Exceções em Java

Miguel Jonathan – DCC/IM/UFRJ *(rev. abril de 2011)*

Resumo dos conceitos e regras gerais do uso de exceções em Java

O uso de exceções permite separar a detecção da ocorrência de uma situação excepcional do seu tratamento, ao se programar um método em Java.

Na forma antiga de se programar, era comum embutir as ações a tomar em cada teste de erro.
Por exemplo, uma função hipotética f() ficava assim:

```
void f() {
     if (<teste da condição de erro 1>) {
          <comandos que determinam o que fazer se o erro 1 ocorreu>
     }

     else if (<teste da condição de erro 2>) {
          <comandos que determinam o que fazer se o erro 2 ocorreu>
     }

     else if....<testando e tratando outros possíveis erros>

     else {
          <comandos para processar a função em condições normais>
     }
}
```

Onde essa forma de programar prejudica o bom design das aplicações?

a) Se um método for usado em aplicações diferentes, o mesmo erro sempre será tratado da mesma maneira. Isso limita a flexibilidade de lidar com situações de exceção.

b) Se for necessário alterar o procedimento a seguir no caso de um determinado erro, o método na forma acima terá que ser alterado. Isso introduz riscos e obrigará a re-testar todo o método, inclusive as partes que já estavam funcionando corretamente.

Como funciona o mecanismo de exceções:

Uma exceção em Java é um objeto da classe `java.lang.Exception`, ou de uma de suas subclasses. Como todo objeto, a exceção é capaz de armazenar dados nas suas variáveis de instância. Quando um erro ou situação anormal é encontrado durante a execução de um método, um objeto exceção é construído, e diversos dados sobre essa ocorrência são registrados nos campos desse objeto.

Nesse momento, o método onde ocorreu a exceção aborta, e o controle da execução retorna ao método que o chamou.

Além disso, por um mecanismo especial, o objeto exceção que foi construído é também enviado ao método chamador. Diz-se que o método onde ocorreu o erro "lança" a exceção para o método que o chamou.

Suponha um método qualquer (por exemplo, `main()`) que chama um método `g()`:

```
public static void main (String[] args) {
     ......
     ......
     g();
     ......
     ......
}
```

Suponha também que, de dentro de `g()`, o método `f()` seja chamado:

```
public void g() {
     ......
     ......
     f();
     ......
     ......
}
```

Vamos admitir que no método `f()` podem ocorrer dois tipos de erros ou situações excepcionais, a exceção A e a exceção B. Usando exceções, o método `f()` poderia ser escrito da forma abaixo.
Obs.: O comando **`throw`** é que se encarrega de lançar a exceção para o método chamador:

```
public void f() {
     if (<teste da condição de erro A>) {
        //constrói uma exceção da classe ExcecaoA e lança para quem chamou f()
        throw new ExcecaoA(....lista de argumentos...);

     else if (<teste da condição de erro B>) {
        //constrói uma exceção da classe ExcecaoB e lança para quem chamou f()
        throw new ExcecaoB(....lista de argumentos...);
     }

     else ....<testando outros possíveis erros e procedendo de forma similar>

     else {
          <comandos para processar f() em condições normais sem erro>
     }
}
```

Agora o método `f()` não precisa mais determinar o que fazer quando cada caso de erro ocorrer. Ele precisa apenas detectar qual o caso de erro que ocorreu. A partir daí, ele constrói, e "lança" para que o chamou, um objeto especial da classe `Exception` (ou de alguma subclasse). Ao construir esse objeto, o método `f()` insere nele as informações que permitirão entender qual erro ocorreu, e qual era o estado da aplicação no momento do erro. Esse objeto poderá ser "capturado" pelo método `g()`, e "tratado" lá, ou mesmo ser novamente lançado por `g()` para ser capturado e tratado por quem o chamou, no caso o `main()`.

