



Java Standard Edition (JSE)

10. Controle de Exceções



Esp. Márcio Palheta

Gtalk: marcio.palheta@gmail.com



Ementa

- 10. Controle de Exceções;
- 11. O pacote java.lang;
- 12. O pacote java.io;
- 13. Collection Framework e Generics;
- 14. Threads;
- 15. Sockets;
- 16. Tipos especiais: Enum;
- 17. Ferramentas jar e javadoc;



Agenda

- Introdução ao controle de exceções;
- Tipos de controle;
- Pilha de execução;
- Tratamento de exceções;
- Tipos de exceção: Checked e Unchecked
- Lançamento de exceções;
- Construtores de exceções;
- Exceções especializadas;



Cenário 01 – Método sacar()

- Imaginemos a classe ContaBancaria e seu método sacar();
- O que acontece quando tentamos sacar um valor maior que o nosso limite?

```
public boolean sacar(double valor){  
    if(valor > this.saldo){  
        System.out.println("Saque fora do limite");  
        return false;  
    }else{  
        this.saldo -= valor;  
        return true;  
    }  
}
```



Operação de saque

- O sistema mostra uma mensagem de erro(**no console**)
- mas o que acontece com quem chamou o método **sacar()** ?
- Como avisar que a operação **NÃO foi realizada** com sucesso?
- Se o saque não for realizado, o usuário deve ser avisado do problema;



Chamada válida ou inválida?

- O que acontece no trecho de código a seguir? Algum problema?
- Sem conhecer o método sacar(), você afirmaria que ele foi executado com sucesso?

```
public static void main(String[] args) {  
    ContaBancaria minhaConta = new ContaBancaria();  
    minhaConta.depositar(300);  
    minhaConta.sacar(500);  
}
```



Como melhorar a comunicação entre as classes?

- Em sistemas reais, é muito comum que quem saiba tratar o erro é aquele que chamou o método `sacar()`;
- A solução tradicional é retornar um valor booleano, a fim de informar que houve um erro no processamento;
- Com isso, a responsabilidade de tratar o erro que ocorreu passa para quem chama o método `sacar()`;



Melhorando a chamada

- Que tal a nova chamada?

```
public static void main(String[] args) {  
    ContaBancaria minhaConta = new ContaBancaria();  
    minhaConta.depositar(300);  
    if(!minhaConta.sacar(500)){  
        System.out.println("O saque não foi realizado");  
    }  
}
```

- **Ops!** Temos que lembrar de tratar o retorno do método. Isso é bom?
- E se o caixa eletrônico “esquecesse” de testar o tipo de retorno?



E os outros erros?

- Mesmo que estejamos tratando o tipo de retorno do método sacar, o que aconteceria se tentássemos sacar um valor negativo?
- A solução instantânea é mudar o tipo de retorno de boolean para int;
- Mas...qual seria a consequência de usar os **magic numbers**?

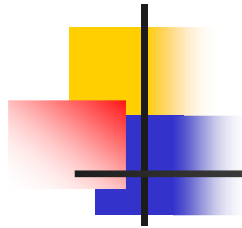


Como tratar os erros?

- Precisamos de uma forma de tratar “situações especiais” que possam gerar erros;
- Daí, surge o conceito de EXCEÇÃO;
- **Exceção** representa uma situação que normalmente não, sendo algo de estranho ou inesperado no sistema.

Cenário 02 – execução de vários métodos

```
public class TesteErro {  
    static void metodo1() {  
        System.out.println("inicio do metodo1");  
        metodo2();  
        System.out.println("fim do metodo1");  
    }  
    static void metodo2() {  
        System.out.println("inicio do metodo2");  
        int[] array = new int[5];  
        for (int i = 0; i <= 10; i++) {  
            array[i] = i;  
            System.out.println(i);  
        }  
        System.out.println("fim do metodo2");  
    }  
    public static void main(String[] args) {  
        System.out.println("inicio do main");  
        metodo1();  
        System.out.println("fim do main");  
    }  
}
```



Entendendo as chamadas

- O método `main` chama `metodo1` que chama o `metodo2`;
- O que ocorre com as variáveis de cada método?
- Como a execução de cada método é controlada de forma separada?
- Como fica a memória?

