Linguagem de Programação II

O mecanismo de exceção

Universidade do Estado do Rio de Janeiro-UERJ
Instituto de Matemática e Estatística-IME
Ciência da Computação
Professor: Alexandre Sztajnberg

Erros

- Não estamos falando de erros de compilação, erros de lógica, programação ruim ... Que não atende aos requisitos ou à especificação
  - Para estes tipos de erro, técnicas de engenharia de software, depuração, testes unitários, testes integrados ou testes de carga devem ser empregados
- Estamos falando de erros em tempo de execução. Temos dois "tipos"
  - Erros que acontecem, mas o programador não deveria deixar acontecer, deveriam ser antecipados e contidos. Exemplos:
    - Divisão por zero; Iteração por um número incorreto de vezes
    - Casting de objetos para classes não herdadas
    - Tentativa de acesso de elemento de *array* com índice inválido
  - Erros que podem ser previstos, mas não tem como antecipar ou conter ... "só na hora". Erros de E/S em geral estão neste "tipo". Exemplo:
    - Tentar abrir um arquivo que não existe, ou para o qual o usuário não tem permissão correta
    - Erro de acesso à rede, URL errada, DNS não resolve o nome, firewall bloqueia o acesso, time-out. URL errada, DNS com problema, erro de roteamento

Erros

- Os erros nos quais estamos interessados, aparecem na maioria das vezes durante a interação de dois "atores"
  - O cliente / chamador / invocador / requisitante, que
    - Prepara os argumentos ou parâmetros de entrada
    - Faz a chamada de uma rotina, procedimento ou método
    - Aguarda o servidor acabar
    - Recebe o controle de volta e, pode receber um resultado como retorno
  - O servidor / procedimento / método / implementação de rotina que:
    - É chamado para realizar o procedimento
    - Recebe os argumentos ou parâmetros de entrada
    - Realiza o procedimento, calcula o resultado (se assim for programado)
    - Retorna o controle para o chamador, e devolve o resultado (se assim for programado)

- **Mas, e onde temos erro nesta sequência?**
- **Não temos (ainda) ...**

Erros

- Se temos a possibilidade de erros, a sequência tem que prever isso!

- O cliente
  - Prepara os argumentos ou parâmetros de entrada
  - Tenta
    - Fazer a chamada de uma rotina, procedimento ou método
    - Aguarda o servidor acabar
  - Verifica se foi tudo bem
    - Recebe o controle de volta e, pode receber um resultado como retorno
  - Se houve erro, tem que ficar sabendo disso de alguma forma
    - E faz o que com essa informação de que houve erro? Trata? Aborta?

- O servidor
  - É chamado para realizar o procedimento
  - Recebe os argumentos ou parâmetros de entrada
  - Verifica os parâmetros de entrada // **Isso é detecção de erro**
    - Se estão errados, não dá para prosseguir! Sinaliza o erro para o cliente. // **Como?**
  - Se não houve erro de parâmetros, tenta
    - Realizar o procedimento, calcula o resultado (se assim for programado)
  - Verifica se foi tudo bem no procedimento // **Isso também é detecção de erro**
    - Se algo deu errado, não pode retornar uma resposta errada! Sinaliza o erro para o cliente. // **Como?**
  - Se tudo deu certo Retorna o controle para o chamador, e devolve o resultado (se assim for programado)

Erros

- Pontos a serem observados

- No cliente
  - O código pode ser bem feito e criticar os dados que serão enviados como parâmetros
    - Preciso deixar fazer *Integer. parseInt ("abdc")?*
    - Ou File.open ("null")?
  - Como saber se a chamada deu certo? Ou se houve erro?
  - Se houve erro
    - Que tipo de informação preciso
    - O que fazer com esta informação?
      - Tenho que corrigir? Tratar?
      - Simplesmente para o programa?
      - Aviso "ao chefe"?
- No servidor
  - Como detectar erros nos parâmetros de entreda?
    - If-then-else?
  - Como detectar erros ou falhas na rotina a ser realizada?
    - Posso monitorar o recurso sendo usado
      - Sim!
    - Posso tratar erros, se tiver que acessar outros serviços?
      - Sim!
  - Se um erro acontece, não posso deixar recursos ou variáveis inconsistentes
  - Como sinalizar erros?
    - True / False?
    - Código de erro?
    - E se erros diferentes forem detectados?

