

# Exceções

# Robustez

- Sistemas robustos devem:
  - Fornecer formas eficientes para recuperação de falhas
  - Fornecer mensagens apropriadas de erros
  - Fazer validação dos dados
  - Evitar que situações indesejadas ocorram
  - Garantir a consistência das operações

# Classe Contas: definição

```
class Conta {  
    private String numero;  
    private double saldo;  
    /* ... */  
    public void debitar(double valor) {  
        saldo = saldo - valor;  
    }  
}
```

Como evitar débitos acima do limite permitido?

# Possíveis soluções

- Desconsiderar operação
- Mostrar mensagem de erro
- Retornar código de erro

# Desconsiderar Operação

```
class Conta {  
    private String numero;  
    private double saldo;  
    /* ... */  
    public void debitar(double valor) {  
        if (valor <= saldo)  
            saldo = saldo - valor;  
    }  
}
```

# Desconsiderar Operação

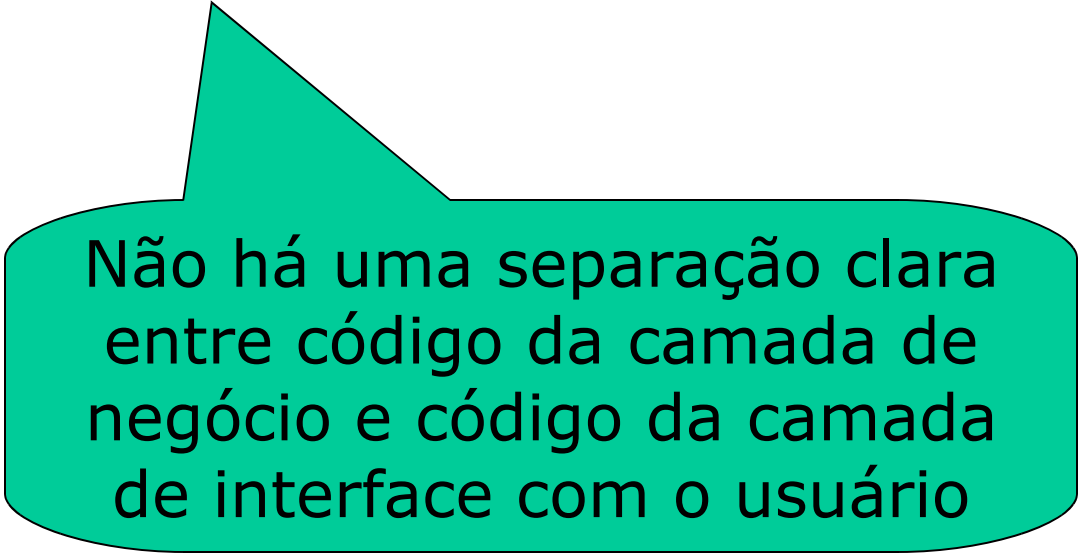
- Problemas:
  - quem solicita a operação não tem como saber se ela foi realizada ou não
  - nenhuma informação é dada ao usuário do sistema

# Mostrar Mensagem de Erro

```
class Conta {  
    static final String msgErro = "Saldo Insuficiente!";  
    private String numero;  
    private double saldo;  
    /* ... */  
    void debitar(double valor) {  
        if (valor <= saldo)  
            saldo = saldo - valor;  
        else System.out.print(msgErro);  
    }  
}
```

# Mostrar Mensagem de Erro

- Problemas:
  - O usuário do sistema recebe uma mensagem, mas nenhuma sinalização é fornecida para métodos que invocam debitar
  - Há uma forte dependência entre a classe `Conta` e sua interface com o usuário



Não há uma separação clara entre código da camada de negócio e código da camada de interface com o usuário



# Retornar Código de Erro

```
class Conta {  
    private String numero;  
    private double saldo;  
    /* ... */  
    boolean debitar(double valor) {  
        boolean r = false;  
        if (valor <= saldo) {  
            saldo = saldo - valor; r = true;  
        } else r = false;  
        return r;  
    }  
}
```

# Código de Erro: Problemas

```
class CadastroContas {  
    /* ... */  
    int debitar(String n, double v) {  
        int r = 0;  
        Conta c = contas.procurar(n);  
        if (c != null) {  
            boolean b = c.debitar(v);  
            if (b) r = 0; // Ok!  
            else r = 2;   // saldo insuficiente  
        } else r = 1;    // conta inexistente  
        return r;  
    }  
}
```

# Retornar Código de Erro

- Problemas:
  - dificulta a definição e uso do método:
    - métodos que invocam **debitar** têm que testar o resultado retornado para decidir o que deve ser feito
    - A situação é pior em métodos que retornam códigos de erro.
  - a dificuldade é ainda maior quando o método já retorna algum tipo:
    - o retorno pode ser o resultado da operação ou um código de erro.

