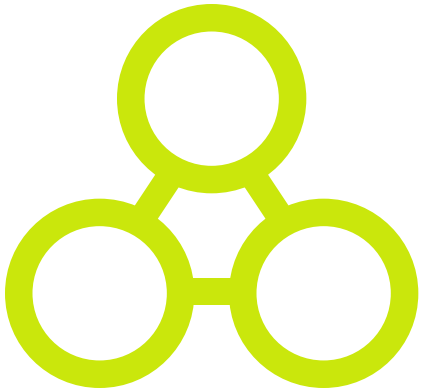


nemo

ontology & conceptual modeling research group

Linguagens de Programação

8 – Exceções

Vítor E. Silva Souza

(vitor.souza@ufes.br)

http://www.inf.ufes.br/~vitorsouza

Departamento de Informática

Centro Tecnológico

Universidade Federal do Espírito Santo

Conteúdo do curso

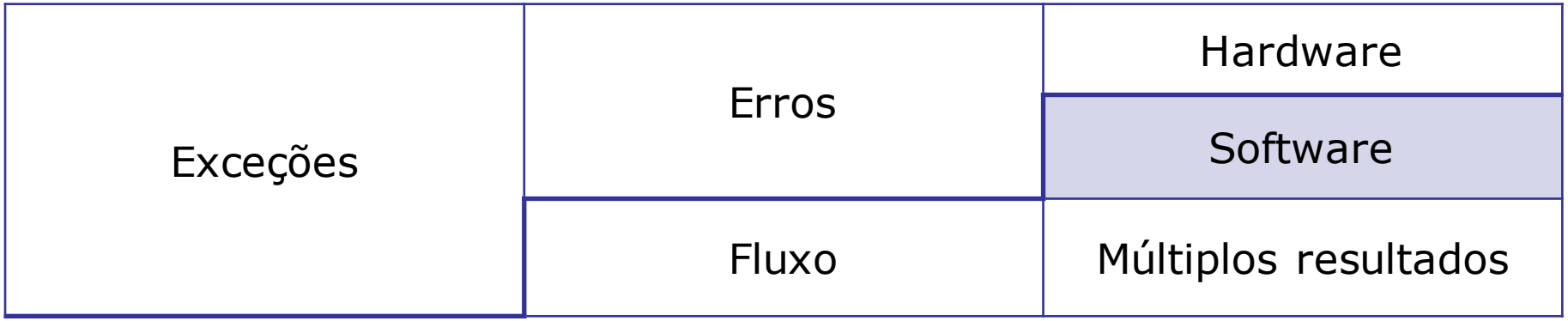
- Introdução;
- Amarrações;
- Valores e tipos de dados;
- Variáveis e constantes;
- Expressões e comandos;
- Modularização;
- Polimorfismo;
- Exceções;
- Concorrência;
- Avaliação de linguagens.

- Estes slides foram baseados em:
  - Slides do prof. Flávio M. Varejão;
  - Livro “Linguagens de Programação – Conceitos e Técnicas” (Varejão);
  - Livro “Linguagens de Programação – Princípios e Paradigmas, 2a edição” (Tucker & Noonan).

Conceituação

- Nem todas condições geradoras de erro podem ser detectadas em tempo de compilação;
- Software seguro e confiável deve implementar um comportamento aceitável na presença dessas condições anormais;
  - Em especial softwares críticos ou que trabalhem com dados sensíveis;
- Termo exceção designa um evento ocorrido durante a execução de um programa que desvia o fluxo normal;
- Uma exceção é uma condição provocada por uma situação excepcional a qual requer uma ação específica imediata.

Causas de exceções



- Erros de hardware:
  - Ex.: falha no disco rígido;
- Erros de software:
  - Ex.: acesso a índice inexistente de vetor;
- Múltiplos resultados:
  - Ex.: leitura de registro ou fim de arquivo.

Abordagens de LPs para exceções

- Não oferecer recursos próprios:
  - Tratamento através de mecanismos já existentes (testes, subprogramas e desvio incondicional);
  - Ex.: C, Pascal e Modula-2;
- Possuir mecanismo de tratamento de exceções:
  - Comandos específicos;
  - Novo tipo de fluxo de execução;
  - Ex.: Ada, C++ e Java.

