Tratamento de Erro e Javadoc

Professores:

Norton T. Roman

Fátima L.S.Nunes

Tratamento de Erro

```
import java.lang.String;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.lang.Integer;

public class teste {
        public static final double Pi = 3.142;
        private int divisor;

        teste(int div) {
                this.divisor = div;
        }

        public double executa() {
                return(Pi/divisor);
        }

        public static void main(String[] args) {
                System.out.print("Divisor: ");
                teste t = new teste(0);
                System.out.println(t.executa());
        }
}
```

O que vai acontecer?

Tratamento de Erro

```
import java.lang.String;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.lang.Integer;

public class teste {
        public static final double Pi = 3.142;
        private int divisor;

        teste(int div) {
                this.divisor = div;
        }

        public double executa() {
                return (Pi/divisor);
        }

        public static void main(String[] args) {
                System.out.print("Divisor: ");
                teste t = new teste(0);
                System.out.println(t.executa());
        }
}
```

O que vai acontecer?

Runtime Error:
Divisor: Infinity

Não testamos o divisor fornecido!

Tratamento de Erro

```
import java.lang.String;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.lang.Integer;

public class teste {
        public static final double Pi = 3.142;
        private int divisor;

        teste(int div) {
                if (div ==0)
                        System.out.println("Erro: O
divisor não pode ser zero!");
                this.divisor = div;
        }

        public double executa() {
                return(Pi/divisor);
        }

        public static void main(String[] args) {
                System.out.print("Divisor: ");
                teste t = new teste(0);
                System.out.println(t.executa());
        }
}
```

O que vai acontecer?

Tratamento de Erro

```
import java.lang.String;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.lang.Integer;

public class teste {
        public static final double Pi = 3.142;
        private int divisor;

        teste(int div) {
                if (div ==0)
                        System.out.println("Erro: O
divisor não pode ser zero!");
                this.divisor = div;
        }

        public double executa() {
                return(Pi/divisor);
        }

        public static void main(String[] args) {
                System.out.print("Divisor: ");
                teste t = new teste(0);
                System.out.println(t.executa());
        }
}
```

O que vai acontecer?

Divisor: Erro: O divisor não pode ser zero!
Runtime Error:
Divisor: Infinity

???

Tratamento de Erro

- Nada no código dizia para o programa parar, ou alertava para algo de errado
  - Ele rodou normalmente, como deveria
  - Continua na memória
- Poderia ser pior!
  - Divisor poderia ser armazenado em um BD, para uso compartilhado por outros sistemas.
- Como evitar isso? Exceções!

Tratamento de Erro

```
import ...
import java.lang.Exception;

public class teste {
        public static final double Pi = 3.142;
        private int divisor;

        teste(int div) throws Exception {
                if (div ==0)
                        throw new Exception("Erro: O divisor não
pode ser zero!");
                this.divisor = div;
        }

        public double executa() {
                return(Pi/divisor);
        }

        public static void main(String[] args) throws Exception {
                System.out.print("Divisor: ");
                teste t = new teste(0);
                System.out.println(t.executa());
        }
}
```

Lançamos a exceção, declarando no cabeçalho do método em que ela está

Em todo método que usar o código que gera a exceção, temos que propagá-la

Tratamento de Erro

```
import ...
import java.lang.Exception;

public class teste {
        public static final double Pi = 3.142;
        private int divisor;

        teste(int div) throws Exception {
                if (div ==0)
                        throw new
Exception("Erro: O divisor não pode ser zero!");
                this.divisor = div;
        }

        public double executa() {
                return(Pi/divisor);
        }

        public static void main(String[] args)
throws Exception {
                System.out.print("Divisor: ");
                teste t = new teste(0);

                System.out.println(t.executa());
        }
}
```

E qual a saída?

Divisor: Exception in thread "main"
java.lang.Exception: Erro: O divisor não pode
ser zero!
        at teste.<init>(teste.java:13)
        at teste.main(teste.java:27)

Tratamento de Erro

```
import ...
import java.lang.Exception;

public class teste {
        public static final double Pi = 3.142;
        private int divisor;

        teste(int div) throws Exception {
                if (div ==0)
                        throw new
Exception("Erro: O divisor não pode ser zero!");
                this.divisor = div;
        }

        public double executa() {
                return(Pi/divisor);
        }

        public static void main(String[] args)
throws Exception {
                System.out.print("Divisor: ");
                teste t = new teste(0);

        System.out.println(t.executa());
        }
}
```

E qual a saída?

Divisor: Exception in thread "main"
java.lang.Exception: Erro: O divisor não pode
ser zero!
        at teste.<init>(teste.java:13)
        at teste.main(teste.java:27)

Melhor... se houvesse alguma atualização a um BD ela não seria feita.

Além disso, o objeto não existe
na memória.

Tratamento de Erro

- Melhor, mas ainda assim não ideal
- Correto seria conseguir identificar o erro e tratar
- Como?
  - Tratamento de Exceções

Tratamento de Erro

- Capturando e Tratando Exceções
  - **Bloco Try – Catch – Finally**
  - Try
    - Testa o comando que pode gerar a exceção
  - Catch
    - Captura a exceção, executando um código que o programador define para tratá-la
  - Finally (Opcional)
    - O código dentro dele sempre será executado, mesmo se houver uma exceção
    - Se o código dentro do try ou catch contiver um return, o código dentro do finally será executado antes do retorno do método

```
import java.lang.String;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.lang.Integer;
import java.lang.Exception;

public class teste {
        public static final double Pi = 3.142;
        private int divisor;

        teste(int div) throws Exception {
                if (div ==0)
                        throw new Exception("Erro: O divisor não pode ser zero!");
                this.divisor = div;
        }

        public double executa() {
                return(Pi/divisor);
        }

        public static void main(String[] args) throws Exception {
                BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));

                System.out.print("Divisor: ");
                String resp = br.readLine();
                teste t = new teste(Integer.parseInt(resp));
                System.out.println(t.executa());
        }
}
```

Nova versão.