O que fazer quando o tipo retornado não é primitivo?

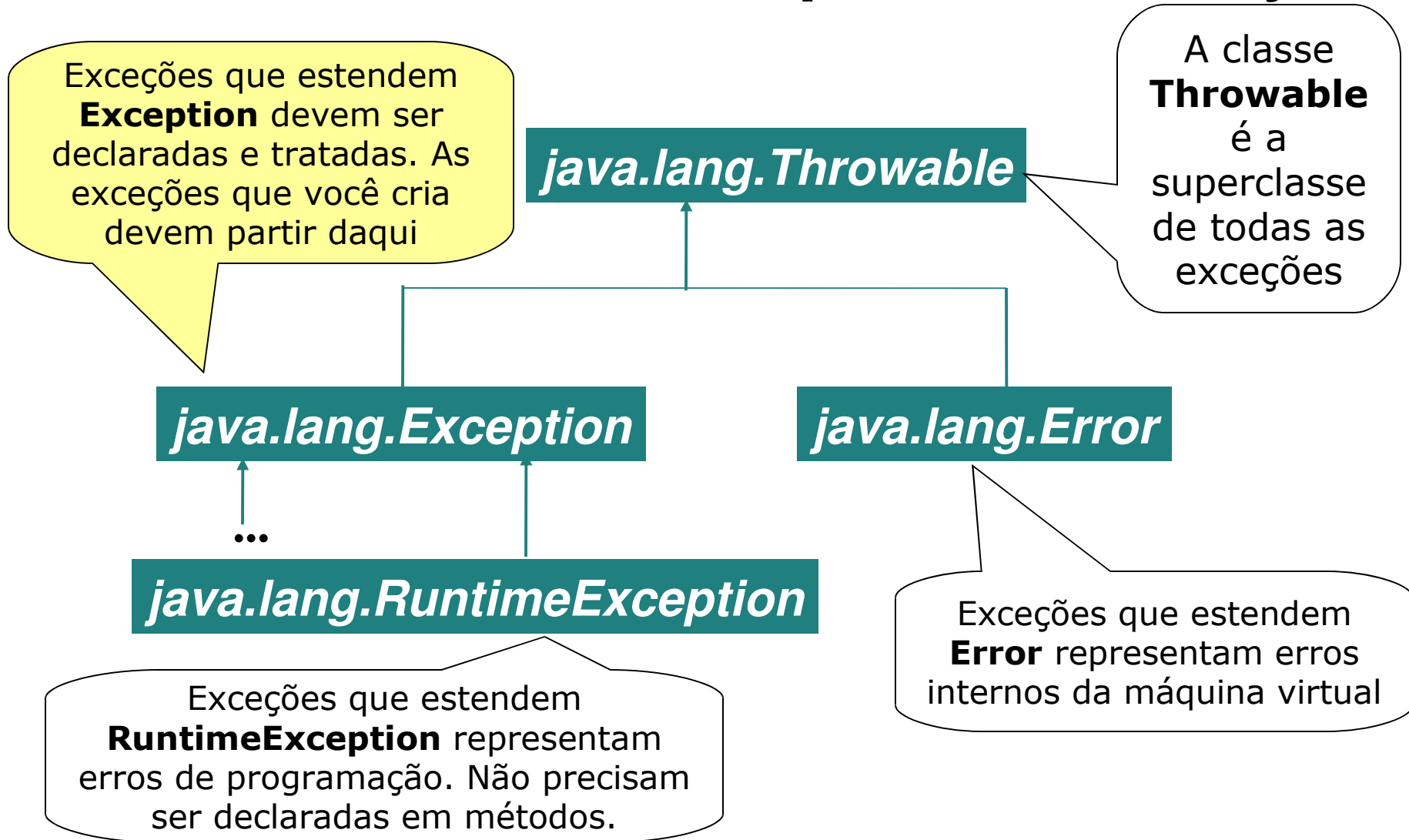
# Exceções (Definição)

- Exceções são o mecanismo utilizado por Java para tratamento de erros ou situações indesejadas
  - Erros de programação: Acesso a uma posição inválida de um array, divisão por zero, invocação de um método em uma referência nula.
  - Situações indesejadas: Uma conexão de rede indisponível durante uma comunicação remota, a ausência de um arquivo procurado localmente