Como erro são tratados em C?

- Com frequência, em caso de erro durante a execução de uma primitiva (função do sistema ou biblioteca), o valor de retorno é igual a -1.
  - Neste caso, a variável global errno será atualizada e seu valor indicará um código de erro preciso. Este código de erro pode ser obtido através da função perror().
    - É necessário neste caso que o arquivo <errno.h> seja incluído do cabeçalho do programa para que errno e perror() possam ser utilizadas.
  - The strerror() function, which returns a pointer to the textual representation of the current errno value.

Como erro são tratados em C?

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
extern int errno ;
int main () {
  FILE * pf; int errnum;
  pf = fopen ("unexist.txt", "rb");
  if (pf == NULL) {
    errnum = errno;
    fprintf(stderr, "Value of errno: %d\n", errno);
    perror("Error printed by perror");
    fprintf(stderr, "Error opening file: %s\n", strerror( errnum ));
  } else {
    fclose (pf);
  }
  return 0;
}
```

- Exame do retorno para verificar se houve erro
- Acessando *errno para identificar o erro específico*

```
Value of errno: 2
Error printed by perror: No such file or directory
Error opening file: No such file or directory
```

Fork() em C

- O retorno da chamada fork() pode ser: 0 (processo filho), -1 (erro), > 1 (PID do processo filho, para o pai)
- E cada processo executa rotinas que podem resultar em erro. **Como tratar e manter o código elegante?**

```
pid=fork() ;
if(pid==-1) { /* erro */
   perror("impossivel de criar um filho");
   exit(-1) ;
} else if(pid==0) { /* filho */
   printf("Sou filho %d\n",getpid());
   for(i=1;i<=5;i++){
     if(read(fd,&c,1)==-1){
       perror("impossivel de ler");
       exit(-1) ;
     }
     printf("Eu li um %c\n", c);
   }
   [...]
} else {/* pai */
   // continuando
   else {/* pai */
     printf("Meu filho tem o PID %d\n",pid);
     while((r=read(fd,&c,1))!=0) {
       if(r==-1) {
         perror("impossivel de ler");
         exit(-1) ; r
       }
     }
     [...]
}
```

Voltando ao Java ...

- Argumentos representam uma séria “vulnerabilidade” para um objeto servidor.
  - Argumentos do construtor inicializam o estado.
  - Argumentos de método contribuem frequentemente com o comportamento.
- Verificar / criticar esses argumentos é uma forte medida defensiva.
- No exemplo do *LivroDeEndereços*, se verificarmos a chave de acesso à entrada, podemos contornar um dos problemas.

Verificando argumentos de entrada

- Antes de prosseguir, verifica os argumentos de entrada
- Verificar ou criticar, significa saber se tem as características necessárias para se realizar o procedimento. Ou seja *if-then-else* mesmo!!
- Se houve erro, não realiza o procedimento

```
public void removeDetalhes(String chave){
   if(chaveEmUso(chave)) {
      DetalhesContato detalhes = (DetalhesContato) livro.get(chave);
      livro.remove(detalhes.getNome());
      livro.remove(detalhes.getTelefone());
      numeroDeEntradas--;
   }
}
```

Verificando argumentos de entrada

- Mas e se houve erro? Fica assim?
- O usuário / cliente não é avisado? Entra “mudo” e sai “calado”?
- Observação, de uma vez por todas. Um método ou procedimento oferece um serviço. Por isso, de forma abstrata e coloquial, chamamos de “servidor”. Este método é chamado ... Quem chama / invoca / solicita é chamado, também de forma abstrata e coloquial, de cliente ou usuário. Não estamos falando de um usuário humano, ok?!

