Paradigmas de Linguagens de Programação

Tratamento de Exceções

Introdução

- Os sistemas de hardware detectam condições de erro como **transbordamento (overflow)** de ponto flutuante.
- Nas primeiras linguagens, o programa **não podia detectar nem tratar** esses erros.
- Quando um erro ocorria, o **programa era terminado** e o controle passava ao sistema operacional.
- A reação típica: o sistema mostrava uma **mensagem de diagnóstico** e encerrava a execução.

Introdução

- Em operações de E/S, o comportamento era diferente.
- O Fortran permitia interceptar erros com a cláusula **Err** e detectar fim de arquivo com **End**.
- Exemplo:

```
Read(Unit=5, Fmt=1000, Err=100, End=999) Weight
```

- Err=100: transfere controle para o rótulo 100 em caso de erro.
- End=999: transfere controle para o rótulo 999 ao chegar ao fim do arquivo.
- Assim, Fortran tratava **erros e término de arquivo** com o mesmo mecanismo.

Introdução

- Alguns erros não são detectáveis por hardware, como **erro de faixa de índice de matriz**.
- Linguagens como **Java** exigem que o compilador verifique esses erros em tempo de execução.
- Em **C**, os índices não são verificados, acreditava-se que o custo não compensava o benefício.
- Compiladores modernos podem permitir **ativar ou desativar** essa verificação.
- A maioria das linguagens contemporâneas inclui **mecanismos de tratamento de exceções**, permitindo que programas reajam a erros e eventos inesperados sem encerrar a execução.

Conceitos Básicos

- Erros detectados por hardware (como leitura de disco ou fim de arquivo) são considerados **exceções**.
- O conceito é estendido para incluir **erros detectáveis por software**, sejam eles:
  - detectados por um **interpretador de software**,
  - ou pelo **código do usuário**.
- Assim, **exceção** é qualquer evento **não usual**, **errôneo ou não**, detectável por hardware ou por software que possa exigir processamento especial.

Conceitos Básicos

- O **tratamento de exceção** é o processamento especial que ocorre quando uma exceção é detectada.
- Esse tratamento é feito por um **tratador de exceção** (ou **manipulador de exceção**).
- Uma exceção é **levantada (raised)** quando o evento ocorre.
  - Em linguagens baseadas em C, diz-se que a exceção é **lançada (thrown)**.
- Diferentes tipos de exceções requerem **tratadores específicos**.
- Algumas linguagens encerram o programa exibindo uma **mensagem de erro**, enquanto outras **permitem interceptar e reagir** ao evento.

Conceitos Básicos

- O **tratamento de exceção** é o processamento especial que ocorre quando uma exceção é detectada.
- Esse tratamento é feito por um **tratador de exceção** (ou **manipulador de exceção**).
- Uma exceção é **levantada (raised)** quando o evento ocorre.
  - Em linguagens baseadas em C, diz-se que a exceção é **lançada (thrown)**.
- Diferentes tipos de exceções requerem **tratadores específicos**.
- Algumas linguagens encerram o programa exibindo uma **mensagem de erro**, enquanto outras **permitem interceptar e reagir** ao evento.

Conceitos Básicos

- Linguagens sem tratamento de exceções embutido dependem de **estratégias alternativas**, como:
  - uso de **variáveis auxiliares** para indicar erro,
  - passagem de **rótulos** como parâmetros,
  - ou definição de um **subprograma tratador** separado.
- Exemplo em C

```
int divide(int a, int b) {
   if (b == 0)
      return -1;
   return a / b;
}

int main() {
   int r = divide(10, 0);
   if (r == -1)
      printf("Erro: divisão por zero\n");
   else
      printf("Resultado = %d\n", r);
}
```

Conceitos Básicos

- Exemplo de verificação manual de erro de faixa (pseudocodigo):
  - Subprograma que faz várias referências a elementos de uma matriz

```
if (row >= 0 && row < 10 && col >= 0 && col < 20)
  sum += mat[row][col];
else
  System.out.println("Index range error on mat, row = " +
                     row + " col = " + col);
```

- Sem tratamento automático, o código precisa incluir verificações para cada possível erro.

