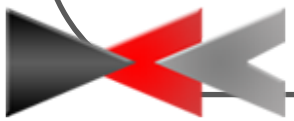
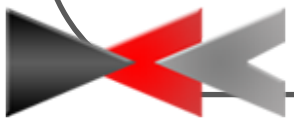
Tratamento de Erros



Tratamento de Erros

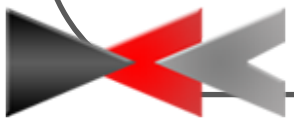
- Um método pode detectar uma falha, devendo repassar o tratamento dessa falha a um outro método apropriado
- Se introduzirmos o tratamento de falhas ao longo do código, a inteligibilidade pode ser comprometida
  - Além da repetição indevida de código
  - e de outros problemas...



Exemplo: Classe Contas

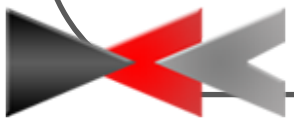
```
class Conta {
   private String numero;
   private double saldo;
   /* ... comportamento da classe ... */

   void debitar(double valor) {
    saldo = saldo – valor;
   }
}
```

Como evitar débitos acima do limite permitido?



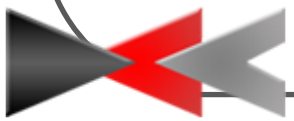
Soluções: Desconsiderar Operação

```
class Conta {
   private String numero;
   private double saldo;
   /* ... comportamento da classe ... */

   void debitar(double valor) {
       if (valor <= saldo)
          saldo = saldo – valor;
   }
}
```



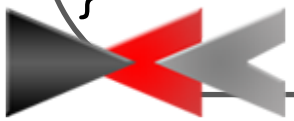
Desconsiderar Operação

- Problemas:
  - Quem solicita a operação não tem como saber se ela foi realizada ou não
  - Nenhuma informação é dada ao usuário do sistema



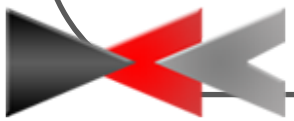
Soluções: Mostrar Msg de Erro

```
class Conta {
   static final String erro = “Saldo Insuficiente!”;
   private String numero;
   private double saldo;
   /* ... comportamento da classe ... */

   void debitar(double valor) {
       if (valor <= saldo)
          saldo = saldo – valor;
       else
          System.out.print(erro);
   }
}
```



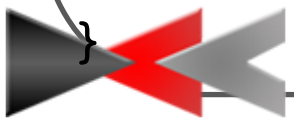
Mostrar Msg de Erro

- Problemas:
  - Informação é dada ao usuário do sistema, mas nenhuma sinalização de erro é fornecida para métodos que invocaram debitar
  - Há uma forte dependência entre a classe Conta e sua interface com o usuário
  - Não há uma separação clara entre o código da camada de negócio e o código da camada de interface com o usuário



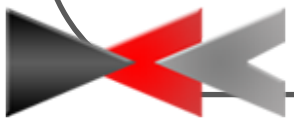
Soluções: Retornar Código Erro

```
class Conta {
   private String numero;
   private double saldo;
   /* ... comportamento da classe ... */

   boolean debitar(double valor) {
       boolean r = false;
       if (valor <= saldo) {
          saldo = saldo – valor;
          r = true;
       }
       return r;
   }
}
```



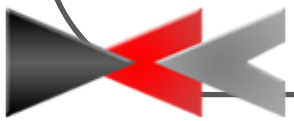
Retornar Código de Erro

- Problemas:
  - Dificulta a definição e o uso do método
  - Métodos que invocam debitar têm que testar o resultado retornado para decidir o que deve ser feito
  - A dificuldade é maior para métodos que retornam valores:
    - Se debitar já retornasse um outro valor?



