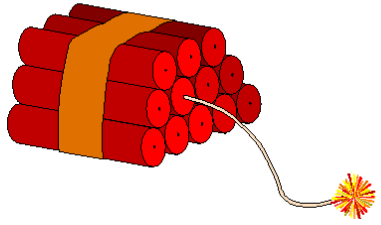


STE: Sistema de Tratamento de Exceções

Disciplina de Modelos de Linguagens de Programação

Tópicos

- Problemática e contextualização
  (qual a real necessidade de um STE?)
- Sistemas de Tratamento de Exceções
  (conceitos, processo, etc.)
- Tratamento local versus não-local
  (processo e diferenças)
- Tipos de exceções
  (checadas e não-checadas; pré-definidas e definidas pelo usuário/programador)

Programas sem tratamento de exceções

O código de tratamento de erros normalmente fica entrelaçado com o código normal:

```
public static void main(String args[]){
   if(args.length != 3){
      System.out.println("Sintaxe: <string> <string> <int>");
      System.exit(0);
   }
   if(intOk(args[2]) == false){
       System.out.println("O 3o parâmetro deve ser um inteiro!");
       System.exit(0);
   }
```

- A manutenção é dificultada!
- Como minimizar estes problemas? Tratar exceções!

Exceções

- São problemas comuns de execução:
- Representam situações anormais ou inválidas durante o processo de execução e que requerem um tratamento:
  - falha na aquisição de um recurso (new, open...)
  - tentativa de fazer algo impossível (divisão por zero, índice inválido...)
  - outras condições inválidas (lista vazia, overflow...)
- Detectadas por HW (erro de leitura em disco...)
- Detectadas por SW (acesso a índice fora dos limites do array...)

Exceções

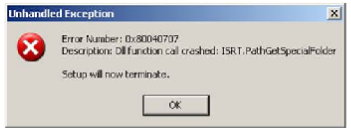
- Se a linguagem não der suporte ao seu tratamento?
  - Problemas detectados pelo HW:
    - normalmente causam o término do programa
  - Problemas detectados por SW:
    - componentes podem reagir:
      - terminando o programa;
      - retornando um valor de erro indicando falha;
      - retornando e ignorando o problema;
      - chamando uma função para tratar o erro, etc.

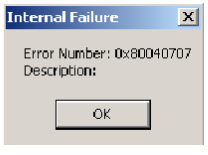
Exceções

- Mesmo assim, no caso anterior, há desvantagens:
  - **Terminando o programa:**
    - Mensagem de erro pode não ser claras
    - Programas em aplicações críticas?
  - **Retornando um valor de erro indicando falha:**
    - Nem sempre existe um valor que não será válido em uma execução sem erro
    - Valor de retorno deve ser SEMPRE verificado (eficiência, disciplina, legibilidade)
    - Funções de biblioteca do C, Windows API
  - **Retornando e ignorando o problema:**
    - Altamente não confiável
  - **Chamando uma função para tratar o erro:**
    - Quem chama? Qual função?

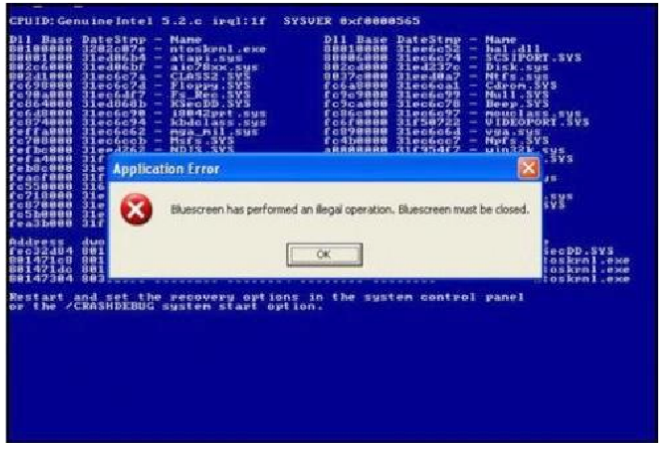
Exceções

- O que seu usuário diria/faria nos casos abaixo?