Conceitos Básicos

Vantagens do tratamento de exceções

- Evita código redundante e facilita a **propagação** de erros.
- Uma exceção levantada em uma unidade pode ser **tratada em outra**, permitindo **reuso de tratadores** e **redução de complexidade**.
- Linguagens com suporte a exceções encorajam o programador a **prever e tratar eventos inesperados**.

Questões de Projeto

- O sistema de tratamento de exceções de uma linguagem deve permitir:
  - Exceções **predefinidas**
    - levantadas automaticamente pelo sistema
  - Exceções **definidas pelo usuário**
    - levantadas explicitamente no código
  - E **tratadores de exceções**.

Questões de Projeto

- Exemplo de exceção implicitamente levantada:

```
void example() {
  . . .
  average = sum / total;
  . . .
  return;
/* tratadores de exceção */
  when zero_divide {
    average = 0;
    printf("Error-divisor (total) é zero\n");
  }
  . . .
}
```

- A exceção de divisão por zero faz com que o controle seja transferido para o tratador apropriado, que é então executado.

Questões de Projeto

- A primeira questão de projeto diz respeito à **forma de vincular uma exceção a um tratador**.
- Esse vínculo ocorre em **dois níveis**:
  - **No nível local:** um tratador é associado a uma exceção levantada em uma sentença específica.
  - **No nível da unidade:** um mesmo tratador pode tratar exceções levantadas em diferentes pontos da unidade.
- Por exemplo, uma função pode conter vários pontos onde ocorrem divisões por zero. Um tratador associado a apenas um desses pontos não seria adequado para todos os casos.
- Assim, deve ser possível **vincular exceções específicas a tratadores apropriados**, mesmo que a mesma exceção possa ser levantada por diferentes sentenças.

Questões de Projeto

- Após a execução do tratador de exceção, é preciso decidir **onde a execução continuará**.
- As opções de projeto incluem:
  - **Continuação:** o programa retoma a execução após o ponto onde ocorreu a exceção;
  - **Término:** a execução da unidade (ou do programa) é encerrada.
- Essa decisão é chamada de **questão da continuação ou reinício**.
- A **Figura 14.1** mostra o fluxo de controle típico:
  - Uma exceção é levantada → o controle é transferido para o tratador → após a execução do tratador, o controle pode continuar ou terminar.