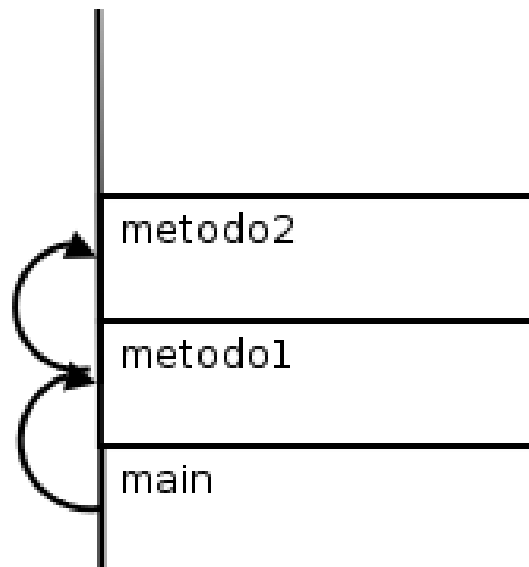


Pilha de execução (stack)

- Toda invocação de um método é empilhada em uma estrutura de memória;
- Cada método tem sua própria área;
- Quando um método termina, sai da pilha e a execução retorna para o método que o chamou;

Pilha de execução

- Graficamente, podemos representar a pilha de execução do nosso trecho de código com a figura a seguir:





Mas...nem tudo é perfeito

- O metodo2 possui um problema: está tentando acessar um índice de array indevido;
- o índice estará fora dos limites do array quando chegar em 5;
- Rode o código.
- Qual é a saída? O que isso representa? O que ela indica?



Rastro de pilha - stacktrace

```
inicio do main
inicio do metodo1
inicio do metodo2
0
1
2
3
4
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
    at bean.TesteErro.metodo2(TesteErro.java:13)
    at bean.TesteErro.metodo1(TesteErro.java:6)
    at bean.TesteErro.main(TesteErro.java:20)
```




Exceções em JAVA

- Quando uma exceção é lançada (**throws**), a JVM verifica se o método tomou algum cuidado ao tentar executar esse trecho de código;
- Como metodo2 não trata o erro, a JVM aborta a execução do metodo2 e volta um nível(**stackframe**), retornando à execução do metodo1;



Seguindo na pilha

- Como `metodo1` não trata nenhum problema chamado `ArrayIndexOutOfBoundsException`, a execução de `metodo1` também é encerrada abruptamente;
- A JVM segue na pilha de execução e chega ao método `main`, onde percebe que também não há tratativa para a exceção lançada;
- A aplicação é encerrada de forma inesperada;