Exceções



Tratamento de exceções

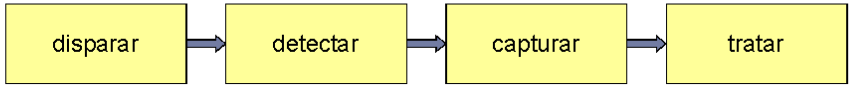
- Premissas:
  - Possíveis problemas de execução de um método podem ser antecipados pelo programador
  - Situações de erro podem ser revertidas
- Solução ideal:
  - Tratamento de problemas separado do código normal
  - Mecanismo: Sistemas de Tratamento de Exceções (STE)

Suporte a tratamento de exceções

- Permite **separar** as partes que devem executar normalmente daquelas que devem ser executadas apenas em caso de problemas
- Oferece legibilidade, confiabilidade, encapsulamento e reuso
- Em caso de problema, sub-rotina "levanta" ou indica a ocorrência de uma Exceção
- Seu tratamento pode ser feito em diferentes níveis, pela sub-rotina que tem condições de fazê-lo

Sistemas de tratamento de exceções (STE)

- Mecanismo oferecido pela LP (moderna)
- Um STE deve ser capaz de:
  - detectar a ocorrência de uma situação anormal de processamento
  - identificar o tipo de ocorrência e enviar um sinal ao programa (disparar uma exceção)
  - capturar a exceção em algum ponto do programa
  - tratar uma exceção em algum ponto do programa (ativar tratador)



Questões de projeto

- Como e onde os tratadores de exceção são especificados e qual o seu escopo?
- Como a ocorrência de uma exceção é vinculada a um tratador?
- Onde a exceção continua após ser tratada?
- Há exceções pré-definidas?
- Exceções pré-definidas podem ser ativadas explicitamente?
- Como as exceções definidas pelo usuário são especificadas?
- Deve haver tratadores padrão?
- Erros detectáveis por HW são tratados como exceções que podem ser manipuladas?
- Deve ser possível desativar exceções?

Exceções: tipos

- Pré-definidas (pela LP)
  - Ada, Modula-3, Java, ML
- Definidas pelo programador
  - Exemplo em Java:

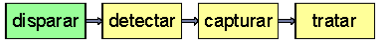
```
class ExceptionIdadeAluno extends Exception { };
```

OBS: em Java e C++ a exceção é um objeto (instância de uma classe)

Vamos definir algumas...

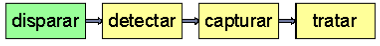
Exceções: observações

- Dependendo da LP, exceção pode ser um tipo primitivo, uma classe...
- Maioria das LPs permitem exceções parametrizadas (ou seja, o código que dispara a exceção pode passar parâmetros/dados para o código tratador)
- Em LPs onde exceções são definidas como classes:
  - atributos da classe contêm as informações
  - são inicializados no método construtor
  - em Java: podem receber uma mensagem e uma causa!



Disparo de exceções

- Duas maneiras: implicitamente ou explicitamente
  - **Implicitamente**, pelo ambiente operacional
    - Ocorrem quando o ambiente de execução não consegue executar um comando em um programa
    - Exemplos: todas as exceções predefinidas
      - divisão por zero
      - Overflow
      - erros de subscritos
      - acesso a conteúdo de ponteiro nulo



Disparo de exceções

- Duas maneiras: implicitamente ou explicitamente
  - **Explicitamente**, pelo programa
    - Ocorrem quando uma função do programa não consegue prosseguir a execução (quando o programador acredita haver um problema):

```
if(<problema>) throw <exceção>;
```

    - Exemplo em Java:

```
if (t==null) throw new NullPointerException();
```

Vamos disparar algumas...

disparar → detectar → capturar → tratar

Detecção de exceções

- Comando try
  - Indica a região do programa que deve ser monitorada pelo sistema de tratamento de exceções
  - Sintaxe:

```
try {<bloco_de_comandos>}
```

  - Exemplo:

```
try {
   idade = Integer.parseInt(stdin.readLine());
   valido = true;
}
```


disparar → detectar → capturar → tratar

Capturar e tratar exceções

- Comando catch:
  - captura um (determinado) tipo de exceção
  - implementa um tratador para aquele tipo de exceção
  - Sintaxe:

```
catch(<tipo><argumentos>){<comandos>}
```

  - Exemplo:

```
catch(NumberFormatException exc) {
  System.out.println("Entrada invalida!");
}
```


disparar → detectar → capturar → tratar

Um catch para cada tipo de exceção

```
try {
    num = Integer.parseInt(stdin.readLine());
    valido = true;
}
catch (NumberFormatException exc) {
    System.out.println("Entrada invalida!");
}
catch (IOException exc) {
    System.out.println("Problema de entrada. Terminando!");
    System.exit(0);
}
```