Questões de Projeto

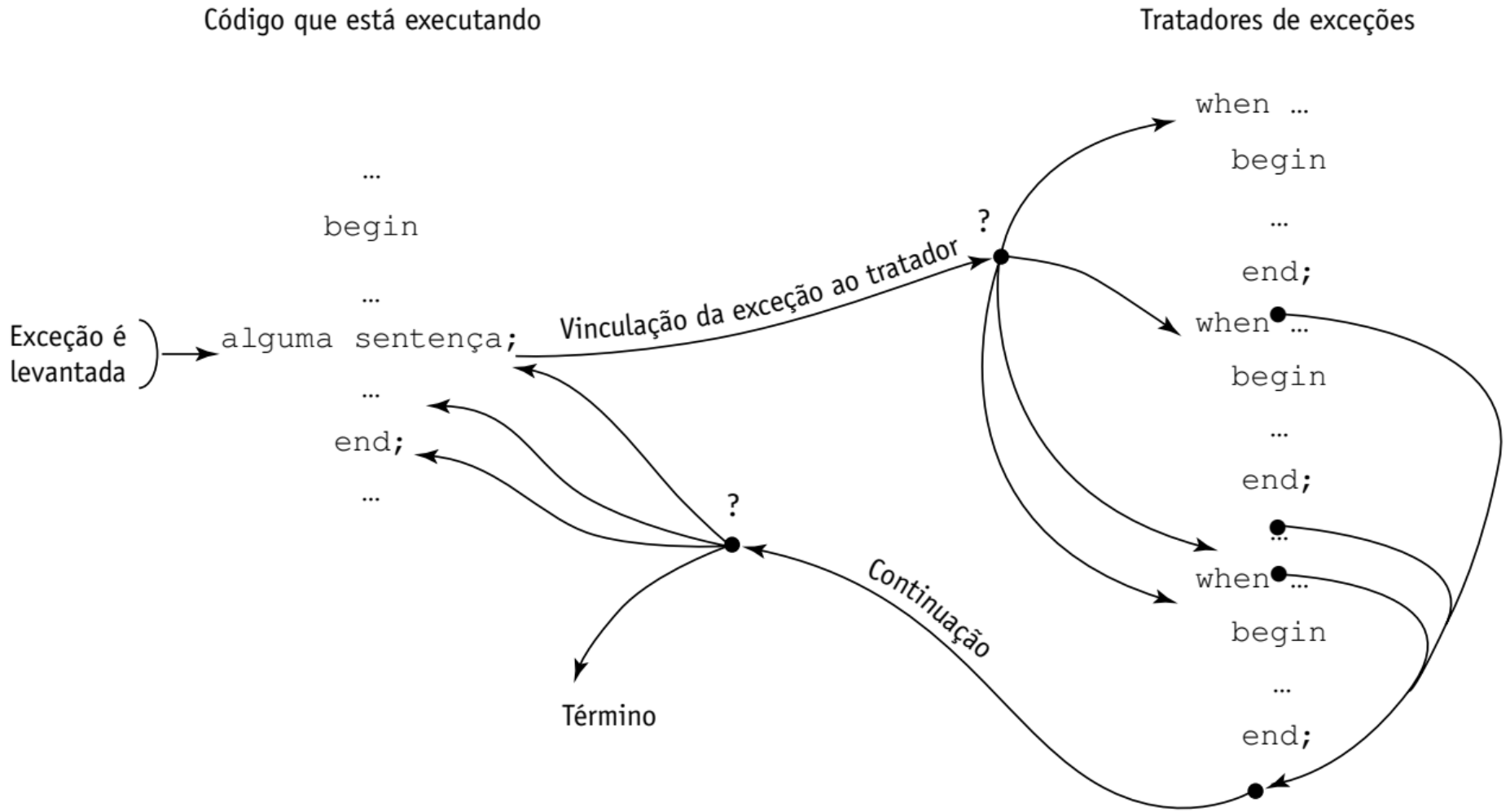

FIGURA 14.1
Fluxo de controle de tratamento de exceções.

Questões de Projeto

Nota Histórica

- **PL/I (ANSI, 1976)** foi pioneira no conceito de permitir que o usuário definisse exceções. A linguagem introduziu o conceito de exceções definidas pelo usuário, que podiam ser tratadas da mesma forma que as exceções predefinidas.
- O PL/I também permitia associar tratadores de exceções a uma lista de diferentes exceções possíveis.
- Desde o PL/I, grande parte do trabalho de projeto de linguagens subsequentes se concentrou em melhorar e generalizar esses mecanismos de tratamento.

Questões de Projeto

- Como e onde os tratadores de exceções são especificados e qual o seu escopo?
- Como uma ocorrência de uma exceção é vinculada a um tratador de exceção?
- A informação acerca de uma exceção pode ser passada para o tratador?
- Onde a execução continua, se é que continua, após um tratador de exceção completar sua execução?

Questões de Projeto

- Alguma forma de finalização é fornecida?
- Como as exceções definidas pelo usuário são especificadas?
- Se existem exceções predefinidas, devem existir tratadores de exceção padronizados para programas que não forneçam os seus próprios?
- As exceções predefinidas podem ser explicitamente levantadas?
- Os erros detectáveis por hardware são considerados exceções que devem ser tratadas?
- Existe alguma exceção predefinida?