Vamos supor que as exceções do tipo ExcecaoA que ocorrerem em f() devam ser capturadas e tratadas apenas pelo método main(). E que as exceções do tipo ExcecaoB devam ser capturadas e tratadas no método g().
Nesse caso, os métodos main() e g() devem ser escritos assim:

```
==========================================
public static void main (String[] args) {
     ......
     ......
// g() lançará exceções tipo ExcecaoA, caso ocorram dentro de f()
// mas vai capturar e tratar as exceções tipo ExcecaoB, que nunca chegarão a main
     try{
       g();
     }
     catch(ExcecaoA exa){
       ....comandos para examinar a exceção referenciada por exa...
       ....comandos para tratar o erro A...
       ..................................
     }
     ......
     ......
}
==========================================

// O cabeçalho informa o compilador (e os usuários) que g() pode lançar ExcecaoA
public void g() throws ExcecaoA {
     ......
     ......
// O bloco try permite capturar exceções e tratá-las nos blocos catch associados:
     try{
       f();
     }
     catch(ExcecaoB exb){
       ....comandos para examinar a exceção referenciada por exb...
       ....comandos para tratar ExcecaoB...
       ..................................
     }
     ......
     ......
}
==========================================
```

Note que exceções do tipo B que ocorram em f() jamais chegam a main(), pois são sempre capturadas em g().

Mas as exceções do tipo A lançadas por f() não são capturadas em g(), e são por ele re-lançadas para main(), onde são finalmente capturadas e tratadas.

O programador tem agora mais flexibilidade para escolher em que ponto da aplicação os erros serão tratados, e de que forma. Apenas no caso de o próprio `main()` não capturar a exceção é que ocorrerá o encerramento anormal do programa (que irá "abortar"), sendo seguido da impressão na console de um relatório mostrando a seqüência de chamadas que originou o erro (essa seqüência é chamada de stack trace). Nos demais casos, o programa nunca aborta, mas os erros são capturados e tratados adequadamente.

Informando o compilador que o método poderá lançar uma ou mais exceções:

No final do cabeçalho de um método que poderá lançar exceções, coloca-se a informação:

```
  throws <lista das classes de exceção que o método poderá lançar>
```

Por exemplo:

```
public void f() throws NumberFormatException, IOException{
.....
}
```

Veremos mais adiante que para certas classes de exceção essa declaração é obrigatória, enquanto que para outras é opcional.

Capturando e tratando exceções: os blocos `try { }`, `catch(){ }`, e `finally { }`

Quando programamos um método em Java, e dentro desse método existirem comandos ou chamadas de métodos onde podem ocorrer uma ou mais exceções, temos a opção de envolver esses comandos em um bloco `try` para capturar e tratar essas exceções dentro do método.

```
try {
  <comandos>
}
```

Um bloco `try` é normalmente seguido de um ou mais blocos `catch`, que possuem o seguinte formato:

```
catch (T e){
   <comandos para tratar a exceção apontada por e>
}
```

onde `T` é um tipo de exceção, ou seja, o nome da classe `Exception` ou uma de suas subclasses.
No caso de algum comando dentro do bloco `try` lançar uma exceção, a execução do bloco será interrompida, e o controle passará para o primeiro bloco `catch` que tenha um parâmetro de tipo compatível com a exceção lançada. Podem haver zero, um ou mais blocos `catch` após um bloco `try`. Caso não haja nenhum bloco `catch` compatível com o tipo da exceção, ele será lançada para o método que chamou o método atual.

O bloco finally

Tipicamente, um bloco `finally` contém comandos de liberação de recursos alocados no bloco `try` (tais como abertura de arquivos, de banco de dados, etc). Se esses comandos ficassem no final do bloco `try`, poderiam nunca ser executados em caso de lançamento de exceção.

O bloco `finally` será sempre executado após o bloco `try` terminar normalmente, ou após algum bloco `catch` executar, mesmo que a saída desses blocos seja causada pelo lançamento de outra exceção não tratada, ou por comando `return`. O bloco `finally` somente não será executado se ocorrer antes uma chamada para terminar a JVM (máquina vitual Java), com `System.exit(0)`.

No caso do bloco `try` terminar sem lançamento de exceção, ou se houver exceção, mas ela for capturada por um bloco `catch` e este terminar normalmente, os demais comandos do método após o bloco `finally` continuarão sendo executados.

Os blocos `catch` e `finally` são opcionais, mas não é permitido haver apenas o bloco `try` sem pelo menos um bloco `catch` ou um bloco `finally` associado.

Por exemplo:

```
public void g() {
  ...comandos...
  try{
    f();
  }
  catch (NumberFormatException nfe){
      <comandos para tratar essa exceção>
  }
  catch (Exception e){
      <comandos para tratar qualquer outra exceção>
  }
}
```

Suponha que ao executar, o método `f()` lance uma exceção do tipo `NumberFormatException`. Ela será capturada pelo primeiro bloco `catch` acima. Se lançar outro tipo de exceção, ela será capturada pelo segundo bloco `catch`. Isso porque o tipo `Exception` pode apontar para qualquer exceção, por ser a superclasse de todas.