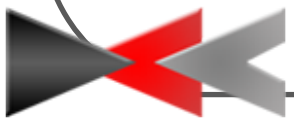
Exceções

- Sempre que acontece uma falha, uma exceção é lançada para sinalizar sua ocorrência
- Uma exceção causa uma interrupção abrupta no trecho de código em execução
- Java oferece suporte ao uso de exceções
  - São representadas por classes
  - São lançadas pelo comando throw
  - São tratadas pela estrutura try-catch-finally



Exceções

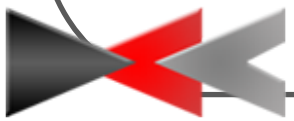
- Ao invés de códigos, teremos exceções...
- São objetos comuns, portanto têm que ter uma classe associada
- Classes representado exceções herdam e são subclasses de Exception
- Define-se subclasses de Exception para
  - Oferecer informações extras sobre a falha, ou
  - Distinguir os vários tipos de falhas



Exemplo

```
public class Calc {
   public int div(int a, int b) {
        return a/b;
   }
}
```

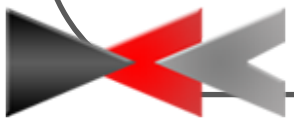
- O método div, se for chamado com o parâmetro b igual a zero, irá gerar um erro
  - Esse erro poderia ser lançado através de uma exceção e tratado através de um código próprio



Criar uma Exceção

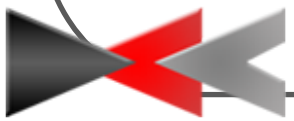
```
public class DivByZero extends Exception {
   public String toString() {
        return "Divisão por zero";
   }
}
```

- A exceção pode ser modelada como acima ou utilizada uma das exceções existentes
  - java.io;
  - java.io.IOException
  - Sinaliza que uma exceção de I/O ocorreu



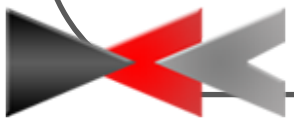
Lançando uma Exceção

```
public class Calc {
   public int div(int a, int b) throws DivByZero {
       if (b == 0) throw new DivByZero();
       return a/b;
   }
}
```



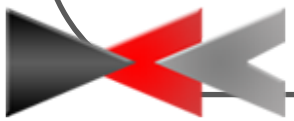
Outro Exemplo

```
class SIException extends Exception {
  private double saldo;
  private String  numero;
  SIException (double s, String n) {
    super ("Saldo Insuficiente!");
    saldo = s;
    numero = n;
  }
  double getSaldo() {return saldo;}
  /* ... */
}
```



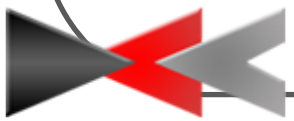
Classe Conta

```
class Conta {
  /* ... */
  void debitar(double v)
    throws SIException {
    if (v <= saldo) saldo = saldo - v;
    else {
      SIException e;
      e = new SIException(saldo,numero);
      throw e;
    }
}
```



Exceções levantadas indiretamente também devem ser declaradas

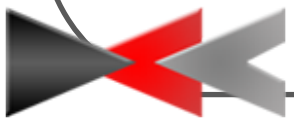
```
class Conta {
  /* ... */
  void transferir(Conta c, double v)
    throws SIException {
    this.debitar(v);
    c.creditar(v);
  }
}
```



Tratando uma Exceção

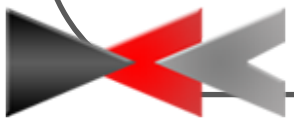
- Utilização da construção try - catch
  - try: código do programa
  - catch: tratamento da exceção gerada

```
...
Calc calc = new Calc();
try {
  int div = calc.div(x, y);
  System.out.println(div);
} catch (DivByZero e) {
   System.out.println(e);
}
...
```



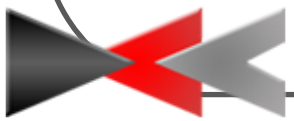
Tratando uma Exceção
Forma Geral

```
try {...
} catch (E1 e1) {
   ...
}
   ...
} catch (En en) {
   ...
} finally {...}
```



Tratando uma Exceção

- O bloco finally é sempre executado, seja após a terminação normal do try, após a execução de um catch, ou até mesmo quando não existe nenhum catch compatível

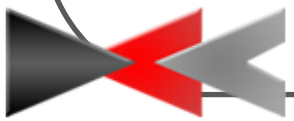


Exercícios

- Crie uma classe Conta e construa um método deposita que lançar uma exception chamada IllegalArgumentException sempre que o valor passado como argumento for inválido.