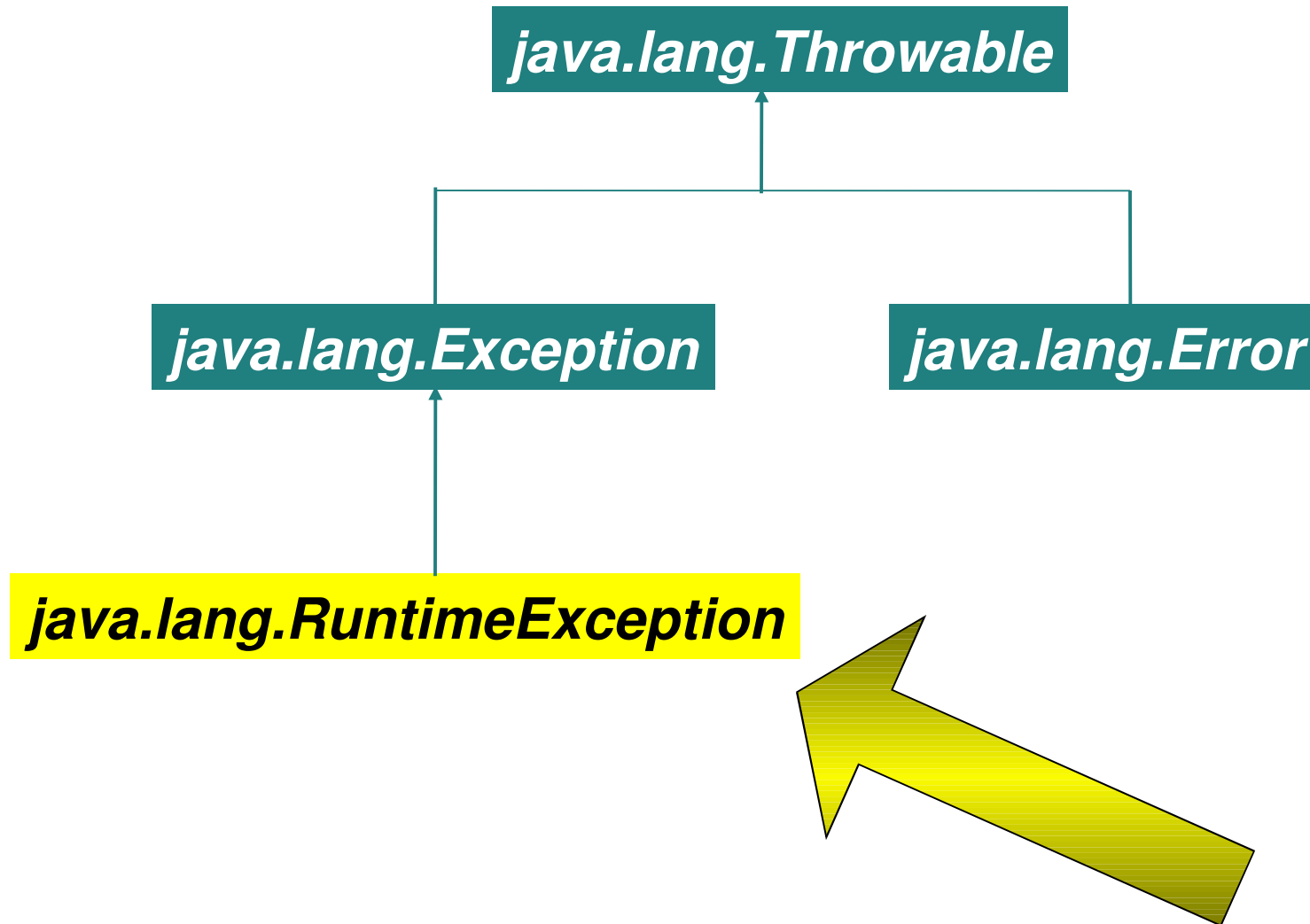
# Exceções

- Exceções são declaradas como classes Java
- Os objetos de uma exceção encapsulam informações relevantes sobre o erro ocorrido
- Exceções podem ser:
  - declaradas
  - lançadas
  - tratadas