Agora lê do usuário (entrada padrão – geralmente teclado)

```
import java.lang.String;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.lang.Integer;
import java.lang.Exception;

public class teste {
        public static final double Pi = 3.142;
        private int divisor;

        teste(int div) throws Exception {
                if (div ==0)
                        throw new Exception("Erro: O divisor não pode ser zero!");
                this.divisor = div;
        }

        public double executa() {
                return(Pi/divisor);
        }

        public static void main(String[] args) throws Exception {
                BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));

                System.out.print("Divisor: ");
                String resp = br.readLine();
                teste t = new teste(Integer.parseInt(resp));
                System.out.println(t.executa());
        }
}
```

Saída
:

```
Divisor: 2
1.571

Divisor: 0
Exception in thread "main"
java.lang.Exception: Erro: O divisor não
pode ser zero!
        at teste.<init>(teste.java:13)
        at teste.main(teste.java:26)
```

Mas ainda não está tratando nada...

```
import ...

public class teste {
        public static final double Pi = 3.142;
        private int divisor;

        teste(int div) throws Exception {
                if (div ==0)
                        throw new Exception("Erro: O divisor não pode ser zero!");
                this.divisor = div;
        }

        public double executa() {
                return(Pi/divisor);
        }

        public static void main(String[] args) throws Exception {
                BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

                boolean saida = false;
                while (!saida) {
                        System.out.print("Divisor: ");
                        String resp = br.readLine();
                        try {
                                teste t = new teste(Integer.parseInt(resp));
                                System.out.println(t.executa());
                        }
                        catch(Exception e) {
                                System.out.println("Tente outra vez");
                                saida = false;
                        }
                }
        }
}
```

Protegemos todo o código sensível dentro de um bloco try

Qualquer exceção nesse código é tratada pelo bloco catch

```
import ...

public class teste {
        public static final double Pi = 3.142;
        private int divisor;

        teste(int div) throws Exception {
                if (div ==0)
                        throw new Exception("Erro: O divisor não pode ser zero!");
                this.divisor = div;
        }

        public double executa() {
                return(Pi/divisor);
        }

        public static void main(String[] args) throws Exception {
                BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

                boolean saida = false;
                while (!saida) {
                        System.out.print("Divisor: ");
                        String resp = br.readLine();
                        try {
                                teste t = new teste(Integer.parseInt(resp));
                                System.out.println(t.executa());
                        }
                        catch(Exception e) {
                                System.out.println("Tente outra vez");
                                saida = false;
                        }
                }
        }
}
```

Saída:

```
Divisor: 0
Tente outra vez
Divisor: 0
Tente outra vez
Divisor: 2
1.571
```

```
import ...

public class teste {
        public static final double Pi = 3.142;
        private int divisor;

        teste(int div) throws Exception { ... }

        public double executa() {
                return(Pi/divisor);
        }

        public static void main(String[] args) throws Exception {
                BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

                boolean saida = false;
                while (!saida) {
                        System.out.print("Divisor: ");
                        String resp = br.readLine();
                        try {
                                teste t = new teste(Integer.parseInt(resp));
                                System.out.println(t.executa());
                                saida = true;
                        }
                        catch(Exception e) {
                                System.out.println("Tente outra vez");
                        }
                        finally {
                                System.out.println("Fazendo alguma limpeza necessária");
                        }
                }
        }
}
```

O que vai acontecer?

```
import ...

public class teste {
        public static final double Pi = 3.142;
        private int divisor;

        teste(int div) throws Exception { ... }

        public double executa() {
                return(Pi/divisor);
        }

        public static void main(String[] args)  throws Exception {
                BufferedReader br =
                        new BufferedReader(new
InputStreamReader(System.in));

                boolean saida = false;
                while (!saida) {
                        System.out.print("Divisor:  ");
                        String resp = br.readLine();
                        try {
                                teste t = new
teste(Integer.parseInt(resp));
                                System.out.println(t.executa());
                                saida = true;
                        }
                        catch(Exception e) {
                                System.out.println("Tente  outra vez");
                        }
                        finally {
                                System.out.println("Fazendo  alguma
limpeza necessária");
                        }
```

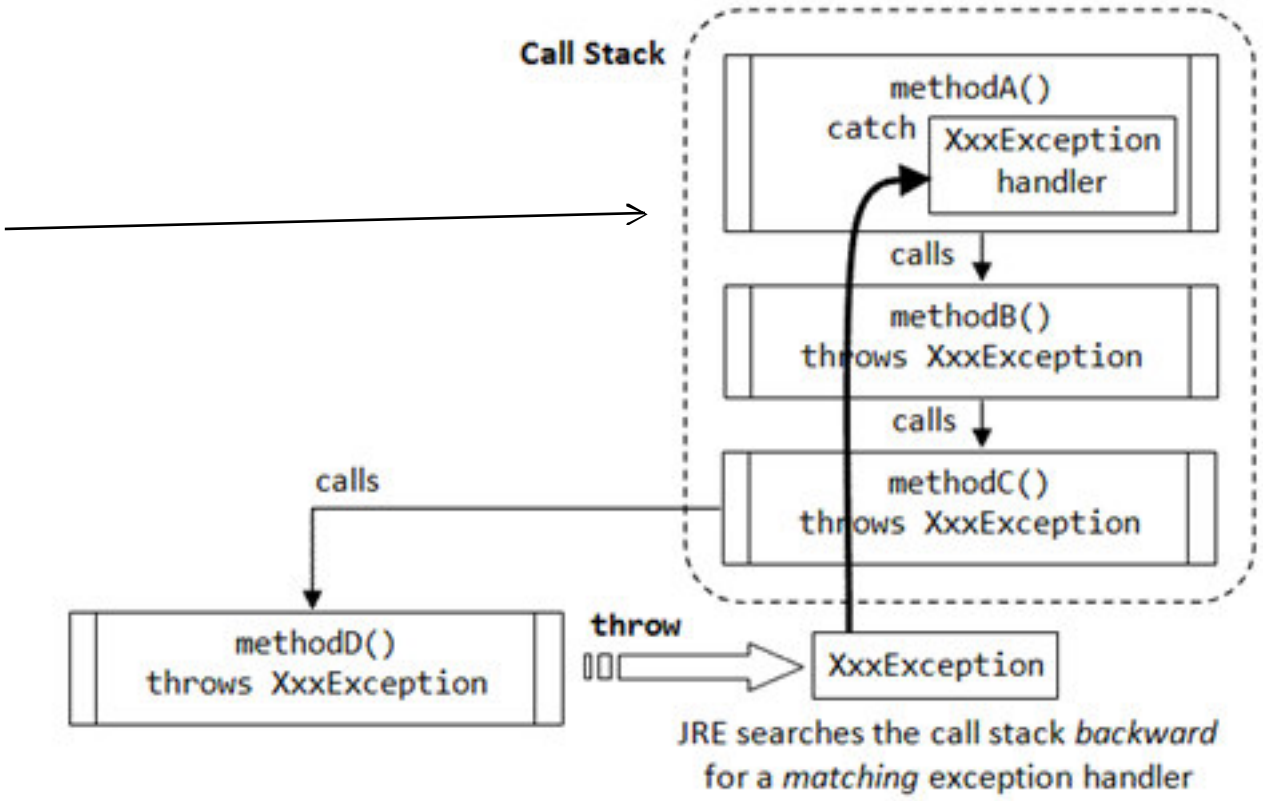
```
Divisor: 2
1.571
Fazendo alguma limpeza
necessária

Divisor: 0
Tente outra vez
Fazendo alguma limpeza
necessária

Divisor: 0
Tente outra vez
Fazendo alguma limpeza
necessária

Divisor: 1
3.142
Fazendo alguma limpeza
necessária
```