Ausência de mecanismo de exceção

- Opções:
  - Deixar o programa abortar;
  - Testar a condição excepcional antes de ela ocorrer e realizar o tratamento imediato;
  - Retornar código de erro indicando a exceção ocorrida em:
    - Uma variável global;
    - No resultado da função; ou
    - Em um parâmetro específico.

Abortar

- Reduz a confiança do usuário no sistema;
  - Pode levar a prejuízos financeiros...
- Dificulta a depuração dos erros;
- Muitas exceções podem ser contornadas sem que seja necessário interromper a execução do programa.

Teste e tratamento imediato

- Carrega muito o texto do programa com código de tratamento:
  - Obscurece a funcionalidade do algoritmo com testes de exceções;
  - Subprogramas para tratamento reduzem esse problema;
- Programador tem de lembrar, identificar e testar todas as possíveis condições causadoras de exceções:
  - Isso normalmente não ocorre;
- Algumas exceções não podem ser tratadas localmente.

Teste e tratamento imediato

```
int divideInteiros(int numerador, int denominador) {
  if (denominador == 0)
     return trata_divisao_zero();
  else
     return numerador / denominador;
}

int trata_divisao_zero(void) {
  printf("Divisao por zero");
  return 1;
}
```

Quando há erro, retorna 1. Será que isso é uma boa ideia?

Teste e tratamento imediato

```
void  executaFuncionalidade(int x) {
  printf ("Faz alguma coisa!!!");
}

void f(int x) {
  if (condicao1(x)) trata1();
  if (condicao2(x)) trata2();
  if (condicao3(x)) {
     printf("Nao consegue tratar aqui");
     exit(1);
  }
  executaFuncionalidade(x);
}
```

Quantas linhas de tratamento e quantas de real funcionalidade em f()?

Retorno de código de erro

- Quem chama deve realizar teste e tratamento para cada código de retorno:
  - Sobrecarga de código fica ainda maior;
  - Testes no local da exceção e no código de chamada;
  - Pode duplicar o tamanho de um programa;
- Resolve o problema de tratamento não local da exceção;
- Experiência mostra que o programador não testa todos os códigos de retorno possíveis;
  - Motivo: não é obrigatório fazê-lo;

Retorno de código de erro

- Resultado da função:
  - Nem sempre possível por incompatibilidade com o resultado normal da função;
- Variável global:
  - Usuário da função pode não ter ciência de que essa variável existe;
  - Motivo: não fica explícito na chamada (ex.: `errno` de C);
  - Outra exceção pode ocorrer antes do tratamento da anterior, um problema maior em programas concorrentes;

Retorno de código de erro

- Parâmetro de saída:
  - Melhor do que o retorno em variável global ou no resultado da função;
  - Nem toda LP dá suporte, pode exigir “gambiarras” (ex.: uso de ponteiros em C);
  - Exige inclusão de um novo parâmetro nas chamadas dos subprogramas;
  - Requer a propagação desse parâmetro até o ponto de tratamento da exceção;
  - Diminui a redigibilidade do código.

Retorno de código de erro

```
int f(int x) {
  if (condicao1(x)) return 1;
  if (condicao2(x)) return 2;
  if (condicao3(x)) return 3;
  executaFuncionalidade(x);
  return 0;
}

void g() {
  int resp;
  resp = f(7);
  if (resp == 1) trata1();
  if (resp == 2) trata2();
  if (resp == 3) trata3();
}
```

Qual abordagem foi usada aqui?

Outras opções em C

- Utilização do sistema de manipulação de sinais de sua biblioteca padrão:
  - Sinais gerados por função `raise()` em resposta a comportamento excepcional;
  - Tratamento na função `signal()`;
- Uso das funções da biblioteca padrão `setjmp()` e `longjmp()`:
  - Salvam e recuperam estado do programa;
  - Função `longjmp()` é um goto não local;
  - Passa o controle do programa para o ponto onde o último `setjmp()` foi executado;

Outras opções em C

- Exigem tratamento imediato da exceção;
- Solução `signal()` concentra o tratamento de todas as exceções em uma única função;
- Solução `setjmp()` e `longjmp()` permite localizar o tratamento em qualquer ponto do programa;
  - Restringe o tratamento ao último `setjmp()`;
- Soluções complexas e com baixa legibilidade;
- Fica a critério do programador C decidir qual a abordagem de tratamento será utilizada.