```
public void removeDetalhes(String chave){
   if(chaveEmUso(chave)) {
      DetalhesContato detalhes = (DetalhesContato) livro.get(chave);
      livro.remove(detalhes.getNome());
      livro.remove(detalhes.getTelefone());
      numeroDeEntradas--;
   }
}
```

Verificando argumentos de entrada

- Informar que o argumento é inválido. Para quem?

Verificando argumentos de entrada

- Informar que o argumento é inválido. Para quem?
  - Ao usuário? (Quais problemas impedem isso?)

Verificando argumentos de entrada

- Informar que o argumento é inválido. Para quem?
  - Ao usuário?
    - Existem usuários humanos?
    - Eles podem resolver o problema?

Verificando argumentos de entrada

- Informar que o argumento é inválido. Para quem?
  - Ao usuário?
    - Existem usuários humanos?
    - Eles podem resolver o problema?
  - Ao objeto cliente?

Verificando argumentos de entrada

- Informar que o argumento é inválido. Para quem?
  - Ao usuário?
    - Existem usuários humanos?
    - Eles podem resolver o problema?
  - Ao objeto cliente? (Sim, como?)

Verificando argumentos de entrada

- Informar que o argumento é inválido. Para quem?
  - Ao usuário?
    - Existem usuários humanos?
    - Eles podem resolver o problema?
  - Ao objeto cliente?
    - Retornando um valor de diagnóstico.
    - Lançando uma **excessão.**

Retornando um diagnóstico

- Agora o método não é mais void ... Retorna um boolean, com ... A informação de que foi tudo ok (true) ou não (false)

```
public boolean removeDetalhes(String chave){
   if(chaveEmUso(chave)) {
      DetalhesContato detalhes = (DetalhesContato) livro.get(chave);
      livro.remove(detalhes.getNome());
      livro.remove(detalhes.getTelefone());
      numeroDeEntradas--;
      return true;
    }
    else
      return false;
}
```

Retornando um diagnóstico

- Quais as possíveis respostas do cliente?

Retornando um diagnóstico

- Quais as possíveis respostas do cliente?
  - Testar o valor de retorno:
  - Ignorar o valor de retorno:

Retornando um diagnóstico

- Quais as possíveis respostas do cliente?
  - Testar o valor de retorno:
    - Possibilidade de recuperar o erro.
    - Evita a falha do programa.
  - Ignorar o valor de retorno:
    - Não pode ser evitado.
    - Possibilidade de levar a uma falha do programa.

Retornando um diagnóstico

- Quais as possíveis respostas do cliente?
  - Testar o valor de retorno:
    - Possibilidade de recuperar o erro.
    - Evita a falha do programa.
  - Ignorar o valor de retorno:
    - Se o programador não quiser fazer nada com o retorno, pode decidir ignorar ...
    - Isso não pode ser evitado pelo compilador, ignorar também vai estar sintaticamente correto.
    - Possibilidade de levar a uma falha do programa.
- Por isso, o uso de **exceções** é preferível.

Vantagens

- Exceções fornecem meios de separar os detalhes do tratamento dado quando algo fora do esperado acontece na execução de um programa

```
lerArquivo{
   Abrir arquivo;
   Tamanho do arquivo;
   Alocar memoria;
   Ler aquivo na memoria;
   Fechar arquivo;
}
```

Vantagens

- A primeira vista a função parece simples, mas ignora as seguintes possibilidades de erro

