Tratamento de erros com exceções

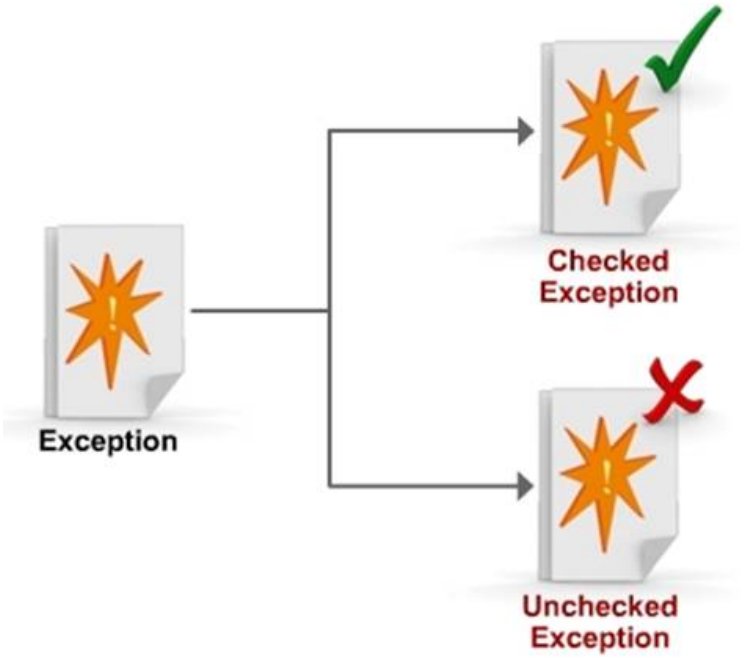
Objetivos da seção

- Aprender a lidar com condições especiais normais e condições especiais anormais, utilizando o mecanismo de exceção
  - Exemplo: parâmetros incorretos, tentar acessar arquivos inexistentes

Por que estudar isto?

- Porque nem todos os erros podem ser detectados em tempo de compilação
  - Muitos problemas precisam ser resolvidos em tempo de execução
  - As exceções permitem que o lugar onde o erro ocorreu seja diferente do lugar onde o erro vai ser tratado e onde ele será tratado que informações sobre o problema estejam disponíveis
- Queremos construir programas robustos
- Simples de usar
  - Relação custo/benefício é muito boa
- A idéia é que quando uma situação diferente ocorrer (exceção) alguém saberá lidar com ela!
  - Não é preciso ficar testando por cada erro particular que poderia ocorrer em todo o código
- Para cada exceção que possa ocorrer precisa haver apenas um cliente que vai tratar essa exceção: o *exception handler*