Observações

- Uma vez definido que um método lança exceção, toda vez que este método for chamado:
  - A exceção deve ser tratada; ou
  - O método de onde este foi chamado deve repassar a exceção (via ***throws*** em seu cabeçalho).
- E se não fizer?
  - Erro de compilação:
    - unreported exception java.io.IOException; must be caught or declared to be thrown

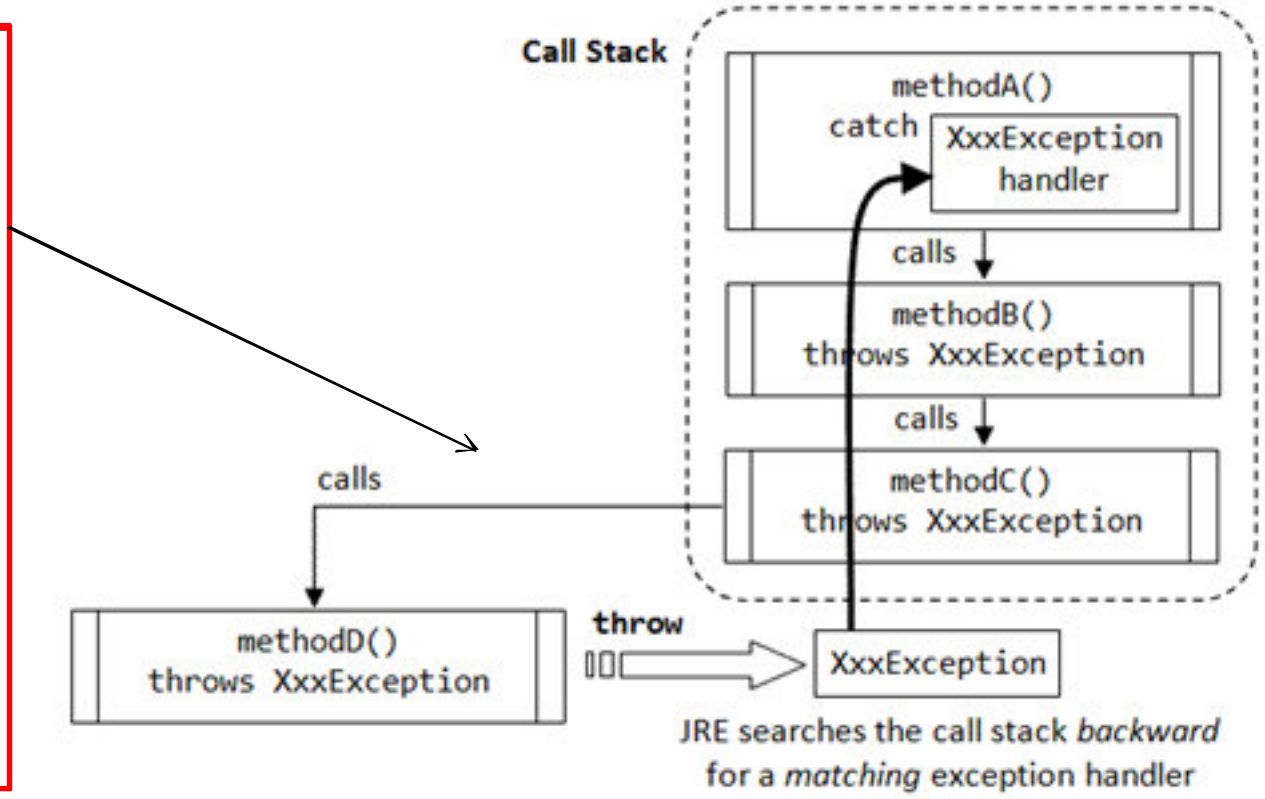
Pilha de Chamadas

À medida em que métodos são chamados dentro de outros métodos, o JRE armazena todos eles em uma estrutura chamada Pilha de Chamadas



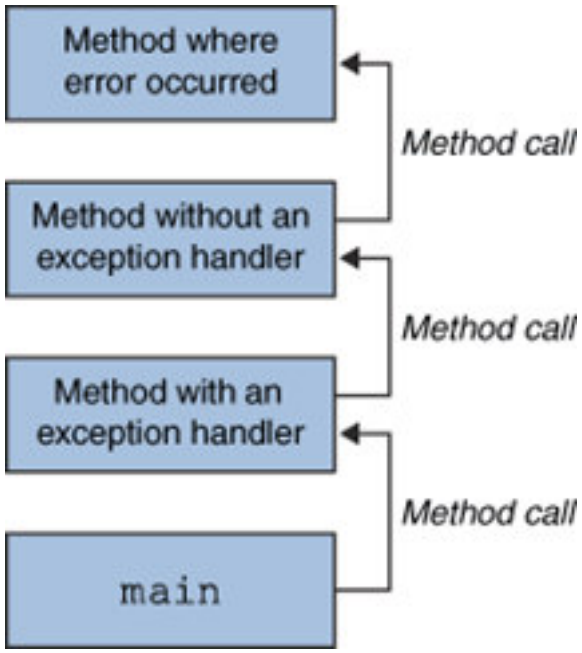
Pilha de Chamadas

Se um determinado método lança uma exceção, o JRE busca seu tratamento na pilha, indo desde o método que chamou o método que lançou a exceção, até atingir main, se nenhum método no caminho tratar da exceção.