Tratamento das exceções

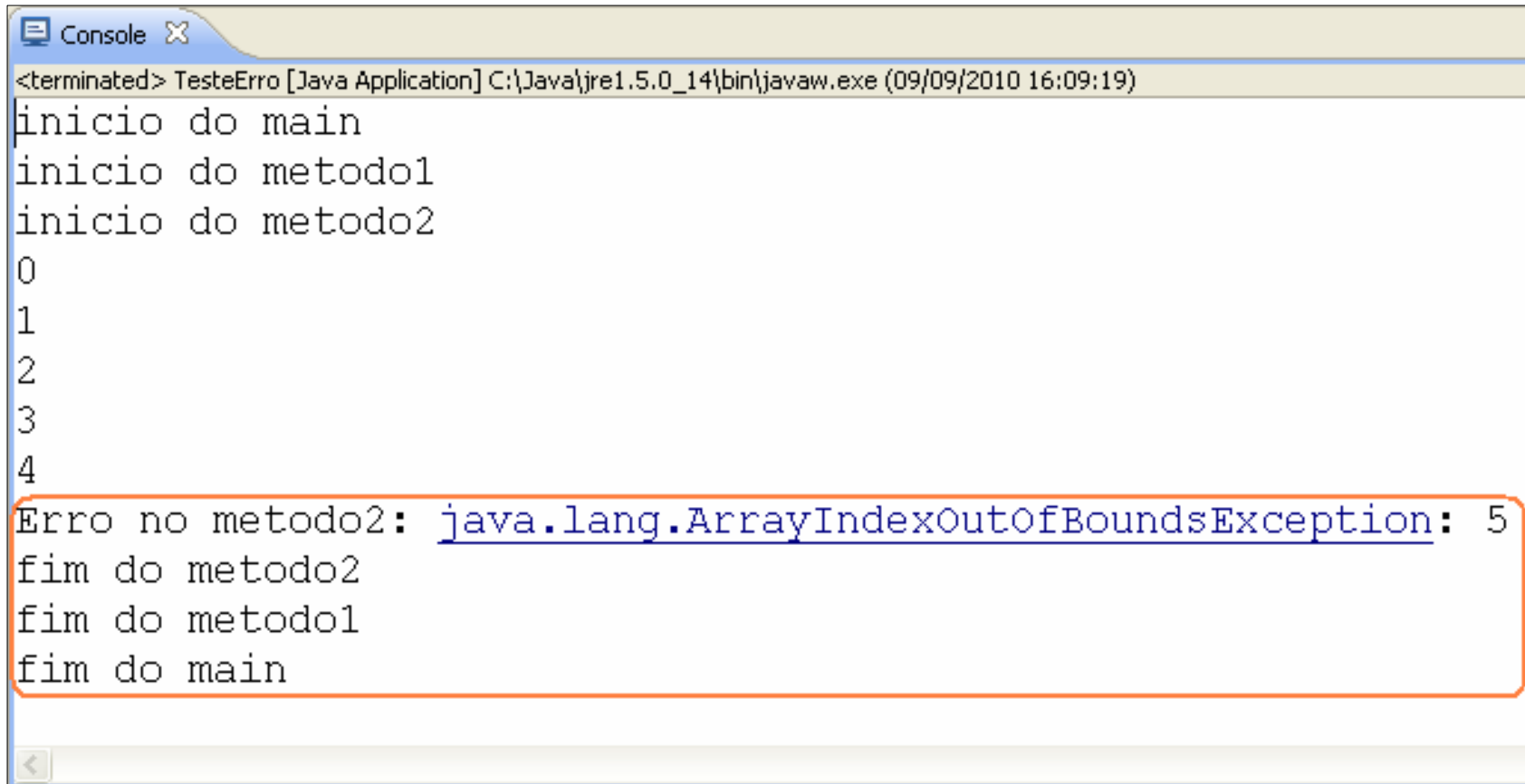
- O erro de acesso indevido ao índice de um array seria resolvido com `length`;
- Porém, a fim de entendermos o controle do fluxo de uma Exceção, vamos tentar(`try`) executar o bloco de comandos e pegar (`catch`) a exceção `ArrayIndexOutOfBoundsException`.



Atualização de metodo2()

```
static void metodo2() {  
    System.out.println("inicio do metodo2");  
    int[] array = new int[5];  
    try {  
        for (int i = 0; i <= 10; i++) {  
            array[i] = i;  
            System.out.println(i);  
        }  
    } catch (ArrayIndexOutOfBoundsException e) {  
        System.out.println("Erro no metodo2"+e);  
    }  
    System.out.println("fim do metodo2");  
}
```

Resultado da execução



```
<terminated> TesteErro [Java Application] C:\Java\jre1.5.0_14\bin\javaw.exe (09/09/2010 16:09:19)
inicio do main
inicio do metodo1
inicio do metodo2
0
1
2
3
4
Erro no metodo2: java.lang.ArrayIndexOutOfBoundsException: 5
fim do metodo2
fim do metodo1
fim do main
```



Itens a considerar

- O que mudou?
- A partir do momento em que a exceção foi tratada(catched), a execução do programa volta ao normal;
- A execução reinicia na linha localizada após o bloco try-catch

Outra exceção de Runtime

```
22 public static void main(String[] args) {  
23     ContaBancaria minhaConta = null;  
24     minhaConta.depositar(300);  
25 }  
26 }
```

Console X

<terminated> ContaBancaria [Java Application] C:\Java\jre1.5.0_14\bin\javaw.exe (09/09/2010 16:46:35)

Exception in thread "main" java.lang.NullPointerException
at bean.ContaBancaria.main(ContaBancaria.java:24)