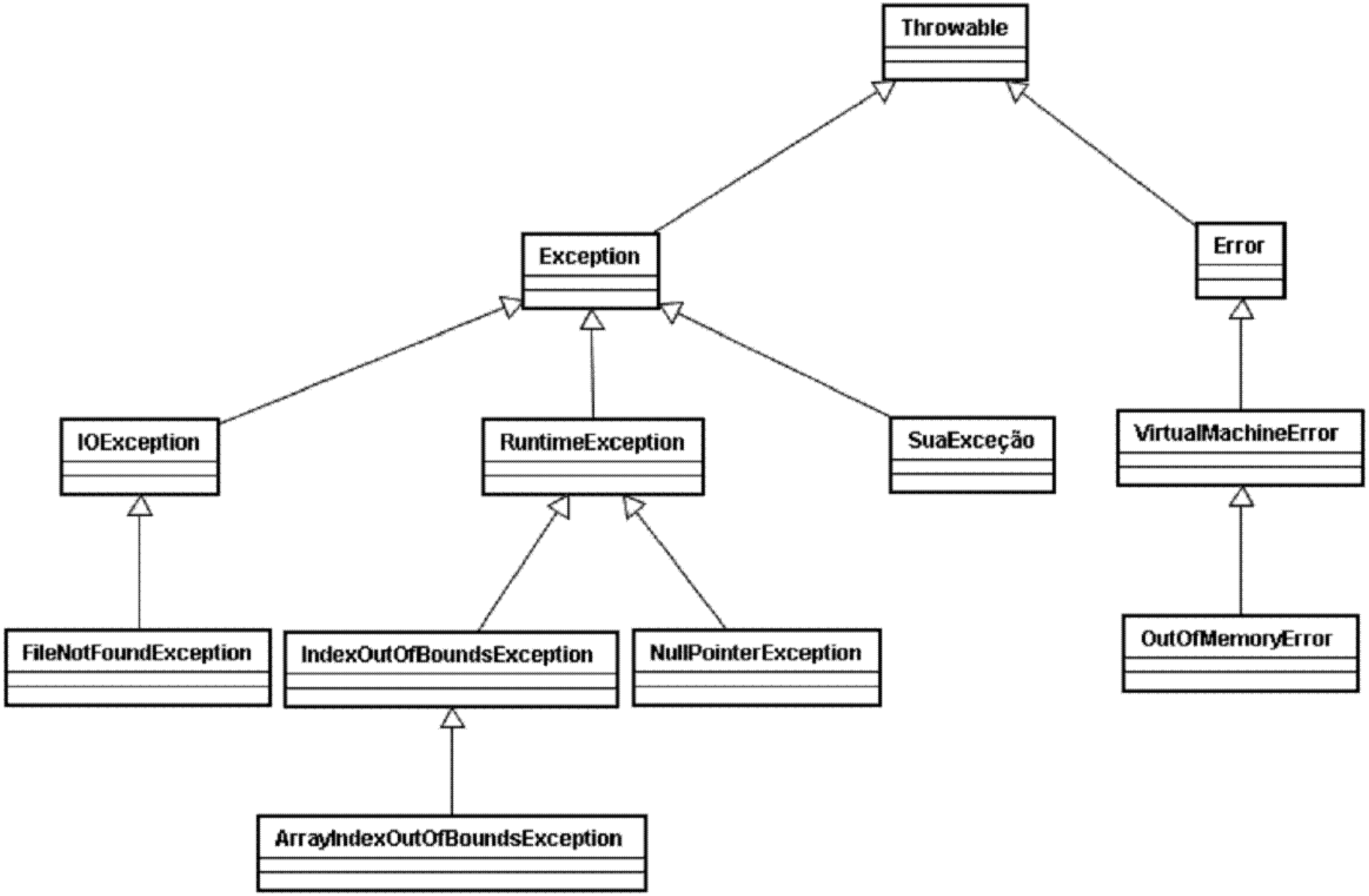
Categorias de Exceções



- **Unchecked**: Você não precisa estar preparado para elas e nem tratá-las se não quiser, provavelmente representam bugs.
  - Ex: tentativa de acessar uma posição fora do array, falhas do ambiente (out of memory)
- **Checked**: você é obrigado a declarar no retorno do método e tratar ou relançar! (como já vimos anteriormente)
- Podemos incluir mais informação numa exceção além da mensagem
  - O compilador ajuda a garantir que você está ciente de todas as exceções checáveis que podem ser lançadas quando você chama um método

Exceções básicas

- A hierarquia de exceções em Java é parecida com o que segue:



- **Error** é lançada quando há um erro interno do Java (é raro) – out of memory, por exemplo
- **RuntimeException** (NullPointerException, ...) é lançado quando seu programa tem um bug que você não tratou
- **Error** e **RuntimeException** são "*unchecked*" – são erros do programa dos quais você não vai

conseguir se recuperar (divisão por zero, acesso a arrays usando índice fora dos limites do array, acesso a métodos de objetos null...)

- O resto é "*checked*" - alguém dentro do programa tem condições de se recuperar dessa exceção, então faz sentido capturá-la e tratá-la em algum momento