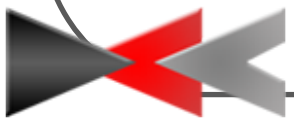
- Crie uma classe TestaDeposita com o método main. Crie uma conta e tente depositar valores inválidos

- Adicione um bloco try/catch para tratar o erro



Soluções

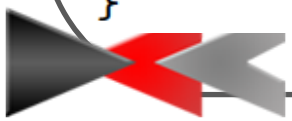
```
void deposita(double valor) {
    if (valor < 0) {
        throw new IllegalArgumentException();
    } else {
        this.saldo += valor - 0.10;
    }
}
```



Soluções

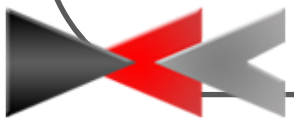
```
public static void main(String[] args) {
    Conta cp = new ContaPoupanca();
    cp.deposita(-100);
}
```

```
public static void main(String[] args) {
    Conta cp = new ContaPoupanca();

    try {
        cp.deposita(-100);
    } catch (IllegalArgumentException e) {
        System.out.println("Você tentou depositar um valor inválido");
    }
}
```



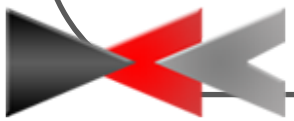
Exercícios

- Ao lançar a exceção IllegalArgumentException passe via construtor uma mensagem a ser exibida.
- A string recebida como parâmetro é acessível depois via o método getMessage() herdado de Exceptions.
- Altere sua classe TestaDeposita para exibir a mensagem de exeção através da chamada do getMessage()



Soluções

```
void deposita(double valor) {
    if (valor < 0) {
        throw new IllegalArgumentException("Você tentou depositar" +
                                           " um valor negativo");
    } else {
        this.saldo += valor  - 0.10;
    }
}
            public static void main(String[] args) {
                Conta cp = new ContaPoupanca();

                try {
                    cp.deposita(-100);
                } catch (IllegalArgumentException e) {
                    System.out.println(e.getMessage());
                }
            }
```

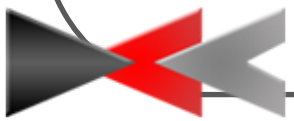


Exercícios

- Crie sua própria Exception, chamada ValorInvalidoException. Para isso, implemente uma classe que estenda de RuntimeException

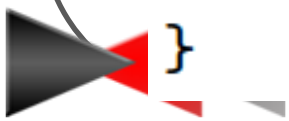
- Coloque um construtor na classe ValorInvalidoException que receba o valor inválido passado como parâmetro

- Lance-a em vez de IllegalArgumentException



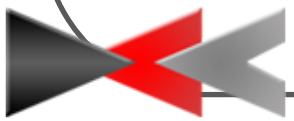
Soluções

```
class ValorInvalidoException extends RuntimeException {

}
    class ValorInvalidoException extends RuntimeException {

        ValorInvalidoException(double valor) {
            super("Valor invalido: " + valor);
        }

    }
    if (valor < 0) {
        throw new ValorInvalidoException(valor);
    }
```



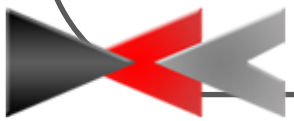
Tipos de Exceções

- Java possui dois tipos de exceções:
  - Unchecked Exceptions: são exceções que devem ser usadas para modelar falhas incontornáveis
  - Não precisam ser declaradas e nem tratadas
  - Checked Exceptions: são exceções que devem ser usadas para modelar falhas contornáveis.
  - Devem sempre ser declaradas pelos métodos que as lançam e precisam ser tratadas



Unchecked Exceptions

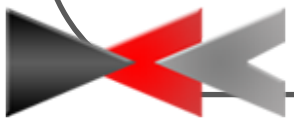
- Para criarmos uma classe que modela uma unchecked exception, devemos estender a classe Error ou RuntimeException
- Esse tipo de exceção não será verificado pelo compilador
- Tipicamente não são criadas exceções desse tipo, elas são usadas pela própria linguagem para sinalizar condições de erro



Exercícios

- Declare a classe ValorInvalidoException como filha direta de Exception em vez de RuntimeException. Ela passa a ser checked

- Nesse caso, o método deposita() precisa tratar a exceção ou lança-la através do throws ValorInvalidoException, se a opção for apenas o lançamento, quem chama esse método precisa tomar uma decisão entre try-catch ou fazer uso do throws novamente



Mesmo tratada a Exceção pode ser repassada

```
Public void f() throws DivByZero {
   Calc calc = new Calc();
   try {
       int div = calc.div(x, y);
       System.out.println(div);
   } catch (DivByZero e) {
       System.out.println(e);
       throw e;
   }
}
```