Tratamento de exceções em C++

- O tratamento de exceções foi incorporado ao **C++** em 1990 e padronizado posteriormente.
- O projeto foi influenciado por linguagens como CLU, Ada e ML.
- Em C++, as exceções podem ser:
  - **Definidas pelo usuário** ou
  - **Levantadas em bibliotecas padrão**.
- O mecanismo de tratamento é composto por:
  - **Blocos try:** delimitam o código que pode gerar exceções.
  - **Blocos catch:** tratam as exceções levantadas.

Tratamento de exceções em C++

- A forma geral de um bloco try-catch em C++:
- Cada **função catch** é um tratador de exceção.
- Um tratador pode ter **um parâmetro formal**, semelhante ao parâmetro de uma função.
- O parâmetro formal indica **o tipo da exceção** ou armazena **informações associadas**.

```
try {
//** Código que pode levantar uma exceção
} catch (formal parameter) {
//** Corpo de um tratador
}
. . .
Catch (formal parameter) {
//** Corpo de um tratador
}
```

Tratamento de exceções em C++

- O parâmetro formal do catch pode ter diferentes formas:
- **Reticências (...):** capturam **qualquer exceção** (tratador genérico).
- **Tipo específico:** como float, int, ou **tipo definido pelo usuário**.
- **Tipo específico com nome de variável:** armazena informações sobre o erro.

Tratamento de exceções em C++

- Exemplo
- O primeiro catch trata uma exceção específica (string).
- O segundo catch(...) captura **todas as exceções** não tratadas anteriormente.
- Os tratadores podem conter **qualquer código C++ válido**.

```
try {
   if (b == 0)
      throw "Divisão por zero";
   result = a / b;
}
catch (const char* msg) {
   cout << "Erro detectado: " << msg << endl;
}
catch (float) {
   cout << ”Exceção do tipo float" << endl;
}
catch (...) {
   cout << "Erro desconhecido!" << endl;
}
```

Tratamento de exceções em C++

- Exceções em C++ são levantadas apenas por meio da sentença explícita **throw**, cuja forma geral em EBNF é a seguinte:

throw [expressão];

- A **expressão** é opcional.
  - Se omitida, o throw **relança** a exceção atual, permitindo tratá-la em outro lugar.
  - Um **throw** sem um operando pode aparecer apenas em um **tratador**.

Tratamento de exceções em C++

- Exemplo:

```
try {
   if (b == 0) throw "Divisão por zero";
   result = a / b;
}
catch (const char* msg) {
   cout << "Erro: " << msg << endl;
   throw; // relança para tratamento externo
}
```

- O comando throw **encerra a execução** da cláusula try e transfere o controle para o tratador adequado.

Tratamento de exceções em C++

- O tipo da **expressão em throw** determina **qual tratador será escolhido**.
- O compilador busca o **tratador mais específico** cujo parâmetro **“case” (combine)** com o tipo da exceção.
- A busca por tratadores segue **ordem sequencial** na lista de catch.
- Assim que ocorre o **primeiro casamento perfeito**, o restante é ignorado.
- Por isso, tratadores **mais específicos** vêm primeiro, seguidos pelos **mais genéricos**.

Tratamento de exceções em C++

```
catch (DivZeroError e) { ... }    // mais específico
catch (Error e) { ... }           // mais genérico
catch (const char* msg) { ... }   // mensagens textuais
catch (...) { ... }               // captura tudo
```

- O último tratador (**catch (...)**) captura **qualquer exceção**.
- Se nenhum tratador adequado for encontrado:
  - A exceção é **propagada** para a função que chamou a atual.
  - O processo continua até encontrar um tratador compatível.
  - Caso nenhum seja encontrado o programa é encerrado.

