Controlando Erros com Exceções

Motivação

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA BAHIA

- O que **aconteceria** ao tentar **chamar** o **método saca** com um **valor fora** do **limite**?

- O **sistema mostraria** uma **mensagem** de **erro**, mas quem **chamou** o **método** saca **não saberá** que isso **aconteceu**.

- Como **avisar** aquele que **chamou** o **método** de que ele **não conseguiu fazer aquilo** que **deveria**?

Motivação

- Em **Java**, os **métodos** dizem qual o **contrato** que eles **devem seguir**. Se, ao tentar **sacar**, ele não **consegue** fazer o que **deveria**, ele **precisa avisar** ao **usuário** que o **saque** não foi **feito**.

- No **exemplo** abaixo: estamos **forçando** uma **Conta** a ter um **valor negativo**, isto é, estar num **estado inconsistente** de **acordo** com a nossa **modelagem**.

```
Conta minhaConta = new Conta();
minhaConta.deposita(100);
minhaConta.setLimite(100);
minhaConta.saca(1000);
//     o saldo é -900? É 100? É 0? A chamada ao método saca funcionou?
```

Motivação

- Em **sistemas** de **verdade**, é **comum** que **quem** saiba **tratar** o **erro** é **aquele** que **chamou** o **método** e não a **própria classe**! Portanto, nada mais **natural** do que a **classe** **sinalizar** que um **erro** **ocorreu**.

- Uma **solução** **simples** utilizada **antigamente** é a de **marcar** o **retorno** de um **método** como **boolean** e retornar **true**, se tudo ocorreu da maneira **planejada**, ou **false**, caso **contrário**:

```
oolean saca(doule quantidade) {                                    b
 if (quantidade > this.saldo + this.limite) { //posso sacar até saldo+limite
        System.out.println("Não posso sacar fora do limite!");
        return false;
  } else {
       this.saldo = this.saldo – quantidade;
       return true;
  }
}
```

Motivação

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA BAHIA

- Um novo **exemplo** de **chamada** o **método** anterior:

```
Conta minhaConta = new Conta();
minhaConta.deposita(100);
minhaConta.setLimite(100);
if (!minhaConta.saca(1000)) {
    System.out.println("Não saquei");
}
```

- **Repare** que tivemos de **lembrar** de **testar** o **retorno** do **método**. **Esquecer** de **testar** o **retorno** desse **método** teria **consequências drásticas**.

```
Conta minhaConta = new Conta();
minhaConta.deposita(100);

// ...
double valor = 5000;
minhaConta.saca(valor); // vai retornar false, mas ninguem verifica!
caixaEletronico.emite(valor);
```

Motivação

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA BAHIA

- Mesmo **tratando** o **retorno** de maneira **correta**, o que **faríamos** se fosse **necessário** sinalizar **quando** o usuário **passou** um valor **negativo** como **quantidade**?

- Uma **solução** seria **alterar** o **retorno** de **boolean** para **int** e retornar o **código** do **erro** que **ocorreu**. O que vcs **acham** disso ?

- Isso é **considerado** uma **má prática**. Além de você **perder** o **retorno** do **método**, o valor **devolvido** só É **legível** perante **extensa documentação**;

Alguns conceitos

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA BAHIA

- Vamos ver **como** a **JVM** age ao **deparar-se** com **situações inesperadas**;
- O **método main** chama o **metodo1** e esse, por sua vez, **chamando** o **metodo2**.
- Quando um **método termina** (**retorna**), ele volta para o **método** que o **invocou**. Ele **descobre** isso **através** da **pilha de execução** (***stack***).

```
class TesteErro {

    public static void main(String[] args) {
        System.out.println("inicio do main");
        metodo1();
        System.out.println("fim do main");
    }

    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[10];
        for (int i = 0; i <= 15; i++) {
            array[i] = i;
            System.out.println(i);
        }
        System.out.println("fim do metodo2");
    }
}
```

Alguns Conceitos

- O **metodo2** propositadamente **possui** um **enorme problema**: está **acessando** um **índice** que estará **fora** dos **limites** da **array** quando **chegar** em **10**!

- Sé **rodarmos** o código **teremos** a seguinte saída.

- O que isso **representa**? O que ela **indica**?