Exceções não-cheçadas

- Como poderíamos evitar exceções como `ArrayIndexOutOfBoundsException` ou `NullPointerException`?
- Em ambos os casos, o erro poderia ser evitado pelo programador;
- É por isso que você não é obrigado a usar o bloco try-catch;
- Essas exceções são chamadas de `Unchecked`



Unchecked Exceptions

- O compilador não precisa checar se uma determinada exceção está sendo tratada pelo programador;
- Não há problemas em tempo de compilação;
- A exceção só é lançada em tempo de execução;



Outro tipo de exceção

- Em java, temos outro tipo de exceção que obriga que o programador trate as exceções;
- A esse tipo chamamos de **Checked**, porque o compilador checa se você está tratando a possível exceção;
- Caso a exceção não seja tratada, ocorrerá um erro em tempo de compilação;



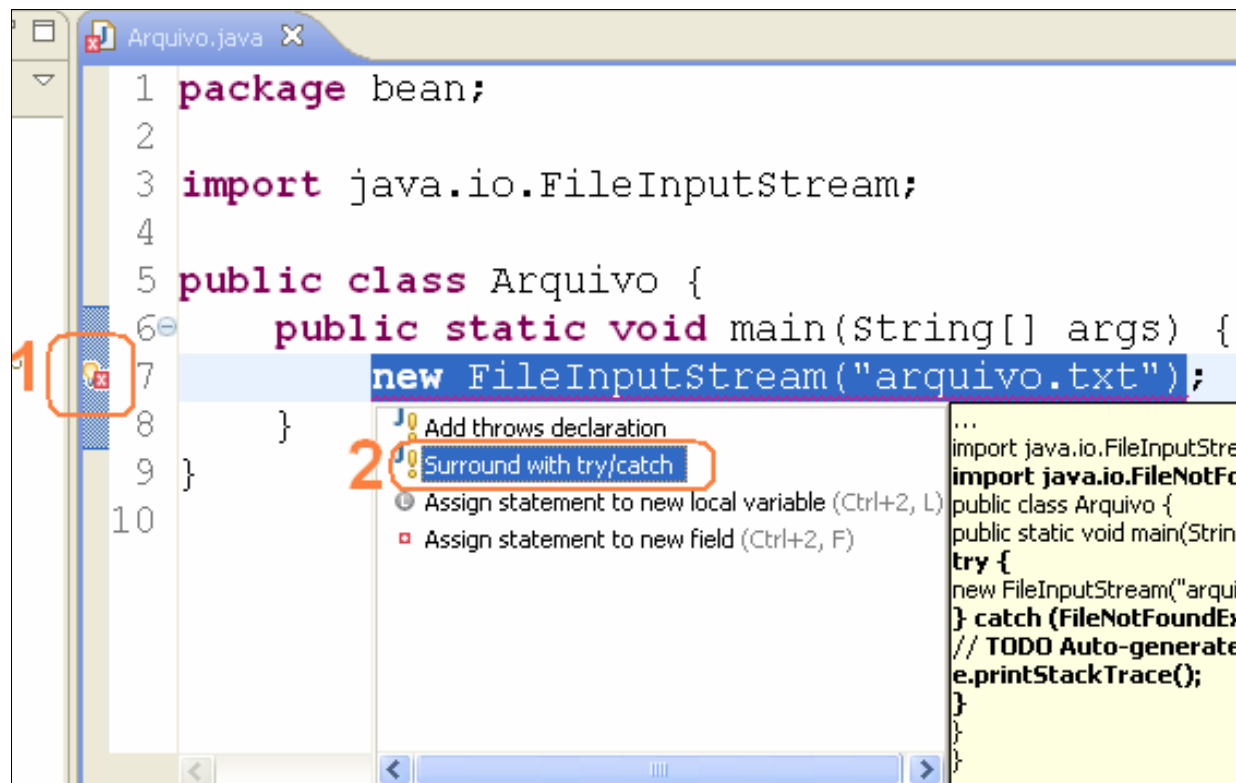
Leitura de arquivos

- O que acontece no código abaixo?
- Por que não compila?
- Como resolver o problema?

```
Arquivo.java x
1 package bean;
2
3 import java.io.FileInputStream;
4
5 public class Arquivo {
6     public static void main(String[] args) {
7         new FileInputStream("arquivo.txt");
8     }
9 }
```