# Tipos de exceções



# Exceções não checadas (unchecked exceptions)

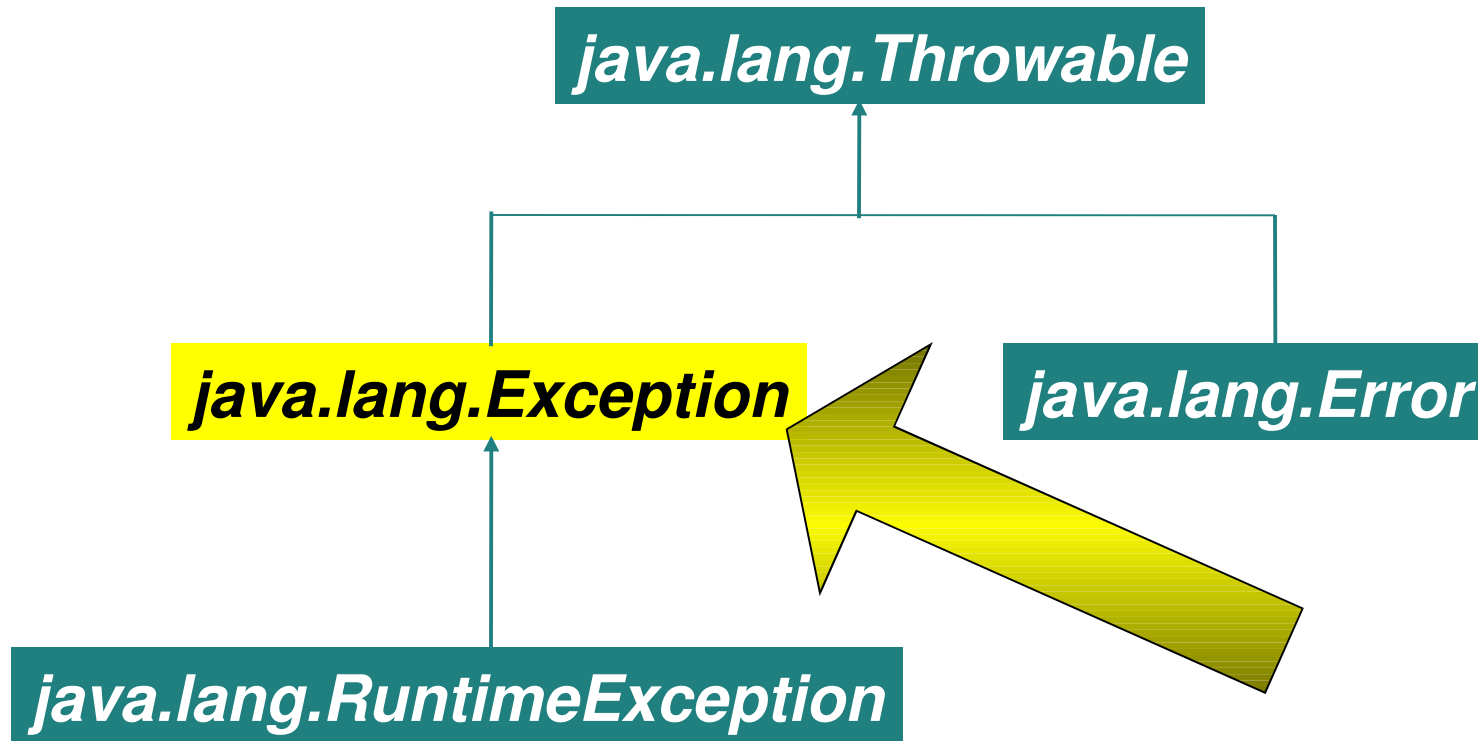


# Exceções não checadas (unchecked exceptions)

- Exceções que estendem `RuntimeException` indicam erros de programação e normalmente não são declaradas. Por exemplo, uma divisão por zero
- Qualquer método pode gerar essas exceções apesar de não explicitar isto na sua definição
- Tratar exceções deste tipo é como tentar corrigir um erro de programação durante a programação. Não faz muito sentido
- O que se faz é capturar a exceção e apresentar uma mensagem de erro agradável ao usuário indicando esta ocorrência



# Exceções checadas (checked exceptions)



# Exceções checadas (checked exceptions)

- Exceções checadas são todas as exceções que são subclasses de **Exception**, exceto **RuntimeException** e suas subclasses
- Devem ser **declaradas e tratadas** no código

# Exceções

- Exceções podem ser definidas pelo programador e devem ser subclasses de **java.lang.Exception**
- Definem-se novas exceções para:
  - oferecer informações extra sobre o erro
  - distinguir os vários tipos de erro/situação indesejada
  - Específicas para uma dada aplicação (exceções de negócio)

# Formato da declaração de de exceções

```
class Nome_Excecao_Exception extends Exception {  
  
    public Nome_Excecao_Exception () {  
  
    }  
  
    /* ... */  
}
```

Por convenção, é aconselhável que o nome de qualquer exceção definida pelo programador tenha o sufixo Exception: SaldoInsuficienteException, ObjetoInvalidoException, etc