Mecanismos de tratamento de exceções

- Buscam garantir e estimular o tratamento das condições excepcionais sem que haja uma grande sobrecarga do texto do programa;
- Quando uma exceção ocorre ela necessita ser tratada;
- Tratador de Exceção:
  - Bloco ou unidade de código que manipula a exceção;
- Sinalização ou disparo da exceção:
  - Ação de indicar a ocorrência da exceção e transferir o controle para o tratador.

Tipos de exceções: diferem na...

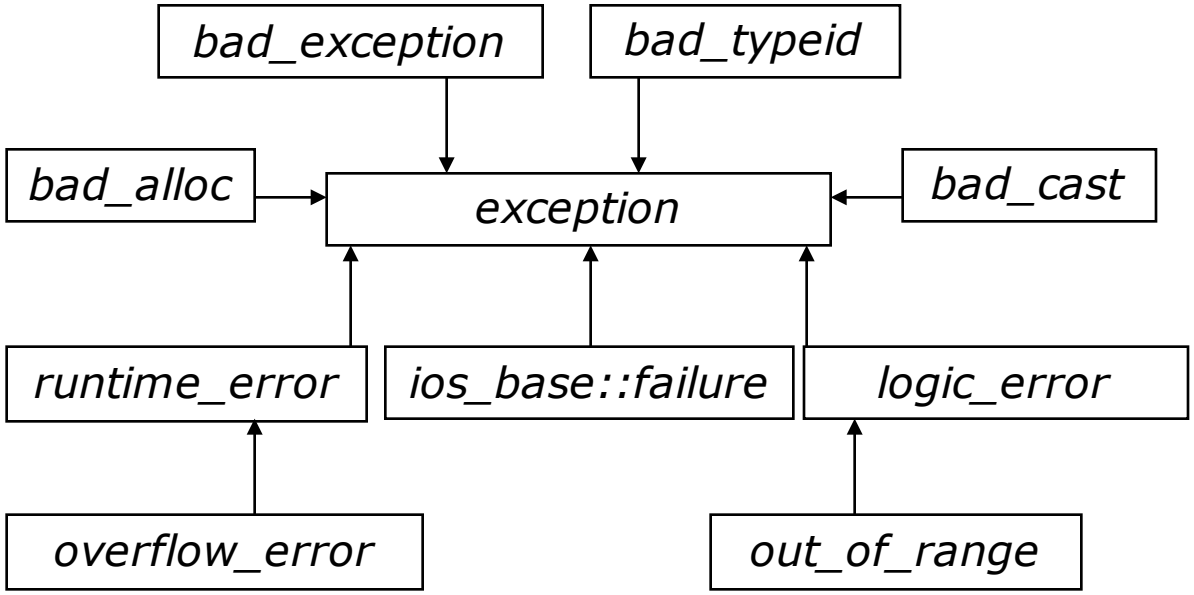
- Definição:
  - Pela LP: overflow em C++;
  - Pelo programador: estoque baixo;
- Sinalização:
  - Pela LP: acesso indevido a vetor em Java;
  - Pelo programador: estoque baixo;
- Tratamento:
  - Obrigatório: programador tem de tratar;
  - Opcional: programador pode não tratar.

Exceções em LPs orientadas a objeto

- Objetos:
  - Podem ser lançados para outras partes do programa seguindo um fluxo de controle distinto do usual;
  - Classes podem ser especiais ou não;
- Podem/devem ser organizadas dentro de uma hierarquia de classes;
  - Exemplo em C++:

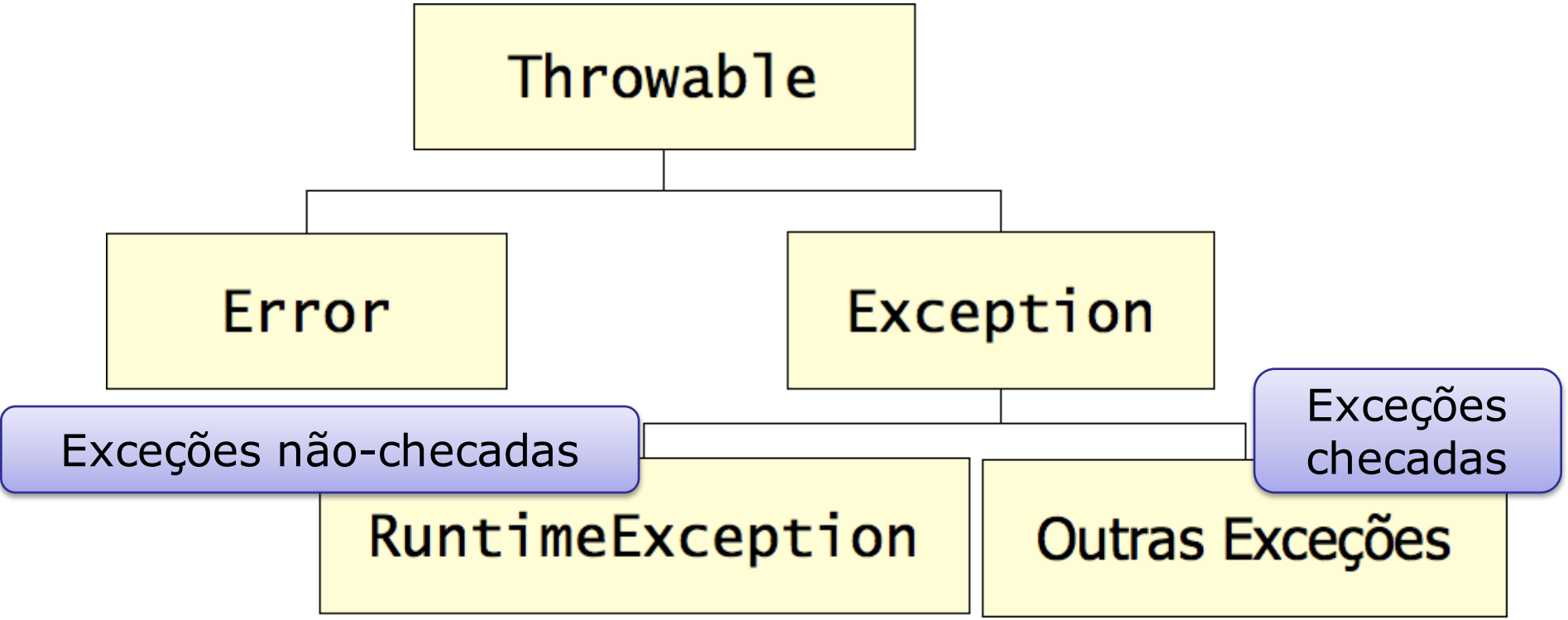
```
class ErroMedico {};
class ErroDiagnostico: public ErroMedico {};
class ErroCirurgia: public ErroMedico {};
```

Exceções padrão de C++



Exceções em Java

- `Throwable`: classe especial, objetos disparáveis pelo mecanismo de exceção;
- `Error`: classes de erros graves (não recuperáveis), programador normalmente não manipula;



Exceções em Java

- `Exception`:
  - Superclasse de todas as exceções manipuláveis pelo programador;
- `RuntimeException`:
  - Subclasse especial de exceções;
  - Geralmente são disparadas pela LP;
  - Programador não é obrigado a tratá-las;
  - Embora normalmente indiquem problemas sérios que devem implicar na terminação do programa, permitem que o programador tenha a opção de tratá-las;
  - Aumenta a redigibilidade e reduz a confiabilidade pois tratamento não é requerido.

Definição de exceções em Java

- Necessário criar subclasse de `Exception` ou de alguma de suas subclasses:

```
class UmaExcecao extends Exception {
  private float f;

  public UmaExcecao(String msg, float x) {
     super(msg);
     f = x;
  }

  public float contexto() {
     return f;
  }
}
```

Exceções em Python

- `BaseException`: classe base, como `Throwable` do Java;
  - `SystemExit`: para interceptar pedidos de saída (`system.exit()`);
  - `KeyboardInterrupt`: para interceptar interrupções (Ctrl+c);
  - `GeneratorExit`: usada por desenvolvedores de generators;
  - `Exception`: classe base para exceções “normais”.