Veja um exemplo completo abaixo. O método `f(int x)` da classe Classe1 lançará uma exceção do tipo `NumberFormatException`, caso o valor do argumento x seja negativo, ou caso seja menor ou igual ao campo valor. Se o argumento for maior que 1000, o método lançará um exceção do tipo `Exception`.
O método lançará uma `ArithmeticException` se houver tentativa de divisão por zero. Essa exceção é lançada automaticamente.

```
public class Classe1 {
  public int valor;
  public Classe1 (int n){
       valor = n;
  }

  public void f(int x) throws Exception, NumberFormatException,ArithmeticException{
      if (x< 0)throw new NumberFormatException("Erro-Argumento negativo: "+ x);
      if (x<= valor)
        throw new NumberFormatException("Erro-Argumento deve ser maior que " + valor);
      if (x > 10000)throw new Exception("Erro-Argumento muito grande: "+ x);
      System.out.println (x/(valor-100));//
  }
}

public class TesteExcecoes {
       public static void main(String[] args) {
              Classe1 c1 = new Classe1(100);
              try{
       //       c1.f(200);
       //       c1.f(-20);
       //       c1.f(20000);
                c1.f(700);
              }
              catch(NumberFormatException nf){
                     System.out.println(nf);
              }
              catch(ArithmeticException ar){
                     System.out.println(ar);
              }
              catch(Exception e){
                     System.out.println(e);
              }
        finally{
              System.out.println("Terminou o método f()");
        }
       }
}
```

Exceções verificadas e não-verificadas:

A linguagem Java admite dois tipos de exceção: As *não verificadas* (*unchecked,* em inglês) são instâncias de subclasses de `RuntimeException`. O compilador não verifica se existe possibilidade de serem lançadas, e não exige que os

métodos onde possam ocorrer façam qualquer tratamento. Elas representam erros ou defeitos na lógica do programa que podem causar problemas irrecuperáveis em tempo de execução (run time).

Por outro lado, instâncias de `Exception`, ou de qualquer outra de suas subclasses, são verificadas (checked) como, p.ex, `IOException`, `ClassNotFoundException` e `CloneNotSupportedException`. Elas representam erros que podem ocorrer em tempo de execução, mas que não dependem da lógica do programa, em geral defeitos nos dispositivos de entrada ou saída (arquivos, rede, etc). O compilador exige que um método onde possam ocorrer exceções verificadas faça uma de duas coisas: ou utilize blocos try-catch-finally para capturar e tratar essas exceções, ou declare que pode lançar essas exceções, colocando uma cláusula "throws" no seu cabeçalho, como por exemplo:

```
public void M() throws IOException, CloneNotSupportedException {
..............
}
```

Essa cláusula é facultativa para o caso de exceções não-verificadas.

Construtores:

A classe `java.lang.Exception`, e todas as suas subclasses, têm pelo menos dois construtores da forma:

a) `<nome da classe de Exceção> (String <mensagem de erro>)`

b) `<nome da classe de Exceção> ()`

A mensagem de erro é sempre retornada pelo método `toString()`.

Toda exceção também aceita a mensagem `printStackTrace()`, que imprime na stream apontada por System.err um stack trace. O stack trace é um relatório detalhado da seqüência de chamadas a métodos que antecederam o lançamento da exceção.

Exemplos:

Em cada um dos exemplos seguintes, uma classe teste tenta executar o método `f()` de uma de duas classes, A ou B. Na classe A, o método `f()` lança uma exceção do tipo "não verificada", no caso é uma `NumberFormatException`. Na classe B, o método `f()` lança uma exceção do tipo "verificada", no caso uma `IOException`. Os diversos exemplos mostram o uso de blocos `try{}, catch{}` e `finally {}` de várias formas, e os comentários esclarecem sobre os efeitos decorrentes.

```
public class A {
/**
Neste exemplo, a classe A tem um metodo f() que pode lançar uma exceção do tipo
NumberFormatException, que e' nao verificada. Por esse motivo, o método f() não precisa
incluir a terminação "throws NumberFormatException".
**/
       public void f(int a){
          if (a<20) throw new NumberFormatException();
          System.out.println("a = "+ a);
       }
}
```

```
import java.io.IOException;
public class B{
/**
Nesse exemplo, como IOException é uma exceção verificada, o compilador exige
que o método f() declare explicitamente que pode lançar a exceção, colocando a frase
"throws IOException" no seu cabeçalho.
**/
    public void f(int a) throws IOException {
      if (a<20) throw
        new IOException ("valor do argumento de f() e' " + a + " (menor que 20)");
      System.out.println("a = "+ a);
   }
}
```

```
public class TesteExc1{
/**
Neste exemplo, a exceção será capturada, e as três mensagens serão exibidas.
Ou seja, mesmo depois de finally executar, o restante do método main continua.
```

```
**/

       public static void main(String[] args){
           try{
                A x = new A();
                int a = 4;
                x.f(a);
           }
           catch(Exception e){
               System.out.println("valor ilegal de a");
           }
           finally{
              System.out.println("fim do bloco try em TesteExc");
           }
           System.out.println("fim do metodo main em TesteExc");
       }
```

```
public class TesteExc2 {
/**
Neste exemplo, o bloco catch não existe. Portanto, a exceção não será capturada, gerando
um stack trace. O bloco finally e' executado, mas não o que segue depois.
**/
       public static void main(String[] args){
           try{
                A x = new A();
                int a = 4;
                x.f(a);
           }

           finally{
                System.out.println("fim do bloco try em TesteExc");
           }
           System.out.println("fim do metodo main em TesteExc");
       }
}
```

```
public class TesteExc3 {
/**
Neste exemplo, o bloco catch não existe, apenas o try e o finally.
Com esse valor de a, a exceção não será lançada.
Nesse caso, o código depois do bloco finally também será executado.
**/
       public static void main(String[] args){
           try{
                A x = new A();
                int a = 34;
                x.f(a);
           }
           finally{
                System.out.println("fim do bloco try em TesteExc");
           }
           System.out.println("fim do metodo main em TesteExc");
       }
}
```

```
public class TesteExc4 {
/**
Neste exemplo, como a exceção que pode ser lançada por f() e' não verificada,
o compilador não reclama por não haver a cláusula throws no cabeçalho de main.
Mas a exceção será lançada, originando um stack trace, e o método main()
não continuará após o ponto da chamada de f().
**/
       public static void main(String[] args){
              A x = new A();
              int a = 4;
              x.f(a);  // com esse valor, f() lancara' excecao
              System.out.println("fim do metodo main em TesteExc");
       }
}
```

```
public class TesteExc5 {
/**
Neste exemplo, como a exceção que pode ser lançada por f() e' do tipo "não verificada", o
compilador não reclama do fato de main() não informar que pode lançar uma exceção, com
"throws NumberFormatException" ou "throws Exception".
Como nesse exemplo a exceção não será lançada, o método main irá até o final.
**/
       public static void main(String[] args){
                A x = new A();
                int a = 34;
                // com esse valor, f() nao lancará exceção
                x.f(a);
                System.out.println("fim do metodo main em TesteExc");
       }
}
```

```
import java.io.IOException;
public class TesteExc6 {
/**
Neste exemplo, usa-se a informação contida no objeto exceção para gerar a mensagem de
erro, pois o método f() da classe B cria exceções com uma mensagem informativa.
**/
       public static void main(String[] args){
           try{
                B x = new B();
                int a = 4;
                x.f(a);
           }

           catch(IOException e){
                System.out.println(e);  // imprime toString(e)
           }
           finally {
                System.out.println("fim do bloco try em TesteExc");
           }
           System.out.println("fim do metodo main em TesteExc");
       }
}
```

```
import java.io.IOException;
public class TesteExc7 {
/**
Neste exemplo, o compilador reclama porque a exceção que pode ser lançada por f()
é do tipo "verificada" (IOException), e o método main() não tem a clausula "throws
IOException"
**/
       public static void main(String[] args){
              B x = new B();
              int a = 34;
              x.f(a);
               System.out.println("fim do metodo main em TesteExc");
       }
}
```

```
import java.io.IOException;
public class TesteExc8{
/**
Neste exemplo, a exceção que pode ser lançada por f() e' verificada (IOException),
e o método main() tem a clausula "throws IOException", compilando OK.
**/
       public static void main(String[] args) throws IOException{
              B x = new B();
              int a = 4;
              x.f(a);
             System.out.println("fim do metodo main em TesteExc");
       }
}
```