Tratando uma exception

- Clique no sinal de erro e escolha a opção **Surround with try/catch**



```
1 package bean;
2
3 import java.io.FileInputStream;
4
5 public class Arquivo {
6     public static void main(String[] args) {
7         new FileInputStream("arquivo.txt");
8     }
9 }
10
```

1

2

- Add throws declaration
- Surround with try/catch
- Assign statement to new local variable (Ctrl+2, L)
- Assign statement to new field (Ctrl+2, F)

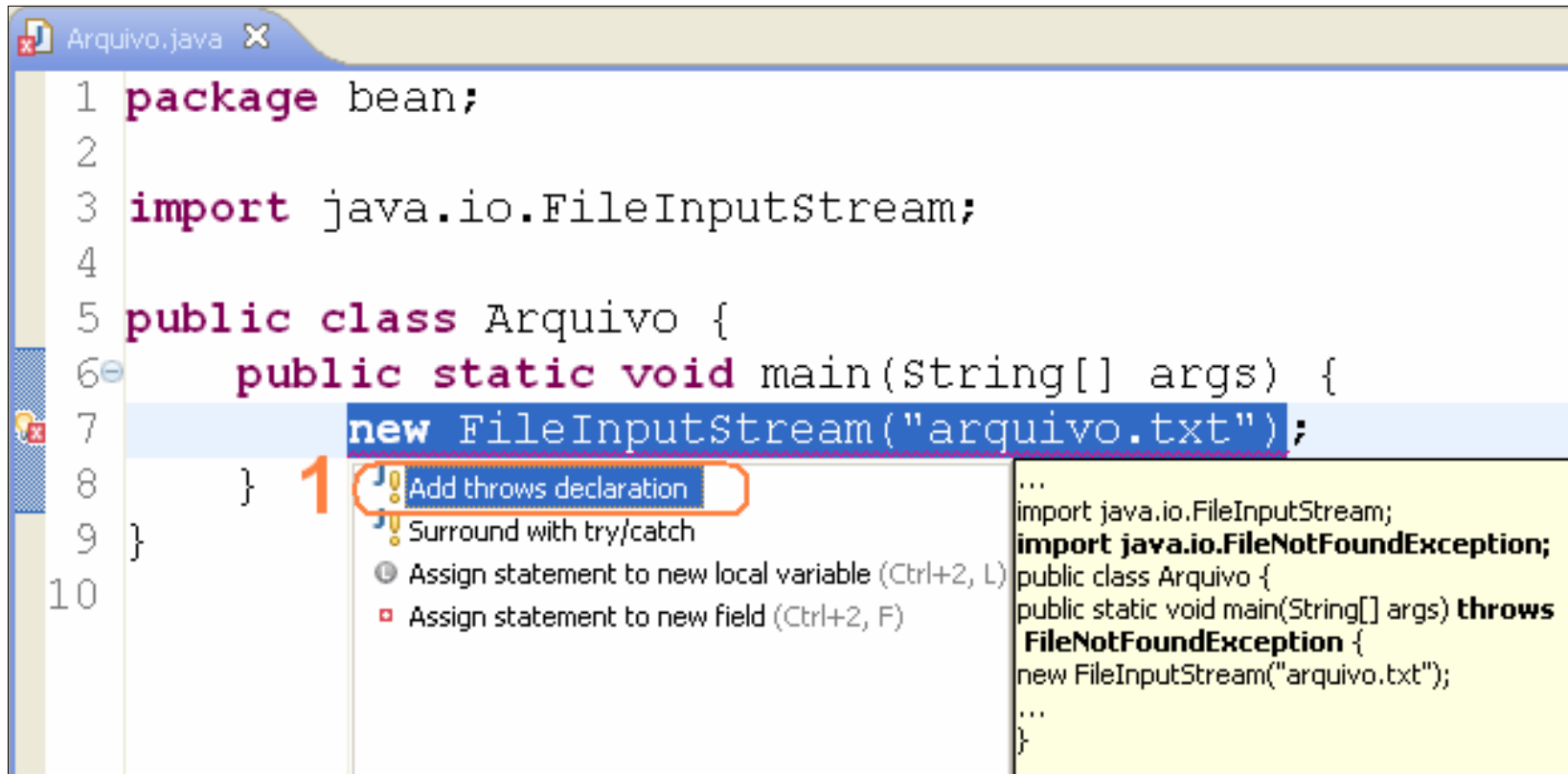
```
...
import java.io.FileInputStream;
import java.io.FileNotFoundException;
public class Arquivo {
    public static void main(String[] args) {
        try {
            new FileInputStream("arquivo.txt");
        } catch (FileNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```



Resultado gerado

