceções fornecido pela Sun Microsystems.

java.sun.com/j2se/1.5.0/docs/api/index.html
Hierarquia do pacote *java.lang* completa.

java.sun.com/docs/books/tutorial/essential/exceptions/runtime.html
A controvérsia das checked exceptions.

André Diniz
(andre.l.diniz-politec@hsbcglt.com.br)
é bacharel em Ciência da Computação (UNIFACS – Universidade Salvador), e atualmente é Software Analyst do HSBC Global Technology Brazil. Também é pós-graduado em Sistemas de Informação com Ênfase em Componentes Distribuídos e Web, pela Faculdade Ruy Barbosa.