Sinalização de exceções

- Pelo próprio mecanismo de exceções;
- Explicitamente pelo programador:
  - Uso de `try` e `throw`;
  - Em C++:

```
try {
  throw ErroMedico();
}
```

  - Em Java:

```
try {
  throw new Exception();
}
```

  - Python: como C++, porém usando `raise`.

Tratadores de exceções

- Trecho de código do programa responsável por tomar atitudes em resposta à ocorrência de uma exceção;
- Não são chamados explicitamente:
  - Não precisam possuir nome;
- Uso de `catch` associado a `try`;
  - Em Python: `except`.

Tratadores de exceções

```
String n = "635";
String d = "27";
try {
  int num = Integer.valueOf(n).intValue();
  int den = Integer.valueOf(d).intValue();
  int resultado = num / den;
}
catch (NumberFormatException e) {
   System.out.println("Erro na Formatação");
}
catch (ArithmeticException e){
   System.out.println("Divisao por zero");
}
```

Tratadores de exceções

- Melhoram a redigibilidade:
  - Não necessita incluir testes de exceções após cada chamada;
- Melhora a legibilidade:
  - Separa código de tratamento do código de funcionalidade.

Tratadores de exceções

- Atenção à distribuição dos tratadores:

```
try {
  // Código no qual várias exceções podem
  // ser sinalizadas.
}
catch (ErroMedico &e){
  // Trata qualquer erro médico.
}
catch (ErroDiagnostico &e){
  // Trata apenas erro de diagnóstico.
}
catch (ErroCirurgia &e){
  // Trata apenas erro de cirurgia.
}
```

Por que a distribuição acima é inapropriada?

Tratadores de exceções

- Em Python:

```
try:
    x = int(raw_input("Digite um número: "))
    print "Numero %d é valido" % x

# Poderia ser Exception (genérica, mas não tanto)
# Poderia ser ValueException (específica para o caso)
except BaseException:
    print "Numero invalido"
```

Captura de qualquer exceção

- Em Java:

```
try {
  int num = Integer.valueOf(n).intValue();
  int den = Integer.valueOf(d).intValue();
  int resultado = num / den;
}
catch (NumberFormatException e){
  System.out.println ("Erro na Formatação");
}
catch (ArithmeticException e){
  System.out.println("Divisão por zero");
}
catch (Exception e){
  System.out.println ("Qualquer outra Exceção");
}
```

Java possui hierarquia de exceções com raiz única.

Captura de qualquer exceção

- Em C++:

```
try {
  // Código que dispara exceções.
}
catch (ErroDiagnostico &e){
  // Trata apenas erro de diagnóstico
}
catch (ErroCirurgia &e){
  // Trata apenas erro de cirurgia
}
catch (ErroMedico &e){
  // Trata qualquer erro médico
}
catch ( ... ) {
  // Trata qualquer outra exceção
}
```

Por não possuir raiz única, C++ possui sintaxe especial.

Propagação de exceções

```
public void teste(int num) {
  try { // 1
     try { // 2
        try { // 3
           switch(num) {
           case 1: throw new NumberFormatException();
           case 2: throw new EOFException();
           case 3: throw new IOException();
           default: throw new NullPointerException();
        }
        catch (EOFException e) { /* Trata no bloco 3. */ }
     }
     catch (IOException e) { /* No bloco 2. */ }
  }
  catch (NumberFormatException e) { /* No bloco 1. */ }
}
```

O que acontece com a NPE? Por que?