Alguns Conceitos

- Isso é o **conhecido** **rastro da pilha** (***stacktrace***). É uma **saída** **importantíssima** para o **programador**.
- O **sistema** de **exceções** do **Java** funciona da seguinte maneira: quando uma **exceção** é **lançada** (***throws***), a **JVM** entra em **estado** de **alerta** e vai **ver** se o **método** atual toma **alguma precaução** ao **tentar** **executar** esse **trecho** de **código**.
- Como o **metodo2** não está **tratando** esse problema, a **JVM** pára a **execução** dele **anormalmente**, sem **esperar** ele **terminar**;

Alguns Conceitos

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA BAHIA

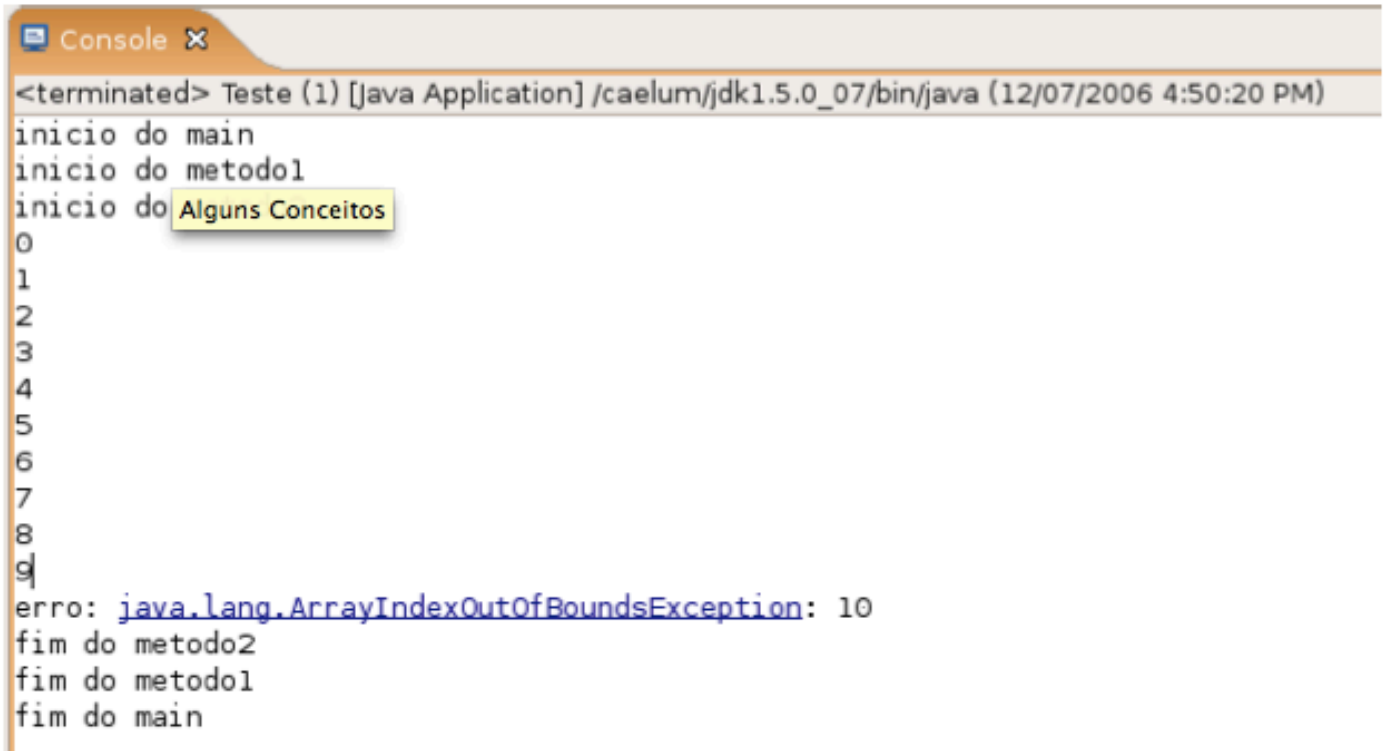
- Para **entender** o **controle** de **fluxo** de uma **Exception**, vamos **colocar** o **código** que vai **tentar** (***try***) ao **executar** o **bloco perigoso** e, caso o **problema** seja do tipo **ArrayIndexOutOfBoundsException**, ele será **pego** (***catched***).

- **Adicione** um **try/catch** em volta do **for**, pegando **ArrayIndexOutOfBoundsException**. O que o código **imprime**?

```
try {

    for (int i = 0; i <= 15; i++) {
        array[i] = i;
        System.out.println(i);
    }
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("erro: " + e);
}
```

Alguns Conceitos



- **Repare** que, a partir do **momento** que uma **exception** foi ***catched*** (**pega**, **tratada**, **handled**), a **execução** volta ao **normal** a partir daquele **ponto**.