disparar → detectar → capturar → tratar

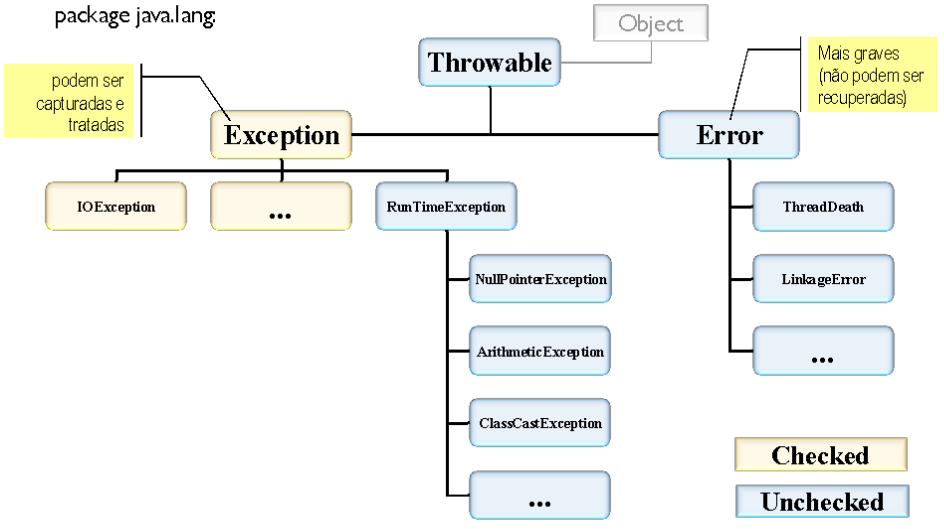
Objetivos do tratador

- Executar alguma operação que permita ao programa se recuperar do erro e continuar sua execução
- Se a recuperação não é possível, deve imprimir uma mensagem de erro significativa
- Se exceção não pode ser tratada no bloco, deve liberar recursos alocados localmente e propagar a exceção para o bloco chamador

Exceções em Java: observações

- **O tratamento pode ser obrigatório ou não:**
  - *unchecked exception* (exceções não-verificadas)
    - não precisam ser tratadas ou propagadas
  - *checked exception* (exceções verificadas)
    - se não tratadas, devem aparecer no cabeçalho
    - o compilador garante que quem usa o método deve tratá-lo!

Hierarquia de exceções em Java



Checked exceptions

- O compilador sempre "verifica" se há tratador para elas e acusa erro se não houver
- Comandos que disparam exceções verificadas devem ser colocados em um bloco *try*
- Úteis nas situações onde há interação com o usuário (as diretamente relacionadas com suas ações)
- Exemplos:
  - DataFormatException,
  - IOException,
  - NoSuchMethodException...

Observações gerais

- Tratar exceções cria um fluxo de controle separado do fluxo normal de execução (o fluxo dito excepcional)
- Fluxo de execução normal:

```
comando anterior;
try {
   num = Integer.parseInt(stdin.readLine());
   valido = true;
}
catch (NumberFormatException exc) {
   System.out.println("Entrada invalida.");
}
catch (IOException exc) {
   System.out.println("Problema de E/S. Terminando!");
   System.exit(0);
}
comando posterior;
```

>> 10

Observações gerais

- Tratar exceções cria um fluxo de controle separado do fluxo normal de execução (o fluxo dito excepcional)
- Fluxo de execução excepcional:

```
comando anterior;
try {
   num = Integer.parseInt(stdin.readLine());
   valido = true;
}
catch (NumberFormatException exc) {
   System.out.println("Entrada invalida.");
}
catch (IOException exc) {
   System.out.println("Problema de E/S. Terminando!");
   System.exit(0);
}
comando posterior;
```

>> AB

Observações gerais

- **O tratamento de uma exceção pode ser local ou não:**
  - tratar a exceção onde ela ocorre (local)
  - tratar a exceção em outro ponto do programa (propagação)
- Considerando que:
  - a unidade de execução pode detectar problemas mas geralmente não sabe como tratá-lo
  - a unidade requisitante não pode detectar problemas mas geralmente sabe como tratá-los
  - é conveniente saber quais são as exceções que um método pode disparar para providenciar um tratador
- Sub-rotina dispara uma exceção mas não a trata (em Java, a interface desta contém tal informação → checked exception)

Cláusula throws (Java)

- Cláusula throws: usada no cabeçalho de um método para indicar que ele propaga a exceção
- Exemplo:

```
public static void main(String[] a) throws Exception {
     int numerador = 10;
     int denominador = 0;
     if (denominador == 0) throw new Exception( );
     else System.out.println(numerador/denominador);
}
```

Observações gerais

Normalmente:

- se a exceção não é tratada dentro da rotina em execução, esta retorna abruptamente e a exceção é disparada no ponto de chamada
- se a exceção não é tratada na rotina chamadora, a propagação ocorre através do *encadeamento dinâmico*
- se atingir o programa principal sem tratamento, um tratador pré-definido é chamado e o programa termina