Propagação para fora do método

```
// Não trata IOException localmente mais.
public void teste(int num) throws IOException {
  try { // 1
     try { // 2
        try { // 3
           switch(num) {
           case 1: throw new NumberFormatException();
           case 2: throw new EOFException();
           case 3: throw new IOException();
           default: throw new NullPointerException();
        }
        catch (EOFException e) {/* Trata no bloco 3. */ }
     }
     catch (NullPointerException e) { /* No bloco 2. */ }
  }
  catch (NumberFormatException e) { /* No bloco 1. */ }
}
```

Propagação para fora do método

```
public class Teste {
  public void outroTeste() {
     try {
        metodoLancador();
     }
     catch (IOException e) {
        System.out.println("Veio de outro método:");
        e.printStackTrace();
     }
  }

  public void metodoLancador() throws IOException {
     throw new IOException();
  }
}
```

Relançamento de exceções

- Exceções podem ser parcialmente tratadas em um bloco e relançadas para o bloco externo:

```
public void outroTeste() throws IOException {
  try { // 1
     try { // 2
        throw new IOException();
     }
     catch (IOException e) {
        // Tratamento parcial 1...
        throw e;
     }
  }
  catch (IOException e) {
     // Tratamento parcial 2...
     throw e;    // O tratamento termina externamente.
  }
}
```

Especificação de exceções lançadas

- Subprograma pode lançar exceção sem tratar;
- Subprograma necessita explicitar exceções que pode disparar;
- Em C++:

```
void f() throw(A,B,C); // Dispara A, B e C
void g() throw();      // Não dispara
void h();              // Pode disparar qualquer exceção
```

  - C++ não verifica se o compromisso assumido na especificação está sendo cumprido;
  - C++ não obriga a função chamadora a tratar todas as exceções possíveis de serem geradas pela função chamada.

Especificação de exceções lançadas (C++)

- Exceção não declarada pode ser lançada;
- Função `set_unexpected()` pode ser usada para determinar o que fazer nesse caso:

```
class ErroI { };

class ErroII { };

void f() throw (ErroI) {
  throw ErroII();
}

void exc() {
  cout << "erro inesperado";
  exit(1);
}
```

```
int main() {
  set_unexpected(exc);
  try {
     f();
  }
  catch (ErroI){
     cout << "erro em f";
  }
}
```

Especificação de exceções lançadas (C++)

- Não é obrigatório tratar as exceções declaradas;
- Neste caso, `set_unexpected()` não ajuda: *terminating with uncaught exception of type ErroI*:

```
class ErroI { };

class ErroII { };

void f() throw (ErroI) {
  throw ErroI();
}

void exc() {
  cout << "erro inesperado";
  exit(1);
}
```

```
int main() {
  set_unexpected(exc);
  try {
     f();
  }
  catch (ErroII){
     cout << "erro em f";
  }
}
```

Especificação de exceções lançadas (Java)

- Todo método é obrigado a indicar em seu cabeçalho as exceções checadas que dispara;
- Avisa aos usuários do método quais exceções podem ocorrer e não são tratadas;
- O programa que chamou o método deve tratar as exceções checadas ou redispará-las;

```
public void teste(int num) throws EOFException,
IOException, NumberFormatException {
  switch(num) {
     case 1: throw new NumberFormatException();
     case 2: throw new EOFException();
     case 3: throw new IOException();
     default: throw new NullPointerException();
  }
}
```

Modos de continuação

- Terminação:
  - Assume o erro como crítico;
  - Não retorna ao ponto no qual a exceção foi gerada;
  - O controle retorna para um ponto mais externo do programa;
- Retomada:
  - Assume o erro como corrigível;
  - A execução pode retornar para o bloco no qual ocorreu a exceção;
  - Experiência indica baixa efetividade dessa opção;
- Maioria das LPs adota o modelo de terminação.

Terminação (em C++)

```
class ErroI { }; class ErroII { }; class ErroIII { };

void f() throw (ErroI) { throw ErroI(); }

int main() {
  cout << "comeca aqui\n";
  try {
     cout << "passa por aqui\n";
     try { f(); }
     catch (ErroIII) { cout << "não passa por aqui\n"; }
     cout << "também não passa por aqui\n";
  }
  catch (ErroI) { cout << "erro I em f\n"; }
  catch (ErroII){ cout << "não passa por aqui\n"; }

  cout << "termina aqui\n";
}
```

Retomada (em Java)

```
class NaoPositivoException extends Exception {}
public class Retomada {
  static Scanner in = new Scanner(System.in);
  public static void main(String[] args) {
    boolean continua = true;
    while (continua) {
      continua = false;
      try {
        System.out.print("Entre um num. positivo:");
        int i = in.nextInt();
        if (i <= 0) throw new NaoPositivoException();
      } catch(NaoPositivoException e) {
        System.out.println("Tente novamente!!!");
        continua = true;
      }
    }
  }
}
```

A cláusula `finally`

- Usada quando queremos que um trecho de código seja executado independente de haver ou não exceção;
- Colocada após o último tratador;
- O bloco `finally` é sempre executado!
- Todo bloco `try` deve ter um ou mais blocos `catch` ou um bloco `finally`;
- Pode ter ambos, formando uma estrutura conhecida como `try` – `catch` – `finally`.