```
Arquivo.java X
1 package bean;
2
3 import java.io.FileInputStream;
4 import java.io.FileNotFoundException;
5
6 public class Arquivo {
7     public static void main(String[] args) {
8         try {
9             new FileInputStream("arquivo.txt");
10        } catch (FileNotFoundException e) {
11            System.out.println("Arquivo não encontrado");
12            e.printStackTrace();
13        }
14    }
15 }
```

Outra forma de tratamento - Lançando a exceção



```
1 package bean;
2
3 import java.io.FileInputStream;
4
5 public class Arquivo {
6     public static void main(String[] args) {
7         new FileInputStream("arquivo.txt");
8     }
9 }
10
```

1 Add throws declaration
Surround with try/catch
Assign statement to new local variable (Ctrl+2, L)
Assign statement to new field (Ctrl+2, F)

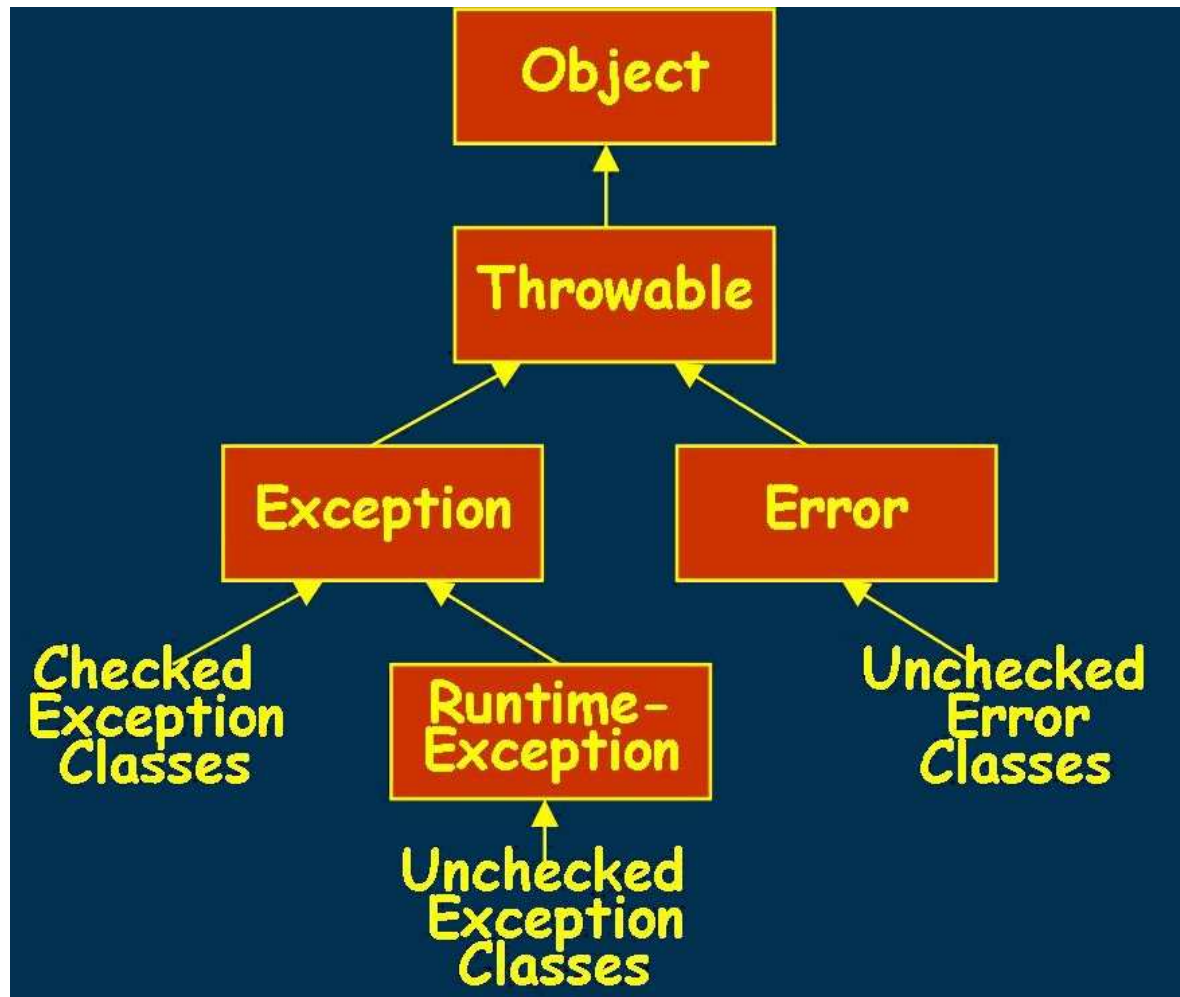
```
...
import java.io.FileInputStream;
import java.io.FileNotFoundException;
public class Arquivo {
    public static void main(String[] args) throws
    FileNotFoundException {
        new FileInputStream("arquivo.txt");
    }
}
```



Resultado gerado