```
lerArquivo{
   Abrir arquivo;            // ... e se o arquivo não pode ser aberto?
   Tamanho do arquivo;       // ... e se o tamanho for não for determinado?
   Alocar memoria;           // ... e se a memória não pode ser alocada?
   Ler aquivo na memoria;// ... e se a leitura falhar?
   Fechar arquivo;           // ... e se o arquivo não puder ser fechado?
}
```

Vantagens

- Como já ilustramos antes, para lidar com esses casos, a função `lerArquivo` precisa ter mais código para detectar, manusear e, eventualmente, reportar os erros.
- Uma solução seria o uso de `ifs` e `elses` para lidar com isso, mas o código ficaria tão grande que o fluxo lógico seria perdido, sendo difícil dizer quando o código está fazendo a coisa certa: Será que o arquivo realmente fechou se a função falha em alocar memória o suficiente? **Vimos isso no exemplo do fork() em C.**
- Exceções permitem manter o fluxo do código e lidar com os casos excepcionais em outro lugar, com mais estrutura.

Exceções

- Princípios do mecanismo de exceções
  - É um recurso especial da linguagem e tem o suporte da JVM em tempo de execução
  - Nenhum valor de retorno "especial", de diagnóstico é retornado
    - ou a chamada retorna o resultado esperado,
    - ou o fluxo normal é interrompido e desviado para um bloco que captura a exceção
  - Existe uma classe de exceções, confirmadas (*checked*), que representam erros não podem ser ignorados pelo cliente
    - **Se ignorar da erro de compilação!**
  - Ações específicas de recuperação são encorajadas

Exceções

❑ No cliente

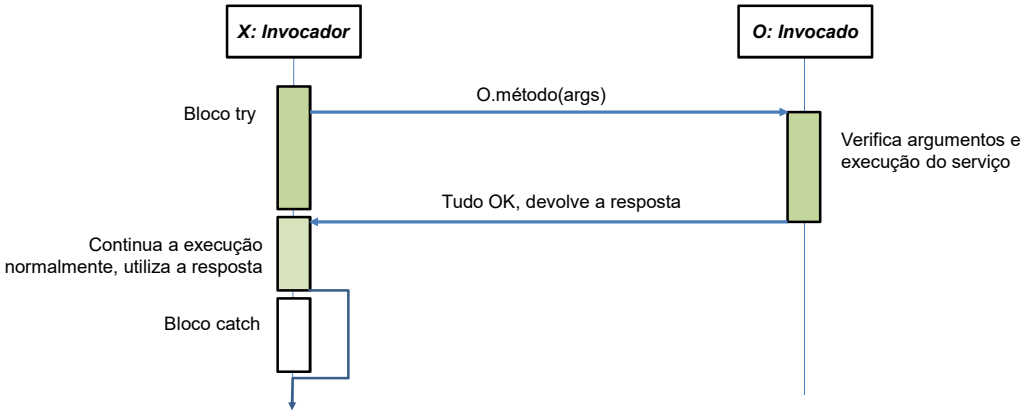
- As chamadas que podem dar erro devem ser contidas em o bloco `try{}`
  - No código podem haver vários blocos try-cacth
- Preparar o bloco `catch{}` para tratar o erro caso ele seja lançado
  - O bloco catch vem logo depois do bloco try
  - No bloco catch a classe de exceção capturada é informada como parâmetro de entrada
  - Pode haver mais de um bloco catch depois do bloco try, especializado em um tipo de exceção que pode ser lançado por um método chamado no bloco try

❑ No servidor