Exemplo exceção

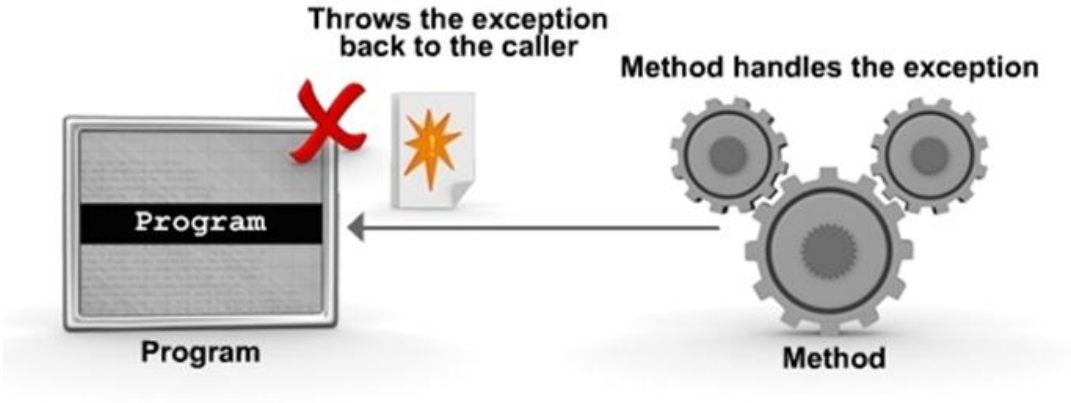
```
public class A {
    public static void main(String args[]) {
           int sum = 0;
           for ( int i = 0; i < args.length; i++ ) {
                sum += Integer.parseInt(args[i]);
           }
           System.out.println("Sum = " + sum);
    }
}
```

- O código acima funciona corretamente se todos os argumentos forem inteiros.
- Porém, ocorre uma falha se um dos argumentos não for inteiro.

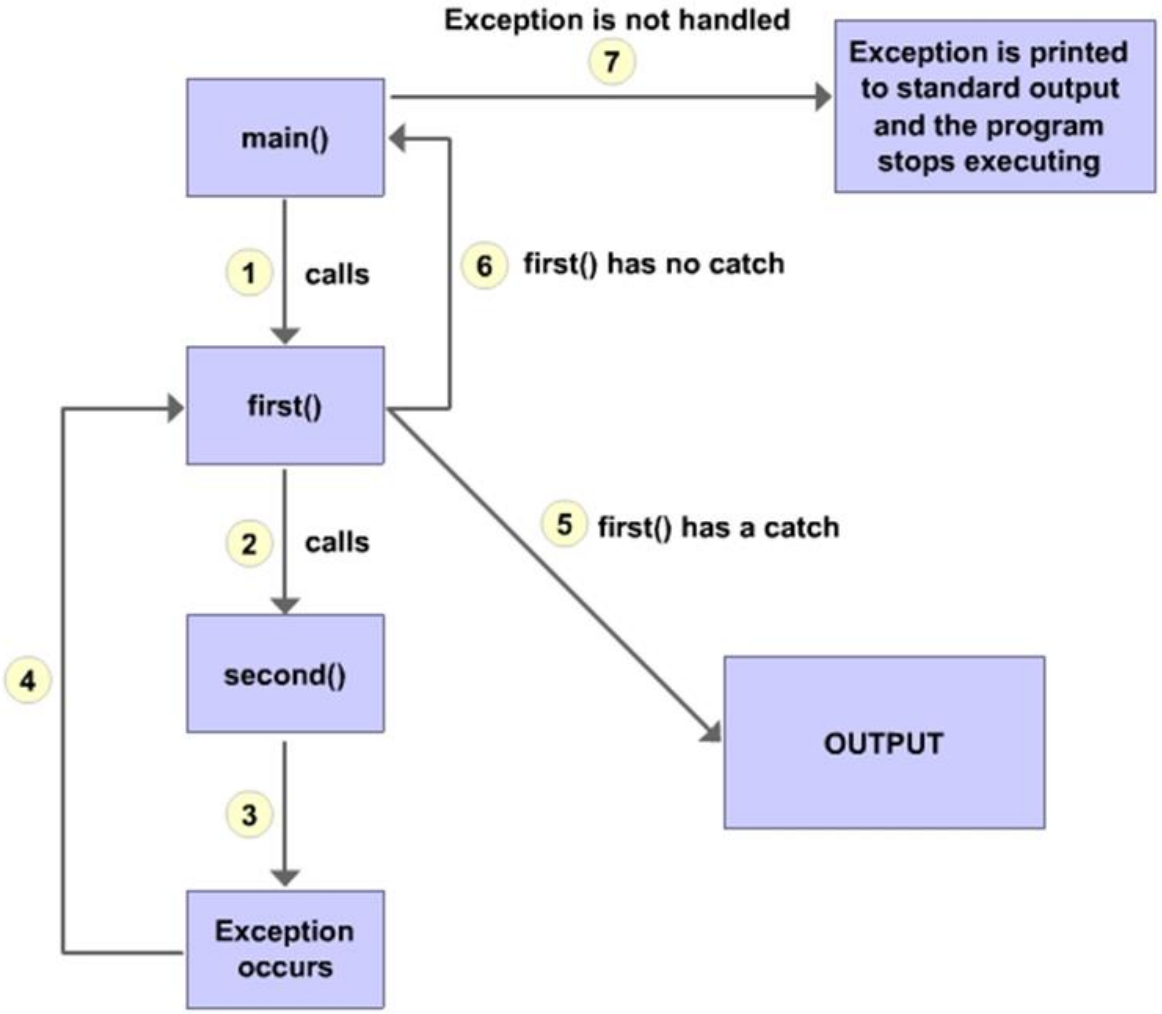
```
Exception in thread "main" java.lang.NumberFormatException: For input string: "oi"
   at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
   at java.lang.Integer.parseInt(Integer.java:447)
   at java.lang.Integer.parseInt(Integer.java:497)
   at A.main(A.java:5)
```