Propagação de exceções

- Para buscar o tratador de exceção:
  - percorre-se a 'cadeia' de ativações (*dynamic chain*), a partir do ambiente local (pilha de execução)
  - durante o caminho, são destruídos os objetos criados nos ambientes percorridos (retirados da pilha de execução)
  - uma vez tratada a exceção, o bloco imediatamente posterior ao do tratador é executado (não o do gerador da exceção)
  - se não for encontrado um tratador, a exceção chega ao método principal, que termina o programa

Propagação e *cleanup*

- Na busca pelo tratador adequado (na cadeia de ativações), os registros de ativação das sub-rotinas afetadas pela exceção são desempilhados (incluindo a restauração de registros que foram salvos na sequência de chamada da sub-rotina)
- C++:
  - Exceção que sai do escopo implica a chamada de métodos destruidores para objetos declarados naquele escopo:
    - **liberação de área do heap, fechamento de arquivos, etc.**
- Java:
  - construção *finally* é usada

Cláusula finally

- A cláusula finally
  - é utilizada para forçar a execução de um bloco de código
  - associada a um bloco try
  - pode ser utilizada com ou sem o bloco catch
- A cláusula finally é executada nas seguintes condições:
  - fim normal do método
  - devido a uma instrução return ou break
  - caso uma exceção tenha sido gerada

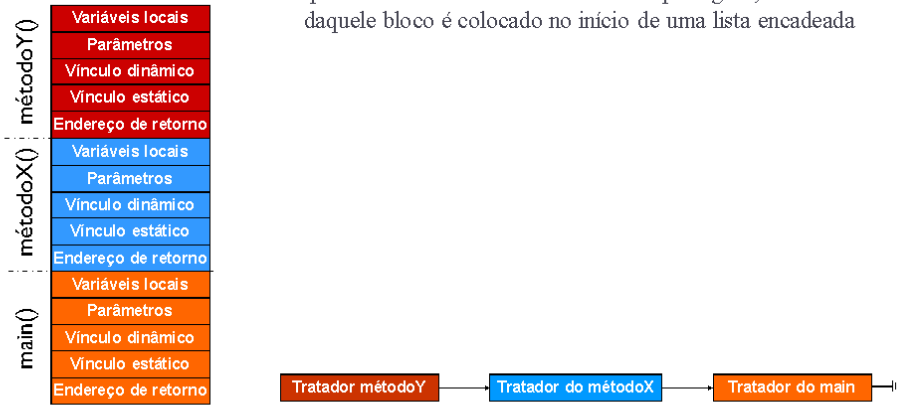
```
try { .... }
catch (Tipo1 exc) { .....}
catch (Tipo2 exc) {......}
finally {<bloco de comandos>}
```

Implementação de Exceções

1. Lista dinâmica
2. Tabela estática

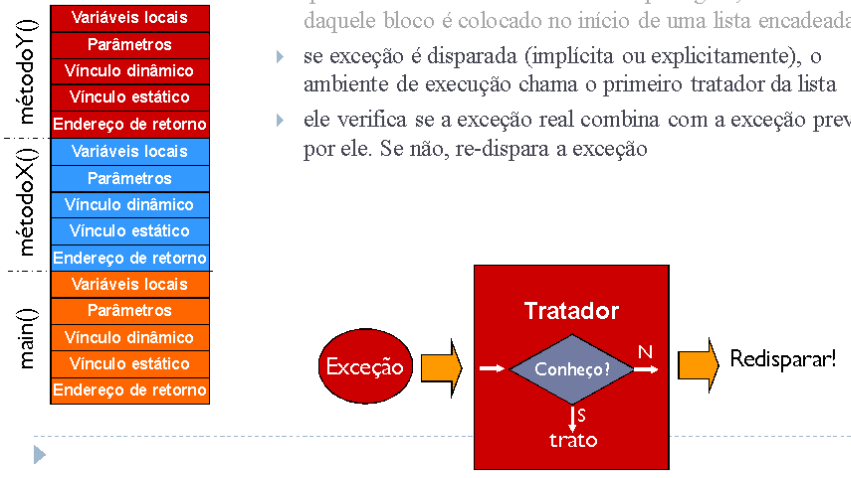
Implementação de Exceções (1)

- Através de lista dinâmica de tratadores:
  - quando o controle entra em um bloco protegido, o tratador daquele bloco é colocado no início de uma lista encadeada