A cláusula finally

```
// Exemplo em Java:

try {
  // Código que pode lançar exceções...
}
catch (ExcecaoA e) {
  // Tratamento da exceção A,
  // ou qualquer subclasse de ExcecaoA.
  // e = instância da classe de exceção.
}
catch (ExcecaoB e) {
  // Tratamento da exceção B.
}
finally {
  // Código executado ao final.
}
```

Perda de exceção em Java

```
public class Perda {
  class InfartoException extends Exception {
     public String toString() { return "Urgente!"; }
  }
  void infarto() throws InfartoException {
     throw new InfartoException ();
  }
  class ResfriadoException extends Exception {
     public String toString() { return "Descanse!"; }
  }
  void resfriado() throws ResfriadoException {
     throw new ResfriadoException ();
  }
  public static void main(String[] s) throws Exception {
     Perda p = new Perda();
     try { p.infarto(); } finally { p.resfriado(); }
    }
}
```

Qual é a saída desse programa?

Exceções e polimorfismo

- Aumenta a complexidade dos programas;
- Necessário definir regras;
- Em Java, com relação a construtores e exceções:
  - Construtores são obrigados a lançar exceções declaradas no construtor da superclasse;
  - Construtores podem lançar exceções que não são declaradas no construtor da superclasse.

Exceções e polimorfismo: construtores

```
// Este código gera erro de compilação:
// Unhandled exception type Exception

class Pai {
  Pai() throws Exception { }
}

class Filho extends Pai {
  Filho() {
    // Chamada implícita à super(),
    // super() lança Exception!
  }
}
```

Exceções e polimorfismo

- Em Java, com relação a exceções e sobrescrita:
  - Não é obrigatório declarar que os métodos da subclasse lançam as exceções declaradas no método da superclasse que foi sobrescrito;
  - Métodos da subclasse não podem propagar exceções que não estão declaradas no método que foi sobrescrito;
  - A exceção: podem propagar exceções que sejam subclasses de uma das exceções declaradas no método que foi sobrescrito.

Exceções e polimorfismo: sobrescrita

```
// Este código gera erro de compilação:
// Exception Exception is not compatible with throws
// clause in Pai.metodo2()

class Pai {
  void metodo1() throws Exception { }
  void metodo2() throws ClassNotFoundException { }
}

class Filho extends Pai {
  @Override
  void metodo1() { } // OK!
  @Override
  void metodo2() throws Exception {
     throws new CloneNotSupportedException();
  }
}
```