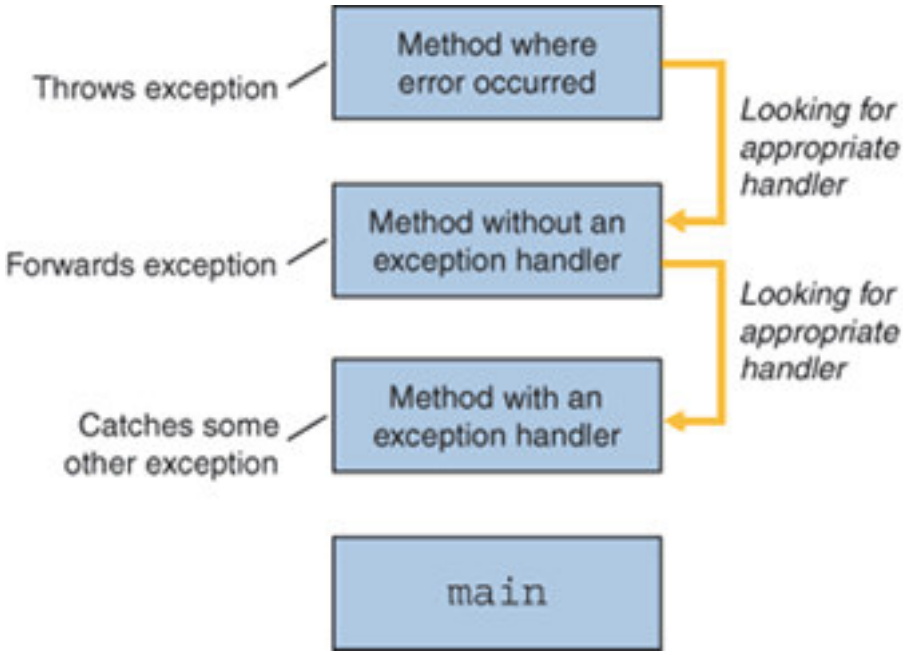


Pilha de Chamadas

Chamada



Busca por tratamento da Exceção



Como podemos ver a pilha de execução?
printStachTrace

```
public static void main(String[] args) throws Exception {
                BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));

                boolean saida = false;
                while (!saida) {
                        System.out.print("Divisor: ");
                        String resp = br.readLine();
                        try {
                                teste t = new
teste(Integer.parseInt(resp));

        System.out.println(t.executa());
                                saida = true;
                        }
                        catch(Exception e) {
                                e.printStackTrace();
                        }
                        finally {
                                System.out.println("Fazendo
alguma limpeza necessária");
                        }
                }
        }
```

```
java.lang.Exception: Erro: O divisor
não pode ser zero!
        at teste.<init>(teste.java:13)
        at teste.main(teste.java:29)
```

Pilha de Chamadas

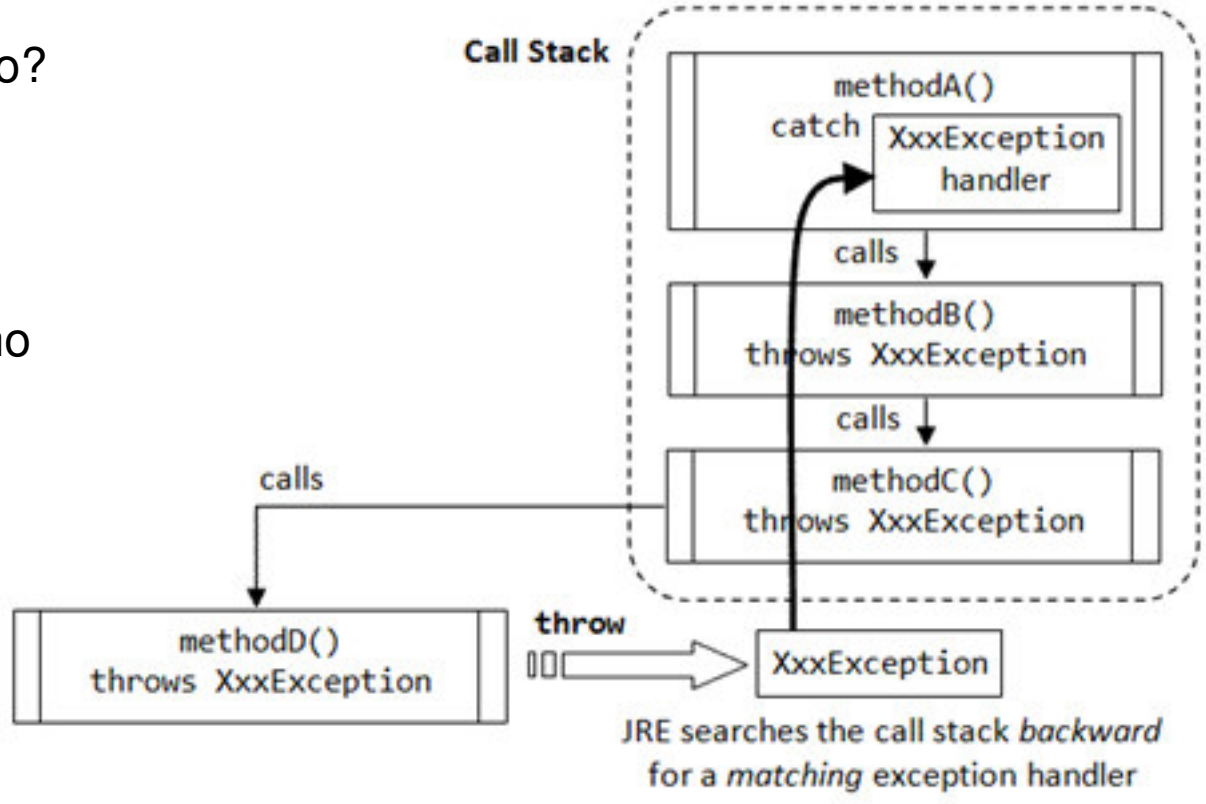
Como tratamos uma exceção?

Catch

Como repassar?

Incluindo “throws” no cabeçalho do método

Já vimos isso!!



```
import ...

public class teste {
        public static final double Pi = 3.142;
        private int divisor;

        teste(int div) throws Exception { ... }

        public double executa() {
                return(Pi/divisor);
        }

        public static void main(String[] args)  throws Exception {
                BufferedReader br =
                        new BufferedReader(new InputStreamReader(System.in));

                boolean saida = false;
                while (!saida) {
                        System.out.print("Divisor: ");
                        String resp = br.readLine();
                        try {
                                teste t = new teste(Integer.parseInt(resp));
                                System.out.println(t.executa());
                                saida = true;
                        }
                        catch(Exception e) {
                                System.out.println("Tente outra vez");
                        }
                        finally {
                                System.out.println("Fazendo alguma limpeza necessária");
                        }
                }
}
```

Mas então, se ou capturamos ou repassamos, e aqui eu capturei, por que
esse throws???

VAMOS REMOVER

```
import ...

public class teste {
        public static final double Pi = 3.142;
        private int divisor;

        teste(int div) throws Exception { ... }

        public double executa() {
                return(Pi/divisor);
        }

        public static void main(String[] args)  {
                BufferedReader br =
                        new BufferedReader(new
InputStreamReader(System.in));

                boolean saida = false;
                while (!saida) {
                        System.out.print("Divisor:  ");
                        String resp = br.readLine();
                        try {
                                teste t = new
teste(Integer.parseInt(resp));
                                System.out.println(t.executa());
                                saida = true;
                        }
                        catch(Exception e) {
                                System.out.println("Tente  outra
vez");
                        }
                        finally {
                                System.out.println("Fazendo
alguma limpeza necessária");
                        }
```

Erro de compilação:
unreported exception java.io.IOException; must be caught or declared to be thrown