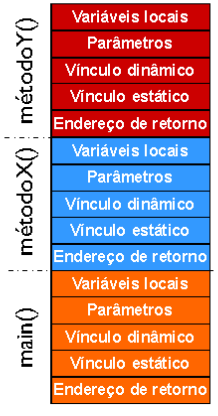
Implementação de Exceções (1)

- Através de lista dinâmica de tratadores:
  - quando controle entra em um bloco protegido, o tratador daquele bloco é colocado no início de uma lista encadeada
  - se exceção é disparada (implícita ou explicitamente), o ambiente de execução chama o primeiro tratador da lista
  - ele verifica se a exceção real combina com a exceção prevista por ele. Se não, re-dispara a exceção



Implementação de Exceções (1)

- Através de lista dinâmica de tratadores:
  - Quando um bloco prevê **vários tratadores** (várias cláusulas *catch*) ele é implementado como um único tratador com uma estrutura de seleção múltipla
  - Propagação: se dá pela *dynamic chain:*
    - Cada sub-rotina tem um tratador implícito que executa o epílogo daquela sub-rotina, acrescido de:
      - Remoção das rotinas de tratamento atual
      - Re-disparar a exceção
    - Desvantagem: acréscimo no custo da execução SEMPRE, mesmo que não haja exceções (empilha/desempilha tratadores de forma dinâmica!)



Implementação de Exceções (2)

- Construção estática da tabela (lista) de tratadores (em tempo de compilação)
- Embasamento:
  - O único objetivo da lista é determinar o tratador ativo em um dado momento
  - blocos de código normalmente ficam em posições contíguas
  - é possível montar estaticamente uma tabela de tratadores indexada pelo endereço inicial de um bloco protegido
- Cada entrada na tabela tem:
  - o endereço inicial de um bloco protegido
  - o endereço do tratador correspondente
- Tabela ordenada pelo endereço do bloco

Implementação de Exceções (2)

- Quando ocorre uma exceção, o ambiente de execução faz uma busca binária usando PC como índice
- Se o tratador re-disparar a exceção, o processo se repete: tratadores são blocos de código e também estarão na tabela
- Único cuidado:
  - tratadores implícitos associados com a propagação entre sub-rotinas
  - estes devem assegurar que, ao re-disparar a exceção, o índice de busca na tabela seja o endereço de retorno da sub-rotina, não o PC atual

Implementação de Exceções

- Custo de se tratar a exceção é maior na segunda abordagem
- Porém, ele só é pago SE houver exceção
  - o que deve acontecer poucas vezes ao longo da execução e de várias execuções
  - custo na execução normal é zero!
- Na segunda abordagem, compilador deve ter acesso a todo o programa
  - Linker deve intervir em LPs que usam compilação separada
  - Java: compilador cria tabelas separadas para cada sub-rotina e o *stack frame* contém um ponteiro para a tabela apropriada

Exceções em C (Visual C - Microsoft)

```
#include "windows.h"
void main(void) {
  int x, y;
  __try {
      x = 5;
      y = 0;
      x = x / y;
  } __except (GetExceptionCode() ==
                  EXCEPTION_INT_DIVIDE_BY_ZERO ?
                  EXCEPTION_EXECUTE_HANDLER :
                  EXCEPTION_CONTINUE_SEARCH ) {
     printf("Divide by zero error.\n");
  }
}
```

Exceções em C++

```
#include <iostream.h>
int divide(int x, int y) {
  if (y == 0) throw int();
  return x / y;
}
void main(void) {
  int x, y;
  try {
      x = 5;
      y = 0;
      x = divide(x, y);
  } catch (int) {
      cout << "A division by zero was attempted.\n";
  }
}
```

Bibliografia / leitura recomendada

- MacLaren, M. D. 1977. Exception handling in PL/I. In *Proceedings of An ACM Conference on Language Design For Reliable Software* (Raleigh, North Carolina, March 28 - 30, 1977). D. B. Wortman, Ed., 101-104. DOI= http://doi.acm.org/10.1145/390019.808316
- Definição de Exception Handling na Wikipedia (e seus links). http://en.wikipedia.org/wiki/Exception_handling
- Exceptions. In: Java Language Specification. http://java.sun.com/docs/books/jls/third_edition/html/exceptions.html#11.2
- Sebesta, R. W. Manipulação de exceções (Capítulo 14). In: Sebesta, R. W. Conceitos de Linguagens de Programação. 5a Ed. Porto Alegre: Bookman, 2003.
- Scott, Michael. Exception Handling (chapter 8.5). In: Scott, Michael. Programming language Pragmatics. San Diego, CA: Academic Press, 2000. pp.464-373.