Capturando exceções

- Podemos capturar tipos específicos de exceções ou qualquer exceção que possa ocorrer



- O método que encontra o erro pode tratá-lo ou lançá-lo para o método que o chamou. Quando uma exceção alcança o topo da thread, a thread é terminada.
- Exemplo de fluxo de uma exceção:



- É importante diferenciar o descobrimento do erro e o tratamento do erro
  - › É muito frequente descobrir algo errado em um lugar mas querer tratar o erro em outro lugar

Blocos Try-catch

```
public class A {
```

```
    public static void main(String args[]) {
      try {
         int sum = 0;
         for ( int i = 0; i < args.length; i++ ) {
              sum += Integer.parseInt(args[i]);
         }
         System.out.println("Sum = " + sum);
      } catch (NumberFormatException nfe) {
         System.err.println("One of the command-line "
                            + "arguments is not an integer.");
      }
    }
}
```

- Captura a exceção que foi lançada e faz o tratamento.
  - Qual a saída do programa para os argumentos: 2 4 “oi”?
- Melhorando o tratamento:

```
public class A {
   public static void main (String args[]) {
       int sum = 0;
       for ( int i = 0; i < args.length; i++ ) {
           try {
               sum += Integer.parseInt(args[i]);
           } catch (NumberFormatException nfe) {
               System.err.println("[" + args[i] + "] is not an integer"
                                  + " and will not be included in the sum.");
           }
       }
       System.out.println("Sum = " + sum);
```

```
    }
}
```

- Dado os argumentos 2 4 “oi”, a saída do programa será:

```
[oi] is not an integer and will not be included in the sum.
Sum = 6
```

- **Dica:** apesar das exceções “unchecked” geralmente estarem relacionadas com bugs. Não é o que acontece no exemplo acima com a exceção **`NumberFormatException`** que é dessa categoria. Portanto, em alguns casos pode ser conveniente tratar exceções “unchekeds”.

- Podem existir vários blocos “catch”, cada um tratando uma exceção específica.
  - A ordem dos blocos é importante, pois uma exceção pode ser capturada por mais de um bloco.
  - Trate as mais específicas primeiro. Se, por exemplo, o método que você vai chamar puder lançar as exceções FileNotFoundException e IOException e você quer tratar FileNotFound de forma diferente de IOException, então o seu bloco try/catch deve ter a seguinte ordem de catchs.

```
try {
        …
```

```
} catch (FileNotFoundException e) {
    System.err.println("FileNotFoundException: "
                       + e.getMessage());
    throw new SampleException(e);

} catch (IOException e) {
    System.err.println("Caught IOException: "
                       + e.getMessage());
}
```

- Um catch tenta capturar uma exceção do tipo que ela indica ou qualquer exceção que seja filha dela. No exemplo acima, se colocarmos o catch pra IOException antes, o próximo catch de FileNotFound nunca será executado.