Exceções Runtime

- O que **acontece** ao **executar** os seguintes **códigos**?

```
public class TestandoADivisao {

    public static void main(String args[
        int i =  71;
        i = i / 0;
        System.out.println("O resultado " + i);
    }
}
```

```
public class TestandoReferenciaNula {
    public static void main(String args[]) {
        Conta c = null;
        System.out.println("Saldo atual " + c.getSaldo());
    }
}
```

- Tais **problemas provavelmente** poderiam ser **evitados** pelo **programador**. É por esse **motivo** que o **java** não te **obriga** a dar o **try/catch** nessas **exceptions** e chamamos essas **exceções** de ***unchecked***.

Checked Exceptions

- Um **outro** **tipo** de **Execeções**, **obriga** a quem **chama** o **método** ou **construtor** a **trata-lá**. **Chamamos** esse tipo de **exceção** de ***checked***;

- Um **exemplo** é o de **abrir** um **arquivo** para **leitura**, onde pode **ocorrer** o **erro** do **arquivo** **não existir**;

```
class Teste {
    public static void metodo() {

        new java.io.FileInputStream("arquivo.txt");
    }
}
```

- O **código** acima **não compila** e o compilador **avisa** que é **necessário tratar** o **FileNotFoundException** que pode ocorrer;

Checked Exceptions

- Para **compilar**, temos duas **maneiras**. O **primeiro**, é tratá-lo com o **try** e **catch** do mesmo **jeito** que usamos no exemplo **anterior**:

```
public static void metodo() {

    try {
        new java.io.FileInputStream("arquivo.txt");
    } catch (java.io.FileNotFoundException e) {
        System.out.println("Nao foi possivel abrir o arquivo para leitura");
    }

}
```

- A **segunda** forma é **delegar** ele para quem **chamou** o nosso **método**, isto é, passar para a **frente**.

```
public static void metodo() throws java.io.FileNotFoundException {

    new java.io.FileInputStream("arquivo.txt");

}
```

Mais de um Erro

- É **possível tratar** mais de **um erro** quase que ao **mesmo tempo**:

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 {

    // ..
}
```

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) {
        // ..
    }
}
```

Lançanco Exceções

- O **método** **saca** da nossa classe **Conta** **devolve** um **boolean** caso **consiga** ou não **sacar**:

```
boolean saca(double valor) {
    if (this.saldo < valor) {
        return false;
    } else {
        this.saldo-=valor;
        return true;
    }
}
```

- **Podemos**, também, **lançar** uma **Exception**, o que é **extremamente** **útil**. Dessa maneira, **resolvemos** o **problema** de alguém poder **esquecer** de fazer um **if** no **retorno** de um **método**.

Lançanco Exceções

- A palavra **chave** **throw**, **lança** uma **Exception**. Bem **diferente** de **throws** que apenas **avisa** da **possibilidade** daquele **método** **lançá**-la, **obrigando** o outro **método** que vá utilizar deste de se **preocupar** com essa **exceção** em questão.

```
void saca(double valor) {
    if (this.saldo < valor) {
        throw new RuntimeException();
    } else {
        this.saldo-=valor;
    }
}
```

- No caso **anterio** um **RuntimeException** que é a **exception mãe** de todas as **exceptions** ***unchecked*** *é lançada*. A **desvantagem** é que ela é muito **genérica**; quem **receber** esse **erro** não sabe dizer **exatamente** qual foi o problema.

Lançanco Exceções

- **IllegalArgumentException** diz um pouco **mais**: **algo** foi **passado** como **argumento** e seu **método** **não gostou**.

- Para **pegar** esse **erro**, não **usaremos** um **if/else** e sim um **try/catch**, porque faz mais **sentido** já que a **falta** de **saldo** é uma **exceção**:

```
Conta cc = new ContaCorrente();
cc.deposita(100);

try {
    cc.saca(100);
} catch (IllegalArgumentException e) {
    System.out.println("Saldo Insuficiente");
}
```

Finally

- Os **blocos** **try** e **catch** podem **conter** uma **terceira cláusula** chamada **finally** que **indica** o que deve ser **feito após** o **término** do **bloco** **try** ou de um **catch** qualquer.

```
try {
    // bloco try
} catch (IOException ex) {
    // bloco catch 1
} catch (SQLException sqlex) {
    // bloco catch 2
} finally {
    // bloco finally
}
```

Exercício