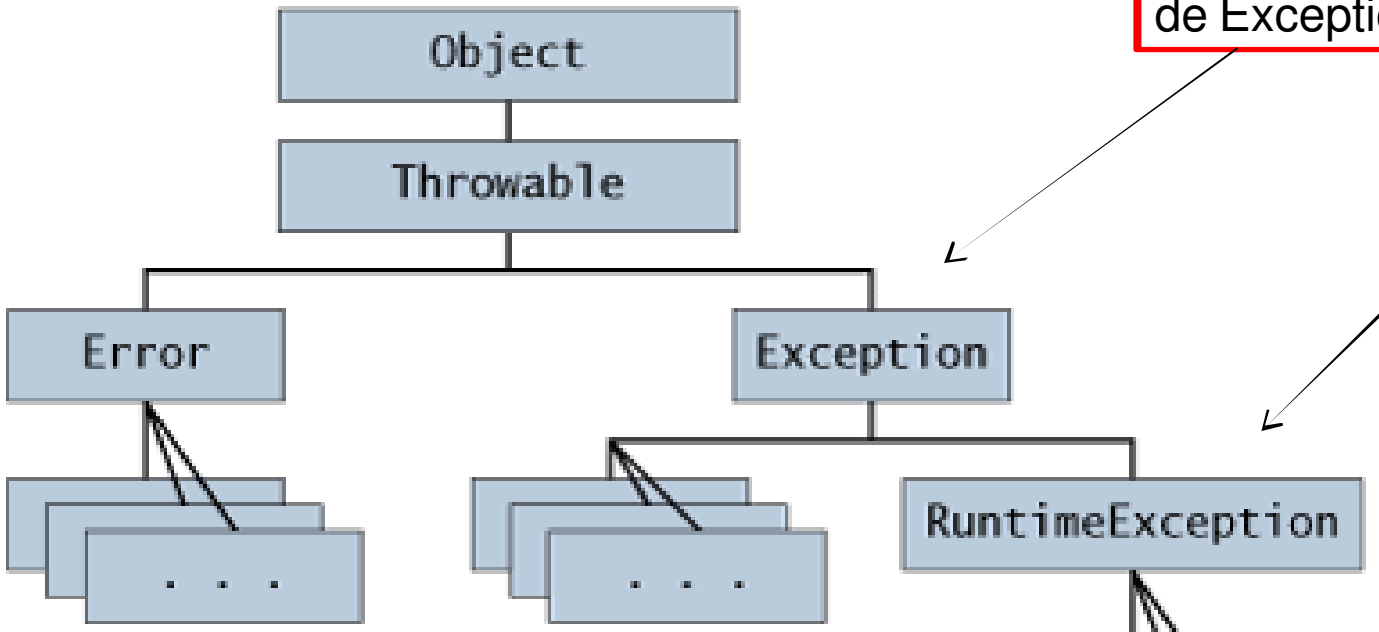
Mas e Exception não deveria funcionar, já que engloba IOException?
E o código sensível, está dentro do try?
Exceções somente são capturadas se o código que as gera está em um bloco try.

IOException ?
Mais tarde...

```
Import ...
import java.io.IOException;

public class teste {
        public static final double Pi = 3.142;
        private int divisor;

        teste(int div) throws Exception { ... }

        public double executa() {
                return(Pi/divisor);
        }

        public static void main(String[] args) throws IOException {
                BufferedReader br =
                        new BufferedReader(new
InputStreamReader(System.in));

                boolean saida = false;
                while (!saida) {
                        System.out.print("Divisor: ");
                        String resp = br.readLine();
                        try {
                                teste t = new
teste(Integer.parseInt(resp));
                                System.out.println(t.executa());
                                saida = true;
                        }
                        catch(Exception e) {
                                System.out.println("Tente outra vez");
                        }
                        finally {
                                System.out.println("Fazendo alguma
limpeza necessária");
                        }
```

É boa política deixar explícito o tipo de exceção usada.

Hierarquia de Exceções



Nossa exceção (IOException é um tipo de Exception)

??? Mais adiante...

Problema sério que uma aplicação não deveria tentar tratar

Exemplos de Exceção

- Existem vários exemplos de exceção
  - java.io.IOException
  - java.sql.SQLException
  - javax.management.modelmbean.XMLParseException
  - java.lang.ClassNotFoundException
  - java.lang.ArithmeticException
  - ...

Construindo uma Exceção

- Todos os exemplos são subclasses de Exception:

```
class MinhaExcecao extends Exception
{
        public MinhaExcecao() {}
        public MinhaExcecao(String msg){
                super(msg);
        }
}
```

Poderíamos construir nossa exceção assim.

```
import ...

public class teste {
        ...

        teste(int div) throws MinhaExcecao {
                if (div ==0)
                        throw new MinhaExcecao("Erro: O divisor não pode ser
zero!");
                this.divisor = div;
        }

        ...

        public static void main(String[] args) throws IOException {
                BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));

                boolean saida = false;
                while (!saida) {
                        System.out.print("Divisor: ");
                        String resp = br.readLine();
                        try {
                                teste t = new teste(Integer.parseInt(resp));
                                System.out.println(t.executa());
                                saida = true;
                        }
                        catch(MinhaExcecao e) {
                                System.out.println("Tente outra vez");
                        }
                }
```

Não precisa incluir no import, pois está no mesmo pacote (diretório)

Saída:

Divisor: 2
1.571

Divisor: 0
Tente outra vez
Divisor: 1
3.142

Múltiplos Blocos Catch

- A existência de múltiplas exceções abre para a possibilidade de múltiplos blocos catch.
- Criamos um para cada tipo de exceção:
  - Com código específico para cada exceção
- Nos permite tomar decisões diferentes conforme o tipo de erro encontrado.

```
public static void main(String[] args) {
                boolean saida = false;
                while (!saida) {
                        System.out.print("Divisor: ");
                        try {
                                BufferedReader br = new
BufferedReader(new InputStreamReader(System.in));
                                String resp = br.readLine();
                                teste t = new
teste(Integer.parseInt(resp));

        System.out.println(t.executa());
                                saida = true;
                        }
                        catch(MinhaExcecao e) {
                                System.out.println("Tente
outra vez");
                        }
                        catch(IOException ioe) {
                                System.out.println("Erro de
E/S.");
                        }
                }
```

Não precisa mais dizer que o método lança a exceção, pois ela foi capturada.

Contudo, para podermos tratar a exceção, tivemos que mover o código que poderia gerá-la para dentro do bloco try.

Se isso não for possível, basta criar um outro try, em outra porção do código. Não há limites para seu uso.

```
Import ...

public class teste {
        public static final double Pi = 3.142;
        private int divisor;

        teste(int div) throws MinhaExcecao {
                if (div ==0)
                        throw new MinhaExcecao("Erro: O divisor não pode ser
zero!");
                this.divisor = div;
        }

        public double executa() {
                return(Pi/divisor);
        }

        public static void main(String[] args) throws IOException, MinhaExcecao {
                BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));

                System.out.print("Divisor: ");
                String resp = br.readLine();
                teste t = new teste(Integer.parseInt(resp));
                System.out.println(t.executa());
        }
```

Da mesma forma que podemos tratar de múltiplas exceções, também podemos especificar quais serão repassadas:

Para Finalizar

- Há dois tipos básicos de exceção
  - Checked:
    - Devem ser explicitamente capturadas ou propagadas
    - Estendem java.lang.Exception
  - Unchecked (ou Run-Time):
    - Não precisam ser capturadas ou propagadas
    - Passam "despercebidas" pelo compilador
    - Estendem java.lang.RuntimeException
    - MUITO CUIDADO!
  - Nesse curso, veremos apenas as Checked

```
public class teste {
        public static final double Pi = 3.142;
        private int divisor;

        teste(int div) throws MinhaExcecao {
                if (div ==0)
                        throw new MinhaExcecao("Erro: O divisor não pode ser
zero!");
                this.divisor = div;
        }

        public double executa() {
                return(Pi/divisor);
        }

        public static void main(String[] args) throws IOException {
                BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));

                System.out.print("Divisor: ");
                String resp = br.readLine();
                teste t = new teste(Integer.parseInt(resp));
                System.out.println(t.executa());
        }
}
```

```
import java.lang.RuntimeException;

class MinhaExcecao extends RuntimeException {
        public MinhaExcecao() {}
        public MinhaExcecao(String msg) {
                super(msg);
        }
}
```

Não precisa declarar...