```
Arquivo.java x
1 package bean;
2
3 import java.io.FileInputStream;
4 import java.io.FileNotFoundException;
5
6 public class Arquivo {
7     public static void main(String[] args) throws FileNotFoundException {
8         new FileInputStream("arquivo.txt");
9     }
10 }
```

A família Throwable





Como tratar mais de uma exceção?

1) Com o try e catch:

```
try {  
    objeto.metodoQuePodeLancarIOeSQLException();  
} catch (IOException e) {  
    // ..  
} catch (SQLException e) {  
    // ..  
}
```

2) Com o throws:

```
public void abre(String arquivo) throws IOException, SQLException {  
    // ..  
}
```



Combinando as técnicas

3) Você pode, também, escolher tratar algumas exceções e declarar as outras no throws:

```
public void abre(String arquivo) throws IOException {  
    try {  
        objeto.metodoQuePodeLancarIOeSQLException();  
    } catch (SQLException e) {  
        // ..  
    }  
}
```



Melhorando o método sacar()

- O que poderíamos fazer, ao invés de devolver um valor booleano?
- Exatamente! Lançar uma exceção;
- Dessa forma, resolvemos o problema de esquecimento de um `if`;
- Para lançarmos uma exceção, precisamos da palavra reservada `throw`



Alterações do método sacar()

- Agora, estamos lançando a exceção chamada RuntimeException

```
public void sacar(double valor) {  
    if(valor > this.saldo) {  
        1 throw new RuntimeException();  
    } else {  
        this.saldo -= valor;  
    }  
}
```



Conhecendo RuntimeException

- É a mãe de todas as exceções `unchecked`;
- A desvantagem é que ela é muito genérica;
- Quem receber esse erro, tem dificuldade para saber o que houve;
- O que fazer?



Uma exception mais específica

```
public void sacar(double valor){  
    if(valor > this.saldo){  
        1 throw new IllegalArgumentException();  
    }else{  
        this.saldo -= valor;  
    }  
}
```



IllegalArgumentException

- É uma exceção que “fala” um pouco mais do erro ocorrido;
- É uma exceção nativa do java;
- É do tipo **unchecked** pois estende de RuntimeException;



Como chamar o método sacar()?

- Agora, podemos tratar a exceção lançada pelo método sacar():

```
public static void main(String[] args) {  
    ContaBancaria minhaConta = new ContaBancaria();  
    minhaConta.depositar(300);  
    1 try {  
        minhaConta.sacar(500);  
    } catch (IllegalArgumentException e) {  
        System.out.println("Saldo Insuficiente");  
    }  
}
```




Construtores em Exceptions

- E se nós invocássemos uma sobrecarga do construtor padrão da nossa exceção?