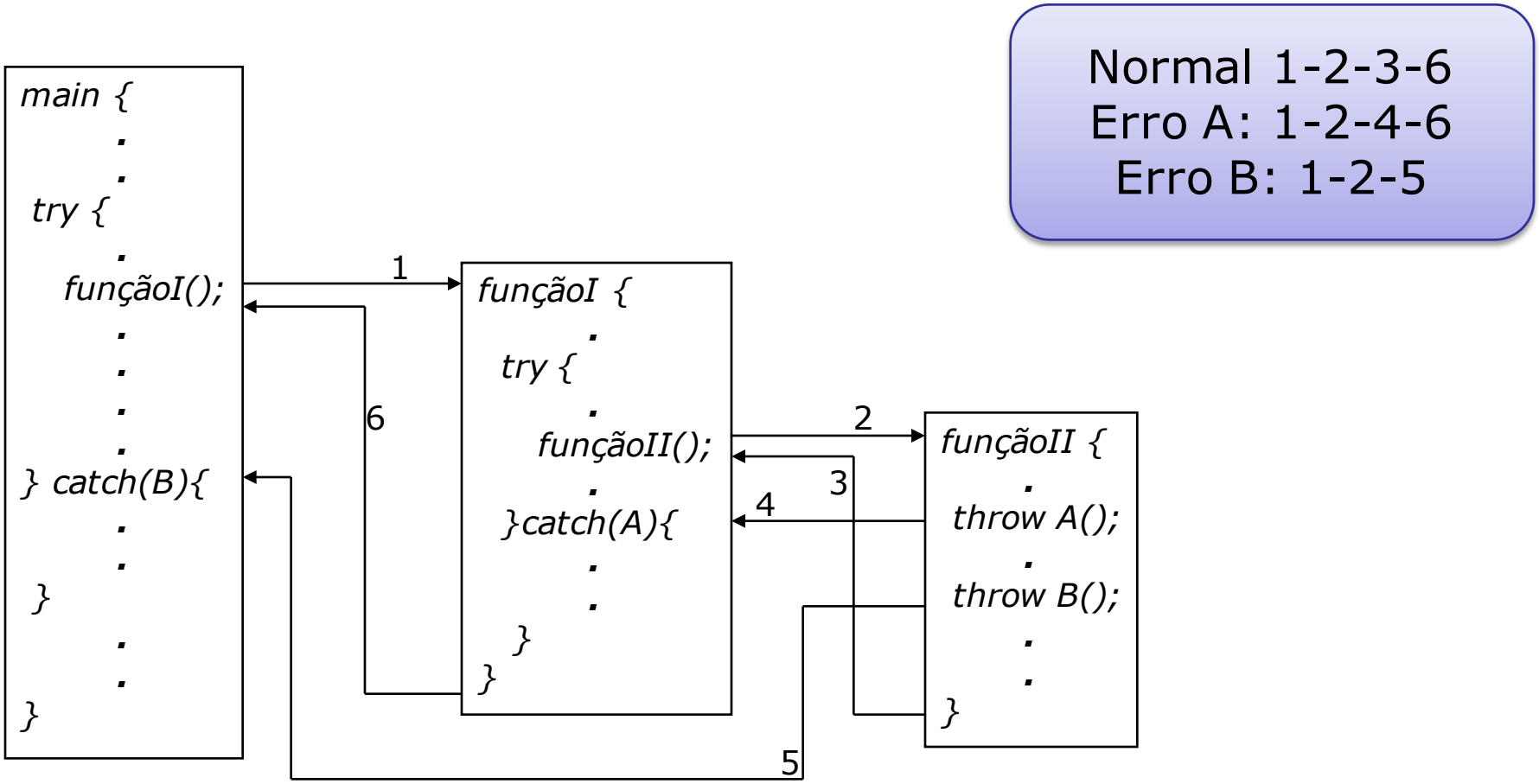
Exceções e polimorfismo: sobrescrita

```
public class Teste {
  public static void main(String[] args) {
    try {
      Pai p = new Filho();

      // Este método está declarado como lançando
      // ClassNotFoundException, porém a
      // implementação no filho lança outra exceção!
      p.metodo2();
    }
    catch (ClassNotFoundException e) {
      e.printStackTrace();
    }
  }
}
```

Conclusões

- Exceções organizam o fluxo de controle:



Conclusões: vantagens

- Exceções melhoram a legibilidade dos programas:
  - Separam o código com a funcionalidade principal do programa do código responsável pelo tratamento de exceções;
- Aumentam a confiabilidade e robustez dos programas:
  - Normalmente requerem o tratamento obrigatório das exceções ocorridas;
- Promovem a ideia de recuperação dos programas mesmo na presença de situações anômalas:
  - Incentivam a reutilização e a modularidade do código responsável pelo tratamento;

Conclusões: desvantagens

- Trazem maior complexidade para o aprendizado da linguagem;
- Podem reduzir a eficiência computacional dos programas nessa linguagem;
- Fragilidades em C++:
  - Número reduzido de exceções pré-definidas na biblioteca padrão;
  - As funções não são obrigadas a especificar as exceções que podem propagar;
  - Não detecção em tempo de compilação da quebra de compromisso com uma dada especificação;
  - Não existe obrigação de explicitar a exceção relançada para o nível superior.

http://nemo.inf.ufes.br/