Tratamento de exceções em C++

Continuação da execução

- Após o tratador (catch) concluir sua execução, o controle **volta para a primeira sentença após o bloco try**.
- Essa sentença representa a **continuação normal do programa** após o tratamento da exceção.

Tratamento de exceções em C++

Outras escolhas de projeto em C++

- O C++ trata **apenas exceções definidas pelo usuário**.
  - Não há exceções predefinidas na linguagem.
- Existe um **tratador padrão** chamado unexpected,
  - executado quando uma exceção não é capturada;
  - sua ação padrão é **encerrar o programa**
- Apesar de C++ não ter exceções predefinidas, as bibliotecas padrão da linguagem definem exceções úteis:
  - out_of_range (acesso inválido a vetor),
  - overflow_error (erro matemático),

Tratamento de exceções em C++

Exemplo

- Demonstrar o uso de **tratadores de exceção simples** em C++.
- O programa lê uma sequência de notas (inteiros) e calcula uma **distribuição de frequência** das notas por categorias (0-9, 10-19, ..., 90-100).
- A entrada termina com um número negativo: **gera exceção NegativeInputException**.
- Notas inválidas (acima de 100) são tratadas por outro **catch interno**

Tratamento de exceções em C++

- Uma **limitação** do C++ é **não possuir exceções predefinidas** detectáveis por hardware.
  - O programador precisa **definir suas próprias exceções**.
- As exceções são associadas aos **tratadores (catch)** com base no **tipo de parâmetro formal**.
- O uso de **tipos genéricos** ou **não significativos prejudica a legibilidade**.
- A boa prática é **criar classes de exceções nomeadas** e organizá-las em uma **hierarquia clara**, tornando o código mais compreensível e modular.
- O parâmetro da exceção pode **carregar informações úteis** para o tratador.

Cláusula finally

- Em algumas situações, o programa precisa executar um código **independente de exceções**, como liberar arquivos, conexões ou memória.
- A cláusula **finally** garante que esse código **sempre será executado**, mesmo que ocorra exceção ou return.
- **Estrutura geral:**

```
try {
   . . .
}
catch (. . .) {
   . . .
}
. . . //** Mais tratadores
finally  {
   . . .
}
```

Cláusula finally

- Exemplo de uso em Java:
- Se houver **exceção tratada** → finally roda **após o catch**.
- Se houver **exceção não tratada** → finally roda **antes da propagação** da exceção.
- Se **não houver exceção** → finally roda **após o try**.

```
try {
   // leitura do arquivo
   FileReader file = new FileReader("dados.txt");
} catch (IOException e) {
   System.out.println("Erro ao ler arquivo.");
} finally {
   file.close(); // sempre executado
}
```

Sentença assert

- A **sentença assert** é usada para **programação defensiva**.
- Ela permite verificar se uma condição esperada é verdadeira durante a execução.
- Caso a condição seja falsa, o programa lança uma exceção AssertionError.
- Formas possíveis:

```
assert condição;

assert condição : expressão;
```

Sentença assert

- Exemplo em Java:

```
int idade = -5;
assert idade >= 0 : "Idade inválida!";
```

- São úteis durante o **teste e depuração**.
- Podem ser desativadas sem alterar o código, economizando verificações em produção.

Tratamento de exceções em Python

- Em Python, **todas as exceções são objetos**.
- A classe base de todas as exceções é **BaseException**, da qual **Exception** é derivada.
- Exceções **predefinidas** derivam de Exception.
- Exceções **definidas pelo usuário** também devem derivar de Exception.
- Principais subclasses predefinidas:
  - ArithmeticError: inclui OverflowError, ZeroDivisionError, FloatingPointError
  - LookupErro: inclui IndexError, KeyError

Tratamento de exceções em Python

- A forma geral de tratamento de exceções em Python é:

```
try:
   O bloco try (a faixa de sentenças onde exceções vão ser observadas)
except Exception1:
   Tratador para Exception1
except Exception2:
   Tratador para Exception2
...
else:
   O bloco else (o que fazer quando nenhuma exceção é levantada)
finally:
   O bloco finally (o que deve ser feito independentemente do que aconteceu)
```

Tratamento de exceções em Python

- As cláusulas **else** e **finally** são opcionais.
- A cláusula **else** é executada **somente se nenhuma exceção for levantada**.
- A cláusula **finally** é executada **em qualquer caso**, inclusive após exceções.