# Exemplo de definição de Exceções

```
class SaldoInsuficienteException extends Exception {  
  
    public SaldoInsuficienteException() {  
        super("Saldo Insuficiente!");  
    }  
  
    /* ... */  
}
```

# Exemplo de definição de Exceções

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

# Declaração e lançamento

- ▶ Exceções são declaradas na assinatura dos métodos, que devem tratar um dado processamento, usando o comando **throws**
- ▶ Exceções são lançadas no corpo dos métodos usando o comando **throw**

```
void metodo(...) throws Exc1, Exc2, ExcN {  
    ...  
    throw new Exc1(...);  
}
```

- ▶ Exceções lançadas no corpo de um método, e não tratadas, devem ser declaradas.

# Quando exceções são lançadas

- Exceções são lançadas quando:
  - um método que lança exceções é chamado
  - você detecta uma situação de erro e levanta uma exceção com **throw**
  - você comete um **erro de programação** como, por exemplo, tentar acessar uma posição inválida de um array: `a[-1]`. Neste caso, Java levanta a exceção indicando o erro.
  - um **erro interno** ocorre em Java



# Declaração e lançamento

- Quando uma exceção é lançada e não é tratada, o fluxo de controle **passa para o método invocador e sobe** pelas chamadas de métodos até que a exceção seja tratada
- Se a exceção não for tratada em lugar nenhum, Java assume o controle e pára o programa

# Lançamento de Exceções

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

# Declarando e lançando exceções

```
class Conta {  
    /* ... */  
    void transferir(Conta c,  
double v)  
        throws  
SaldoInsuficienteExce
```

**debitar** levanta  
uma exceção!

```
    this.debitar(v);
```

Exceções levantadas indiretamente  
também devem ser declaradas!

```
}
```

```
}
```

# Tratamento de exceções

- ▶ Exceções são tratadas usando blocos **try-catch**

```
try {  
    // chamada aos métodos que podem lançar exceções  
}  
catch (Excl e1) {  
    // código para tratar um tipo de exceção  
}  
catch (ExcN eN) {  
    // código para tratar outro tipo de exceção  
}
```

# Tratamento de exceções

```
class CadastroContas {  
    /* ... */  
  
    void debitar(String n, double v)  
        throws SaldoInsuficienteException,  
            ContaInexistenteException {  
  
        Conta c = contas.procurar(n);  
        c.debito(v);  
  
    }  
}
```

# Tratamento de exceções

```
public static void main(String args[]){  
  
    try {  
  
        contas.debitar("123-4", 90.00);  
        System.out.println("Débito efetuado");  
  
    } catch (SaldoInsuficienteException sie) {  
        System.out.println(sie.getMessage());  
        System.out.print("Conta/saldo: ");  
        System.out.print(sie.getNumero() + "/" +  
            e.getSaldo());  
    } catch (ContaInexistenteException cie) {  
        System.out.print(cie.getMessage());  
    }  
    ...  
}
```

# Tratamento de exceções

- A execução do **try** termina ao final do bloco ou assim que uma exceção é levantada
- O primeiro **catch** de um supertipo da exceção é executado e o fluxo de controle passa para o código seguinte ao último catch
- Exceções mais específicas devem ser capturadas primeiro. Caso contrário, um erro de compilação é gerado
- Se não houver nenhum catch compatível, a exceção e o fluxo de controle são passados para o método que invocou

# Usando finally

- ▶ um trecho de código com **finally** é **sempre** executado, havendo ou não

```
try {  
    // chamada aos métodos que podem lançar exceções  
}  
catch (Exc1 e1) {  
    ...  
}  
catch (ExcN eN) {  
    ...  
}  
finally {  
    // Este código é sempre executado  
}
```



# Usando finally

```
public static void main(String args[]){
    ...
    try {

        contas.debitar("123-4", 90.00);
        System.out.println("Débito efetuado");

    } catch (SaldoInsuficienteException e) {
        System.out.println(e.getMessage());
        System.out.print("Conta/saldo: ");
        System.out.print(e.getNumero() + "/" + e.getSaldo());

    } catch (ContaInexistenteException e) {
        System.out.print(e.getMessage());

    } finally {
        System.out.println("Obrigado, volte sempre!");
    }
    ...
}
```

# Exceções e redefinição de métodos

- Métodos redefinidos não devem declarar exceções que não sejam **as mesmas** ou **subclasses** das exceções declaradas no método original
- Métodos redefinidos podem declarar um número menor ou maior de exceções que o declarado no método original contanto que a condição acima se verifique