```
public void sacar(double valor){  
    if(valor > this.saldo){  
        throw new IllegalArgumentException("Saldo Insuficiente");  
    }else{  
        this.saldo -= valor;  
    }  
}
```



Acesso às mensagens

- O método getMessage() da classe Throwable, mãe de todas as exceptions, retorna a mensagem informada:

```
public static void main(String[] args) {  
    ContaBancaria minhaConta = new ContaBancaria();  
    minhaConta.depositar(300);  
    try {  
        minhaConta.sacar(500);  
    } catch (IllegalArgumentException e) {  
        1 System.out.println(e.getMessage());  
    }  
}
```

O que deve ser protegido no try-catch?

```
20 public static void main(String[] args) {  
21     ContaBancaria minhaConta = new ContaBancaria();  
22     minhaConta.depositar(300);  
23     try {  
24         1 minhaConta.sacar(500);  
25     } catch (IllegalArgumentException e) {  
26         2 System.out.println(e.getMessage());  
27     }  
28     3 System.out.println("Saque concluído com sucesso");  
29 }  
30 }
```

Problems Javadoc Declaration Servers Progress Console

<terminated> ContaBancaria [Java Application] C:\Java\jre1.5.0_14\bin\javaw.exe (10/09/2010 09:38:16)

Saldo Insuficiente
Saque concluído com sucesso 4

Melhorando o fluxo

- O que mudou? Faz sentido?

```
20 public static void main(String[] args) {
21     ContaBancaria minhaConta = new ContaBancaria();
22     minhaConta.depositar(300);
23     try {
24         1 minhaConta.sacar(500);
25         System.out.println("Saque concluído com sucesso");
26     } catch (IllegalArgumentException e) {
27         2 System.out.println(e.getMessage());
28     }
29 }
30 }
```

Problems Javadoc Declaration Servers Progress Console

<terminated> ContaBancaria [Java Application] C:\Java\jre1.5.0_14\bin\javaw.exe (10/09/2010 09:47:03)

Saldo Insuficiente 3



Exceções personalizadas

- Java possui uma boa API de exceções;
- É comum que o programador queira controlar melhor as exceções;
- Em Java, podemos criar e controlar nossas próprias exceções;
- Uma exception é uma classe java comum, que é uma extensão de algum tipo de exceção;



Criação e uso de exceptions

```
1 class SaldoInsuficienteException extends RuntimeException{  
    public SaldoInsuficienteException(String mensagem) {  
        super(mensagem);  
    }  
}  
  
public void sacar(double valor){  
    if(valor > this.saldo){  
2        throw new SaldoInsuficienteException("Saldo Insuficiente");  
    }else{  
        this.saldo -= valor;  
    }  
}
```

- Como mudar de Unchecked para Checked ?



Uma nova clausula

- Em um bloco de try-catch, são executados os comandos do bloco **try**;
- Em caso de exceptions, são executados os comando do bloco **catch**;
- Mas o Java nos permite criar um terceiro bloco, que SEMPRE será executado: **finally**



Estrutura do bloco

- Na figura abaixo, é apresentada a estrutura padrão para controle de exceções:

```
try {  
    // bloco try  
} catch (IOException ex) {  
    // bloco catch 1  
} catch (SQLException sqlex) {  
    // bloco catch2  
} finally {  
    // bloco finally  
}
```




Exercícios

- Criar as exceções:
ValorInvalidoException e
SaldoInsuficienteException;
- Atualizar o método sacar() para lançar
as exceções criadas;
- Criar a classe TestaExcecao para usar o
método sacar() e tratar suas exceções;
 - O valor a ser sacado deve ser informado
pelo usuário;



Bibliografia

- Java - Como programar, de Harvey M. Deitel
- Use a cabeça! - Java, de Bert Bates e Kathy Sierra
- (Avançado) Effective Java Programming Language Guide, de Josh Bloch



Referências WEB

- SUN: www.java.sun.com

Fóruns e listas:

- Javaranch: www.javaranch.com
- GUJ: www.guj.com.br

Apostilas:

- Argonavis: www.argonavis.com.br
- Caelum: www.caelum.com.br



Java Standard Edition (JSE)

10. Controle de Exceções



Esp. Márcio Palheta

Gtalk: marcio.palheta@gmail.com