O Bloco Finally

- Define um bloco de código que sempre será executando, independente da ocorrência de uma exceção
- Exemplo:

```
        try {
           iniciaConexaoBD();
               User user = UserBO.getUser("gustavo");
            } catch (UserException e) {
                System.err.println("Usuário não encontrado");
```

```
        } finally {
        fechaConexaoBD();
        }
```

- Um bloco finally vai quase sempre executar
  - Ele não executará se no bloco try ou no bloco catch System.exit() for chamado
  - Ele executará se no bloco try ou no bloco catch existir uma declaração de return, inclusive, o valor de retorno do bloco finally irá sobrepor qualquer valor de retorno que existisse nos blocos try ou catch
  - Ele executará se uma exceção *unchecked* ocorrer no bloco try ou catch

Declarando uma exceção que o método pode lançar

```
public void iniciaConexao() throws ConexaoException {...}
```

- **Dica:** ao sobrescrever um método que lança uma exceção através de polimorfismo, o método só poderá lançar exceções da mesma classe ou de subclasses.

Duas formas de testar lançamento de exceções

- O JUnit 4 nos oferece dois modelos para testar exceções: em um deles podemos testar várias exceções no mesmo método de teste e no outro apenas uma exceção no método de teste
- Veja o código que lança a exceção

```
package p2.exemplos;

import java.util.Calendar;
```

```
public class Amigo {
   private String nome;
   private Calendar aniversario;

   public Amigo(String nome, Calendar niver) throws Exception {
      if(nome == null || nome.equals("")) {
         throw new Exception("Nome invalido.");
      }
      if(niver == null) {
         throw new Exception("Aniversario invalido.");
      }

      this.nome = nome;
      aniversario = niver;
   }

      ...
}
```

- Veja os possíveis testes que verificam se exceções estão sendo lançadas quando devem ser lançadas:

```
package p2.exemplos;

import java.util.GregorianCalendar;
import junit.framework.Assert;
import org.junit.Test;

public class AmigoTest {
```

```java
@Test(expected=Exception.class)
public void testConstrutorParamNomeNull() throws Exception {
   new Amigo(null, new GregorianCalendar());
}

@Test(expected=Exception.class)
public void testConstrutorParamNomeVazio() throws Exception {
   new Amigo("", new GregorianCalendar());
}

@Test(expected=Exception.class)
public void testConstrutorParamAniversarioNull() throws Exception {
   new Amigo("Raquel", null);
}

@Test
public void testaConstrutor() {
   try {
      new Amigo(null, new GregorianCalendar());
      Assert.fail("Devia ter lançado exceção de nome nulo.");
   } catch(Exception e) {
      Assert.assertEquals("Nome invalido.", e.getMessage());
   }

   try {
      new Amigo("", new GregorianCalendar());
      Assert.fail("Devia ter lançado exceção de nome vazio.");
   } catch(Exception e) {
      Assert.assertEquals("Nome invalido.", e.getMessage());
   }
```

```
        try {
            new Amigo("Raquel", null);
            Assert.fail("Devia ter lançado exceção de nome vazio.");
        } catch(Exception e) {
            Assert.assertEquals("Aniversario invalido.", e.getMessage());
        }
    }

}
```

- Note que se você usar o “expected” apenas uma exceção poderá ser testada por vez, caso contrário você terá código de teste não executado (o que vier depois do lançamento da primeira exceção)

Escrevendo suas próprias exceções

- Para criar uma nova classe de exceções devemos herdar de uma classe da hierarquia de exceções já existente
  - A que for semanticamente mais próxima da sua nova classe de exceções (nem sempre é possível)
- A maneira mais simples de criar sua exceção é como segue:

```
package p2.exemplos.exceptions;

public class UserException extends Exception {}
```

- Como usar esta nova exceção?
  - Use o comando “**throw**” para lançar sua exceção.

```
if (user == null) {
```

```
            throw new UserException();
        }
```

- Cadê o construtor?
  - › O compilador criou um construtor default, que simplesmente chama o construtor da classe base
- Poderíamos ter sobrescrito o construtor que recebe uma mensagem (String) que traz informação sobre a exceção:

```
package p2.exemplos.exceptions;

public class UserException extends Exception {
   public UserException(String message) {
       super(message);    }
}
```