Tratamento de exceções em Python

- Se **nenhum tratador for encontrado**, a exceção é **propagada** para o bloco try mais externo.
- Se ainda assim não houver tratador, o programa **termina** com uma mensagem de erro e rastreamento (stack trace).
- Exemplo de tratador universal:

except Exception as ex_obj:

print(ex_obj)

- Esse tratador captura **todas as exceções derivadas de Exception**.
- O objeto de exceção pode armazenar **mensagens** ou **dados adicionais** úteis para o tratamento.

Tratamento de exceções em Python

- O lançamento de exceções em Python usa a palavra **raise**, equivalente a throw em C++/Java.

```
raise IndexError
# ou
raise IndexError("índice fora do intervalo")
```

- **raise** cria implicitamente uma instância da exceção nomeada.
- A exceção pode carregar informações úteis passadas como argumento (ex: mensagem de erro).

Tratamento de exceções em Python

- A sentença **assert** é usada para **programação defensiva**, garantindo que certas condições sejam verdadeiras.
- Se a condição for falsa, uma exceção AssertionError é levantada.

assert condição

assert condição, expressão

- Exemplo:

```
x = -5
assert x > 0, "x deve ser positivo”
```

- Se x <= 0, o programa lança:

```
AssertionError: x deve ser positivo
```

Tratamento de exceções em Python

- Internamente, o assert equivale a um **teste condicional**:

```
if __debug__:
    if not test:
        raise AssertionError(data)
```

- __debug__ é **True** por padrão.
- Quando o programa é executado com a opção -O:
  - ex: python -O programa.py
  - o Python **desativa** todas as sentenças assert.
  - Isso permite remover verificações de teste e manter o programa mais leve em produção.

Tratamento de exceções em Java

- O tratamento de exceções em Java é **baseado no modelo de C++**, mas foi projetado para estar mais alinhado com o paradigma de linguagem **orientada a objetos**.
- Java inclui **muitas exceções predefinidas** que podem ser lançadas **automaticamente** em tempo de execução.
- As exceções em Java são **objetos**, todas descendem da classe **Throwable**.

Tratamento de exceções em Java

- Throwable possui duas subclasses principais:
  - **Error**: erros graves do sistema (ex.: falhas de memória, JVM, etc.)
  - **Exception:** condições que o programa pode tratar.
- Exceções do tipo **Exception** podem ser:
  - **Verificadas (checked):** exigem tratamento explícito (IOException, SQLException).
  - **Não verificadas (unchecked):** herdam de RuntimeException (NullPointerException, IndexOutOfBoundsException).
- Usuários podem **criar suas próprias exceções** herdando de **Exception**.

Tratamento de exceções em Java

- Criando uma exceção personalizada
- O parâmetro message é passado à superclasse Exception.
- Pode ser recuperado com getMessage() ou printStackTrace().

```
class MyException extends Exception {
    public MyException() {}
    public MyException(String message) {
        super(message);
    }
}

// Lançamento da exceção
throw new MyException("Mensagem de erro personalizada");
```

Tratamento de exceções em Java

Clausula throws

- Usada para **declarar que um método pode lançar exceções**.
- Indica ao compilador que a exceção será tratada pelo **chamador**.
- Exemplo:

```
void readFile() throws IOException {
    // código que pode lançar IOException
}
```

- Se um método **chama outro** que lança uma exceção verificada, ele:
  - Deve tratá-la com **try/catch**, ou
  - Propagá-la com **throws**.