Unchecked

```
public static void main(String[] args) throws IOException  {
                BufferedReader br =
                        new BufferedReader(new
InputStreamReader(System.in));

                boolean saida = false;
                while (!saida) {
                        System.out.print("Divisor: ");
                        String resp = br.readLine();
                        try {
                                teste t = new
teste(Integer.parseInt(resp));
                                System.out.println(t.executa());
                                saida = true;
                        }
                        catch(MinhaExcecao e) {
                                System.out.println("Tente outra
vez");
                        }
        }
}
```

Embora não precise, elas podem ser capturadas

Unchecked

- Vantagens:
  - Em uma hierarquia de chamadas longa, evita que blocos try – catch sejam usados com frequência
  - Poupa tempo do programador
- Desvantagens:
  - O programador só descobre que elas existem quando há um erro que para o programa
  - Sem saber que ela existe, ele não a captura

Políticas

- Use Checked para condições das quais se possa recuperar:
  - Ex: um arquivo que não existe, erros do usuário
  - O programador é forçado a saber que a exceção existe
  - Cabe a ele ignorá-la, conscientemente, embora seja melhor tratar
- Use Unchecked para condições que indiquem erro do programador
  - Ex: colocar elementos demais em um array
  - Coisas que o programador não devia fazer

Javadoc
Motivação

Javadoc
Motivação

- Você desenvolveu uma biblioteca de classes e precisa distribuí-la
- Contudo
  - Não é de seu interesse que o usuário tenha acesso aos detalhes internos da classe
  - O usuário deve ter acesso a informação suficiente para que possa fazer uso da biblioteca
- O que fazer?

Solução Inicial

- Construir um manual detalhado, descrevendo como funciona cada método público da classe
- Problemas:
  - Para evitar confusão, você já comentou todo o código interno → há trabalho duplicado.
  - Quaisquer mudanças feitas no código devem ser refletidas no documento
    - Podem ocorrer erros no meio do caminho

Solução Melhor

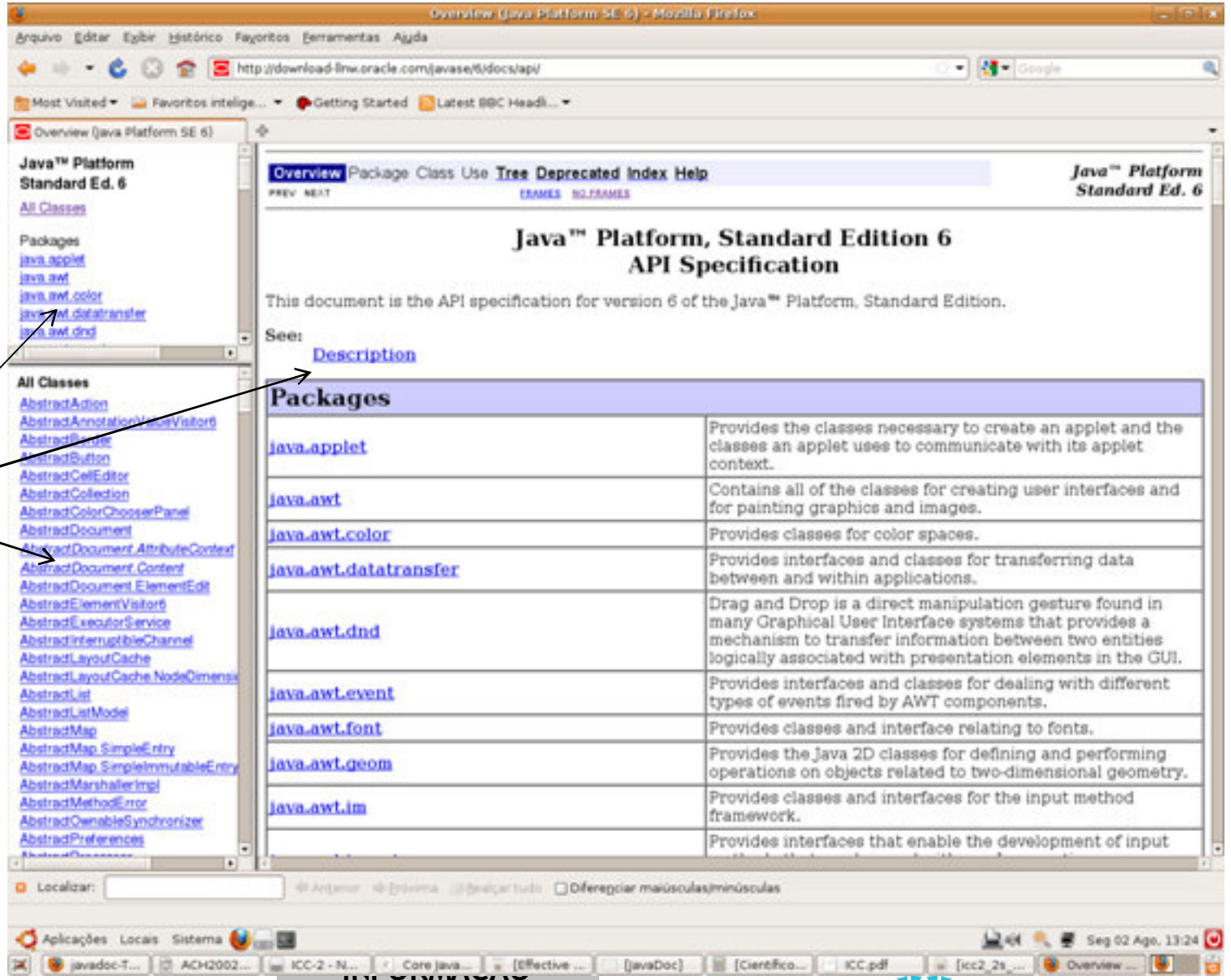
- Ter um modo de criar a documentação para o usuário (da API), diretamente a partir da documentação feita para os mantenedores do código
  - Criar um manual para o usuário a partir dos comentários feitos para os programadores da classe
- Javadoc

Javadoc

- Ferramenta que transforma comentários em código Java em páginas HTML, contendo:
  - Classes públicas e privadas
  - Classes internas
  - Construtores
  - Interfaces
  - Métodos e Campos
- Processa apenas as partes públicas do código