- O mais importante é escolher um bom nome para a sua exceção!! A exceção acima poderia se chamar UserNotFoundException e não precisaria de uma mensagem.
- Você pode querer enviar as saídas de erro para a saída de erro padrão
  - › Para tal use: System.err
- É interessante identificar a seqüencia de métodos chamados até que a exceção ocorresse
  - › Exception. printStackTrace ()
    i) Mostra o trace na saída default de erro
  - › Exception. printStackTrace (System.out)
    i) Mostra o trace na saída padrão

Logando informações de erro

- Informações básicas sobre log
  - Você deve estudar mais sobre isso, especialmente para seu projeto de LP2 ;)
- Java oferece um serviço de log: http://java.sun.com/javase/6/docs/api/java/util/logging/package-summary.html
- Pode logar outras informações, que não sejam relacionadas às exceções que ocorrem no programa
  - Mas no contexto que estamos lidando aqui, vamos ver como usar este serviço para logar informações sobre exceções

```
package p2.exemplos;

import java.io.IOException;
import java.io.PrintWriter;
import java.io.StringWriter;
import java.io.Writer;
import java.util.logging.Logger;

public class LoggingException extends Exception {
   private static Logger logger = Logger.getLogger("LoggingException");

   public LoggingException() {
      Writer trace = new StringWriter();
      printStackTrace(new PrintWriter(trace));
      logger.severe(trace.toString());
   }
}
```

```
package p2.exemplos;

public class LogandoExcecoes {

    /**
     * @param args
     */
    public static void main(String[] args) {
        try {
            throw new LoggingException();
        } catch (LoggingException e) {
            System.err.println("Capturada: " + e);
        }
    }

}
```

- Isto faz com que cada exceção criada *logue* automaticamente o seu trace, isto é, os métodos chamados que levaram a ela
- Essa classe constrói toda a infra-estrutura de *log* sob a infra de exceções
- Geralmente iremos estar capturando e *logando* exceções criadas por terceiros
  - Então teremos que *logar* explicitamente, no momento de tratar as exceções
  - Veja o código abaixo

```
    public void meuMetodo() {
        Logger logger = Logger.getLogger("UserException");
        try {
            throw new UserException("Usuário não Encontrado");
        } catch (Throwable e) {
            logger.log(Level.SEVERE, e.getMessage());
```

```
        }
    }
```

- getMessage() retorna uma mensagem mais detalhada da exceção
  - O método getMessage() está para as exceções como o toString() está para as demais classes “normais”
  - Existem outros métodos que podem nos ajudar: getLocalizedMessage, toString, getStackTrace, etc.
    i) O programa a seguir não apresenta uma lista exaustiva de todos os métodos que podemos sobrescrever (você terá que descobrir isso sozinho(a) estudando a API de Java)
    ii)Você não precisará sobrescrever a maioria destes métodos na maioria das vezes, mas é bom saber o que você pode fazer!
    iii) Lembre-se de sempre ter bons nomes para suas exceções!

```
package p2.exemplos;

public class ExcecaoComMetodos extends Exception {
    @Override
    public String getMessage() {
        return "getMessage() de ExcecaoComMetodos";
    }

    @Override
    public String getLocalizedMessage() {
        return "getLocalizedMessage() de ExcecaoComMetodos";
    }

    @Override
    public String toString() {
        return "toString() de ExcecaoComMetodos";
```

```
    }
}
```

```
package p2.exemplos;

public class UsandoMetodosDeExcecoes {

   public static void main(String[] args) {
      try {
         throw new ExcecaoComMetodos();
      } catch (Exception e) {
         System.out.println("e.getMessage(): " + e.getMessage());
         System.out.println("e.getLocalizedMessage(): " +
e.getLocalizedMessage());
         System.out.println("e.toString(): " + e.toString());
         System.out.println("e.printStackTrace(): ");
         e.printStackTrace(System.out);
         //o default é ir para a saída padrão de erro!
      }
   }
}
```

- A saída deste programa é:

```
e.getMessage(): getMessage() de ExcecaoComMetodos
e.getLocalizedMessage(): getLocalizedMessage() de ExcecaoComMetodos
e.toString(): toString() de ExcecaoComMetodos
e.printStackTrace():
toString() de ExcecaoComMetodos
   at p2.exemplos.UsandoMetodosDeExcecoes.main(UsandoMetodosDeExcecoes.java:7)
```

Voltar