- Definir as condições de erro
- Se o método pode lançar uma ou mais exceções confirmadas, sinalizar com ***`throws`*** na assinatura
- Verificar os argumentos/parâmetros de entrada
- Se alguma das condições de erro for detectado nos argumentos ou na execução da rotina do método
  - Cria um objeto da classe **Exception** ou herdeiros
  - Este objeto pode conter informações para facilitar o tratamento do erro
  - Notifica o usuário, lançando (ou jogando) o objeto de exceção criado usando o comando ***`throw`***.

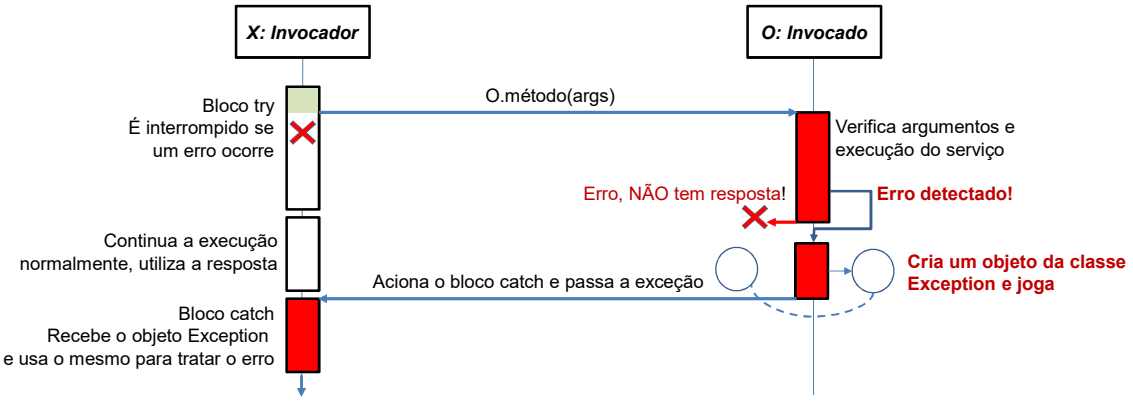
Mecanismo de exceções

❑ Fluxo normal



Tratamento de exceções

- Com um exceção, não executa a rotina normal, 'vai' para o bloco *catch*



Os efeitos de uma exceção

- No cliente
  - Até a exceção ocorrer, tudo o que foi executado no bloco try está valendo: chamadas a métodos, atribuições a variáveis, etc. **Se isso é um problema, dê um jeito!**
  - Nenhum valor de retorno é recebido. Se a chamada feita teve erro, nenhum resultado deveria retornar mesmo, ou seria inconsistente
  - Controle não retorna ao ponto da chamada do cliente.
    - Não pode prosseguir o fluxo normal de qualquer maneira, porque aquilo o que era esperado, não vai chegar!
  - Se existe um bloco catch que "casa" com a exceção jogada (*instanceOf*), ele é acionado
  - Caso contrário, ou o código "rejoga" a exceção, ou o programa é abortado.
- No servidor
  - Ao detectar um erro
    - Cria um objeto que herda de "*Exception*"
    - Lança este objeto para a JVM
      - A JVM é que vai encaminhar o objeto para o chamador, caso existe um bloco catch que "case" com o objeto de exceção
  - Neste ponto a execução do método termina prematuramente
    - nada mais é realizado
    - O registro de ativação do método sai da pilha de execução
  - Nenhum valor de resultado é retornado

JVM heap

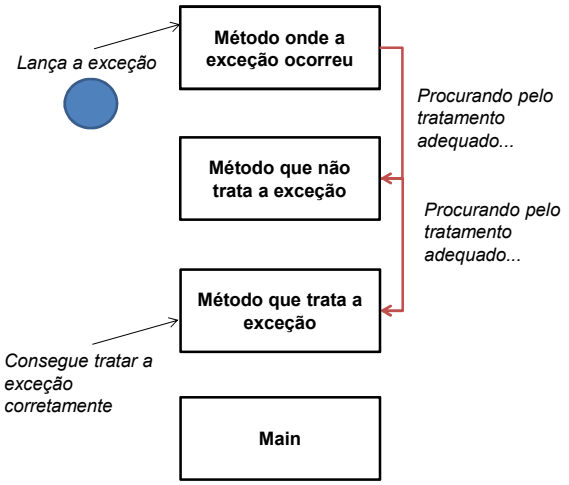
- Quando um método é chamado é criado uma estrutura de dados chamada registro de ativação.
  - O registro de ativação contém as variáveis locais, variável para retorno, passagem dos parâmetros e o endereço de retorno do chamador.