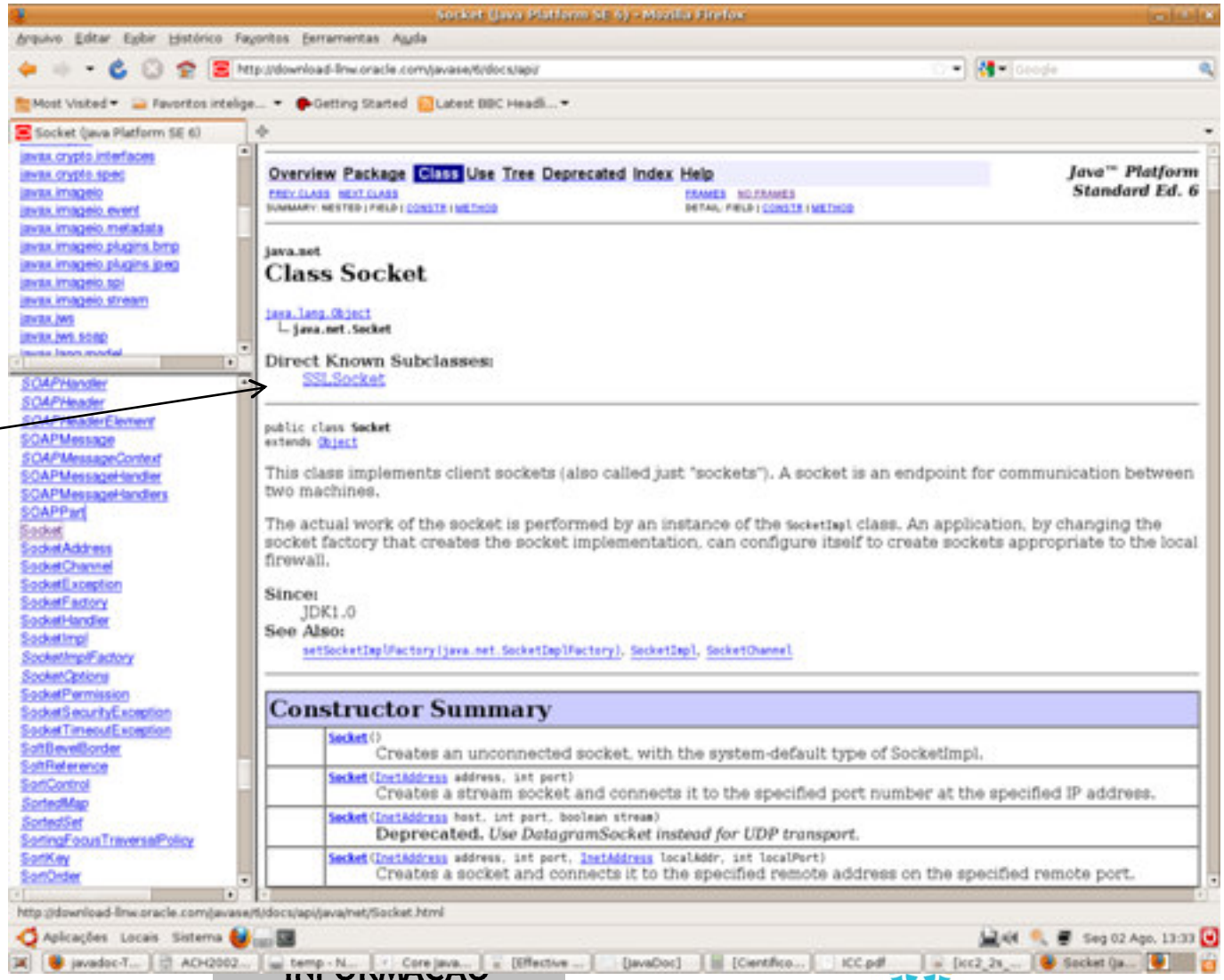
Javadoc

Visão geral da estrutura de classes e pacotes



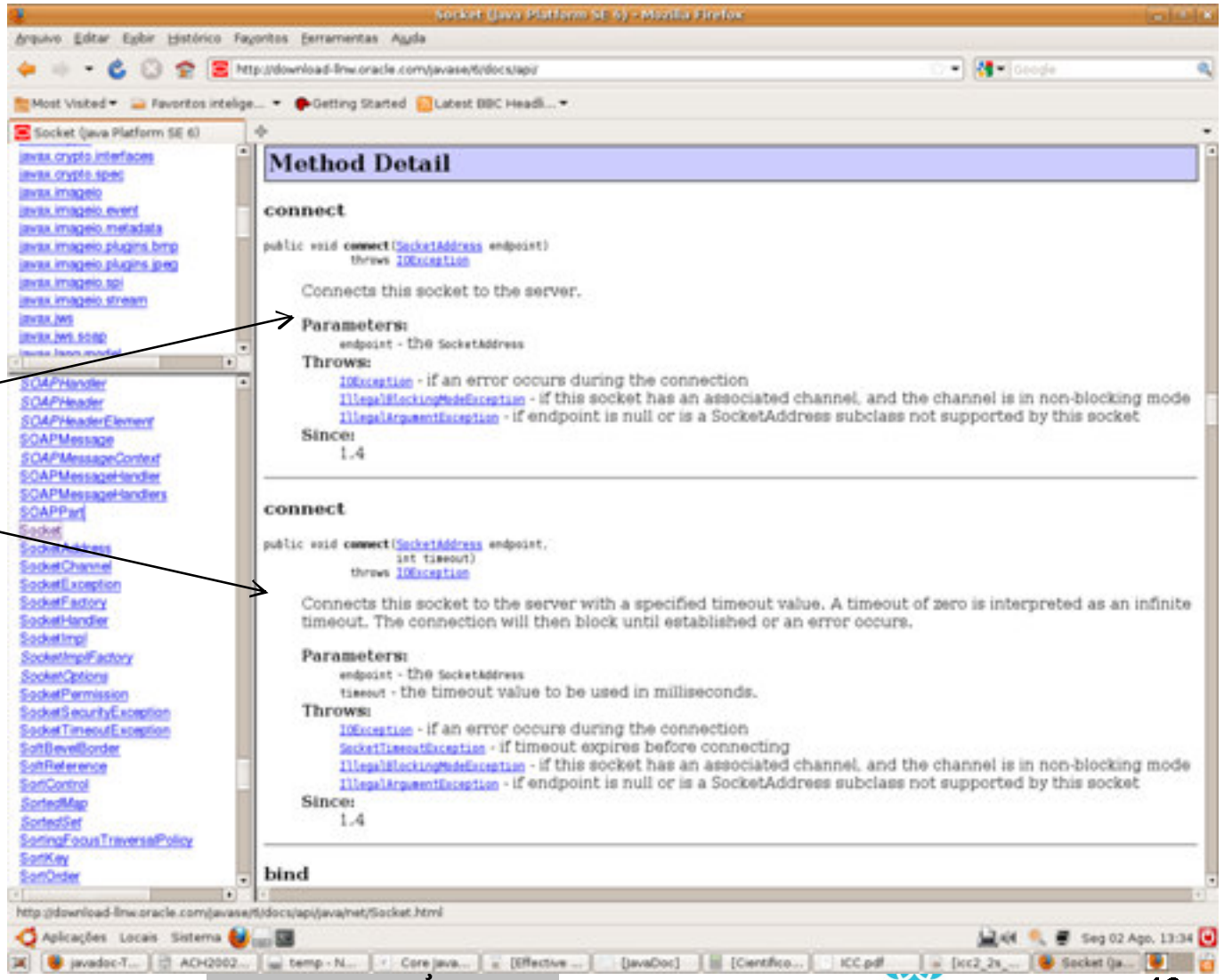
Javadoc

Cabeçalho e informações iniciais de cada classe



Javadoc

Informação sobre cada método, com breve descrição, parâmetros etc



Inserindo Comentários

- Comentários javadoc são inseridos de forma natural no código
  - Todo texto entre /** e */ é processado
  - Deve estar no cabeçalho de métodos e classes
  - Também pode estar no comentário associado a campos
- Existem códigos especiais para definição de autor, parâmetros, versão etc

Códigos Javadoc

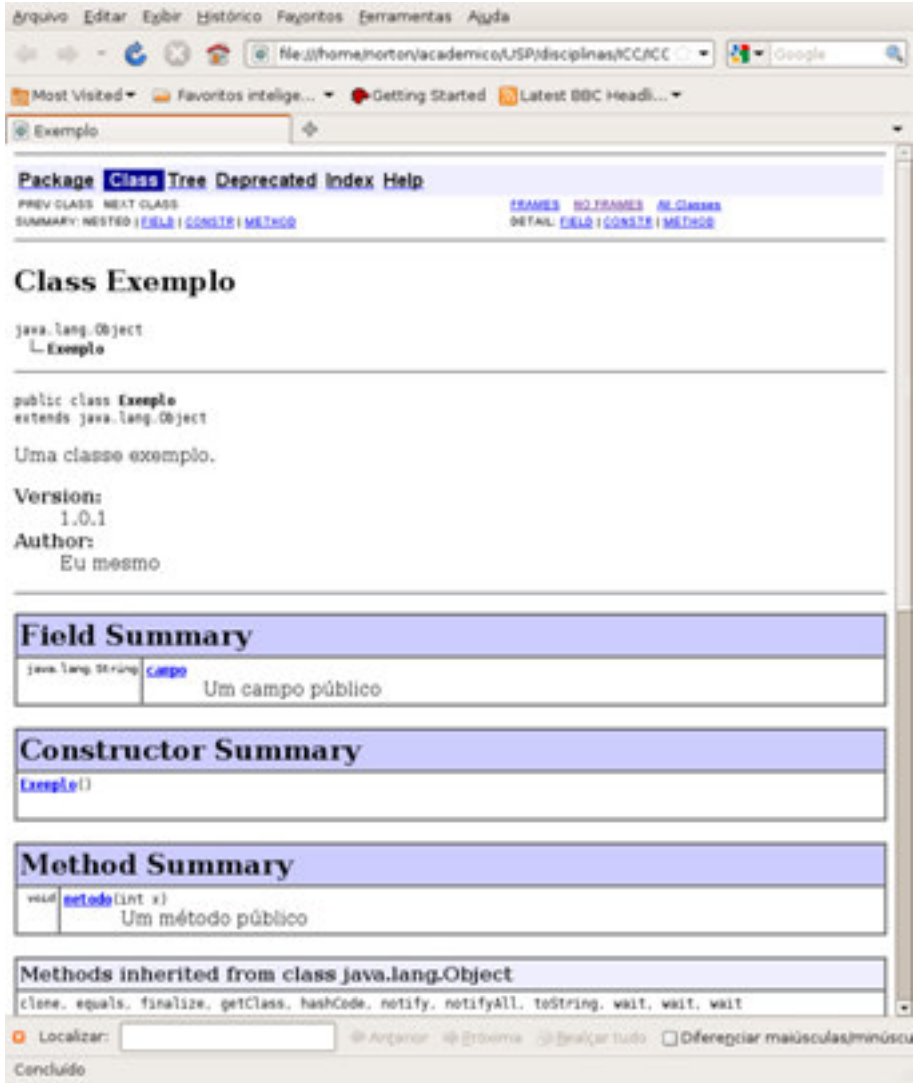
- Parâmetros:
  - @param variável descrição
- Valor de retorno:
  - @return descrição
- Exceções lançadas:
  - @throws classe (do tipo exceção) descrição
- Autor
  - @author nome do autor

Códigos Javadoc

- Versão da classe, pacote etc
  - @version texto sobre a versão
- Indicando a partir de que versão algo está disponível
  - @since texto explicando a partir de qual versão usar um método ou classe
- Ligando javadocs diferentes
  - @see classe relacionada
  - Ex: @see java.lang.Object

Exemplo

```
/**
        Uma classe exemplo.

        @author Eu mesmo
        @version 1.0.1
*/
public class Exemplo implements IExemplo {
        // um campo privado
        private double x;

        /** Um campo público */
        public String campo;

        /**