Tratamento de exceções em Java

- Exemplo notas similar ao exemplo em C++

Tratamento de exceções em Java

- O modelo de **tratamento de exceções em Java** é uma **melhoria** em relação ao C++.
- **Somente objetos** que sejam instâncias de Throwable (ou suas subclasses) podem ser lançados.
  - Evita lançar valores arbitrários, tornando o sistema mais seguro.
- A **cláusula throws** obriga o programador a **declarar** exceções verificadas.
  - O leitor do método sabe **quais exceções podem ocorrer**.
  - O compilador garante que exceções verificadas sejam tratadas ou declaradas.

Tratamento de exceções em Java

- A **cláusula finally** é uma **adição útil**, permitindo executar código de **limpeza** independentemente de como o bloco try termina.
- O **tempo de execução do Java** lança automaticamente várias exceções predefinidas que podem ser tratadas por qualquer programa do usuário:
  - IndexOutOfBoundsException
  - NullPointerException
  - ArithmeticException
- **C#** utiliza um modelo semelhante, mas **não possui** a cláusula throws.

Tratamento de exceções em Ruby

- Em Ruby, exceções são objetos.
- A maioria das exceções tratáveis herda de StandardError (que herda de Exception).
- Exemplos de subclasses de StandardError:
  - ArgumentError, IndexError, IOError, ZeroDivisionError.
- Toda exceção tem:
  - message (mensagem legível),
  - backtrace (rastreamento da pilha).

Tratamento de exceções em Ruby

- Levantar exceção: **raise**
  - **raise** "bad parameter" cria um RuntimeError com essa mensagem.
  - **raise** TypeError, "Float parameter expected" levanta a classe e a mensagem.
- Estrutura básica de tratamento:

```
begin
  # bloco que pode levantar exceções
rescue ExceptionClass => e
  # tratador específico
rescue AnotherClass
  # outro tratador
else
  # executa se nenhuma exceção foi levantada no begin
ensure
  # executa sempre (equivalente ao finally)
end
```

Tratamento de exceções em Ruby

- Diferença importante :
  - rescue posiciona os tratadores,
  - else executa quando **nenhuma** exceção ocorre,
  - ensure executa **sempre**, com ou sem exceção (limpeza/fechamento).
- Ruby permite **retry** no final do tratador para **tentar novamente** o bloco que falhou.

Tratamento de exceções em Ruby

- Exemplo notas similar ao exemplo em C++

Introdução ao tratamento de eventos

Ideia geral de tratamento de eventos

- **Evento**: notificação de que “algo ocorreu” (ex.: clique de mouse, tecla, envio de formulário).
- **Tratador de evento** (handler/listener): segmento de código executado **em resposta** ao evento.
- Em GUIs (e páginas Web), usuários **disparam** eventos; o programa reage **assíncronamente**.
- Difere do fluxo “convencional”: a ordem da execução passa a depender das **ações do usuário**.

Introdução ao tratamento de eventos

Onde aparecem e como ligamos handlers

- Em GUIs, widgets (botões, campos de texto, etc.) **emitem** eventos.
- Ligação (binding) = **registrar** um tratador no widget:
  - Java/Swing: componente.addXxxListener(tratador).
  - C#/WinForms: controle.Evento += Handler;.
  - Web: element.addEventListener("click", handler).
- Ao ocorrer o evento, o **runtime** invoca o handler com dados do evento (ex.: qual botão foi selecionado).

Introdução ao tratamento de eventos

Exemplo em Java e Javascript

- Tela com um campo de texto e **botões do tipo radio** para escolher o **estilo de fonte**:
  - “Plain”, “Bold”, “Italic”, “Bold+Italic”.
- O handler do grupo de botões **ajusta** a fonte do campo de texto de acordo com a seleção atual.

Paradigmas de Linguagens de Programação

Tratamento de Exceções