                Um método público

                @param x Um parâmetro
                @return o dobro do parâmetro de
entrda
        */
        public void metodo(int x) {
                return(2*x);
        }
}
```



Exemplo

Criado com:

javadoc -version -author -d doc ex_javadoc.java

-author inclui informação @author

-version inclui informação @version

-d <dir> diretório onde os arquivos javadoc estarão (html e css)

Outras Possibilidades

- Incluir a parte privada:
  - Basta inserir os comentários no formato javadoc
  - Executar:
    - javdoc -version -author -private -d doc *.java
- Enfeitar o texto:
  - Marcações html (como negrito, itálico, tamanhos diferentes de letra etc) também são aceitos pelo javadoc

Fontes

- http://download.oracle.com/javase/1.3/docs/ tooldocs/win32/javadoc.html
- Kon, F.; Goldman, A.; Silva, P.J.S. "Introdução à Ciência de Computação com Java e Orientado a Objetos", IME - USP, 2004.
- http://tutorials.jenkov.com/java-exception-handling/basic-try-catch-finally.html
- http://download.oracle.com/javase/tutorial/essential/exceptions/index.html
- Horstmann, C.S.; Cornell, G.: Core Java 2: Volume I - Fundamentals. Prentice Hall. 2002.
- http://www3.ntu.edu.sg/home/ehchua/programming/java/J5a_Exception.html
- Bloch, J.: Effective Java: Programming Language Guide. Addison-Wesley. 2001,

Tratamento de Erro e Javadoc

Professores:

Norton T. Roman

Fátima L.S.Nunes