- Esse registro de ativação, que representa o método chamado e contem as variáveis locais do método, é empilhado para o topo da JVM heap, que nada mais é do que uma pilha de trabalho.
- Quando o método termina, no retorno, ele é desempilhado e o método chamador desse primeiro método assume o top da pilha.

JVM heap

- Isso também ocorre quando uma exceção é lançada.
- Mesmo o método não tendo terminado de forma correta, a exceção é jogada para JVM e o método sai da pilha, funcionando como um call stack.
  - Com a exceção “nas mãos” a JVM começa a “caçar” na pilha um método que esteja preparado para fazer a captura da exceção e depois o tratamento.
  - É verificado se o método possui o `catch` necessário para tratar a exceção.
    - Se o método não consegue tratar, o método é retirado do topo da pilha.
    - Se o método consegue tratar a execeção, ela é jogada para ele.
    - Existe também a opção do método conseguir capturar e rejogar a exceção.

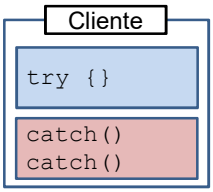
Depois de receber o objeto de exceção, a JVM ...

- O sistema de runtime da JVM começa a procurar um tratador/*handler* para a exceção (um bloco *try* que "case/match").
- Procura a partir do topo da pilha de chamadas ou heap (elemento interno da JVM que mantém o rastro de quem fez a última chamada de método e "quem chamou quem").
- Se não encontrar um tratador, o runtime termina (o programa é abortado e a JVM para)
- Informações da exceção são apresentadas.



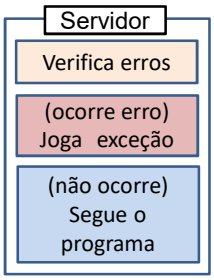
Cliente e servidor

- Cliente
  - Cliente ou chamador.
  - Chama o método que pode dar certo ou pode dar errado (nesse caso tem que capturar e tratar exceções).



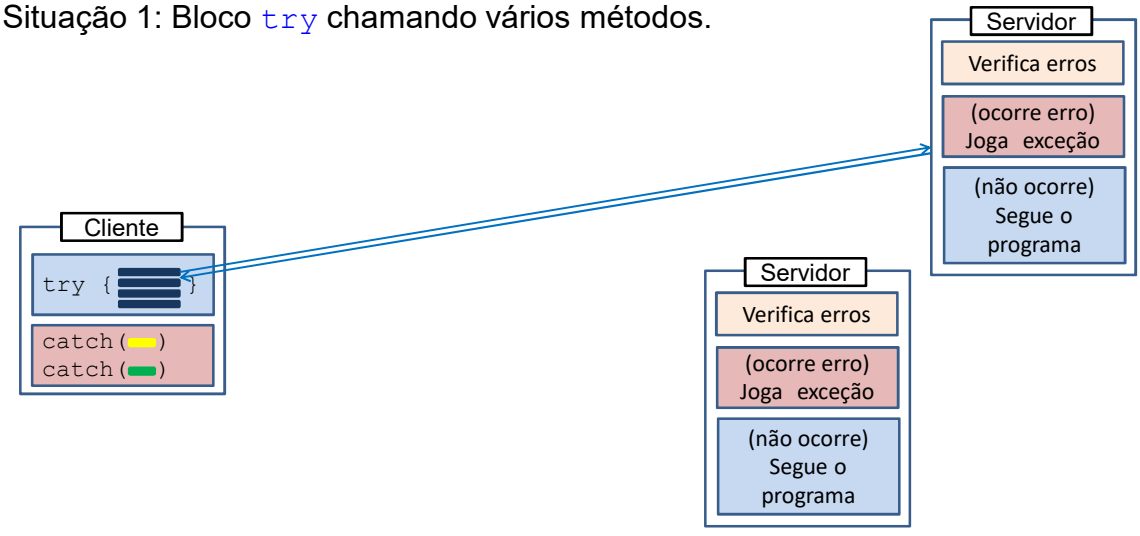
Cliente e servidor

- Servidor
  - Servidor, implementador ou método sendo chamado.
  - Precisa detectar o problema.
  - Se não houver erro nada no fluxo do programa é mudado.
  - Se ocorrer o erro não existe retorno do método chamado e o fluxo do programa é interrompido.



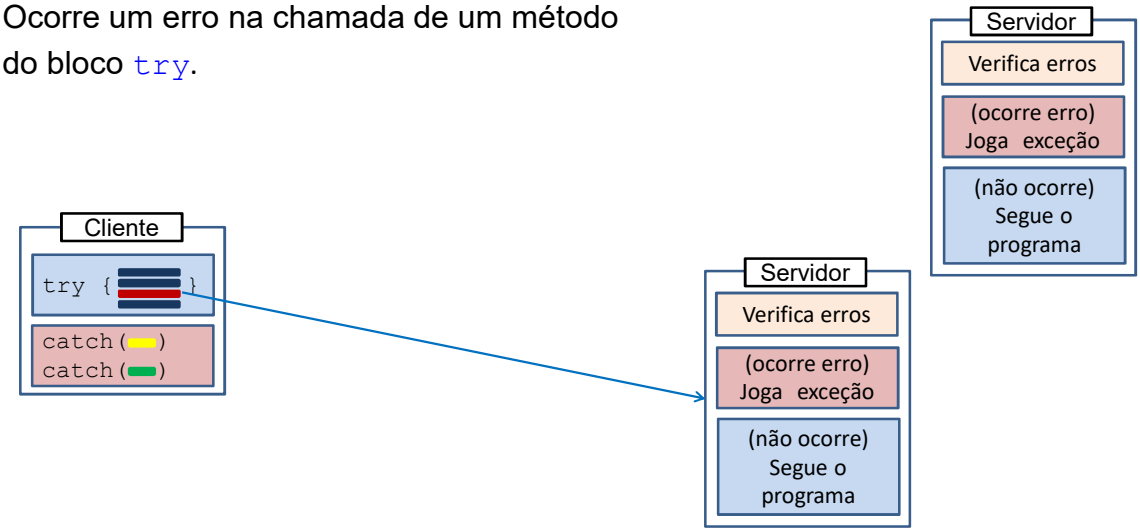
Cliente e servidor

- Situação 1: Bloco `try` chamando vários métodos.



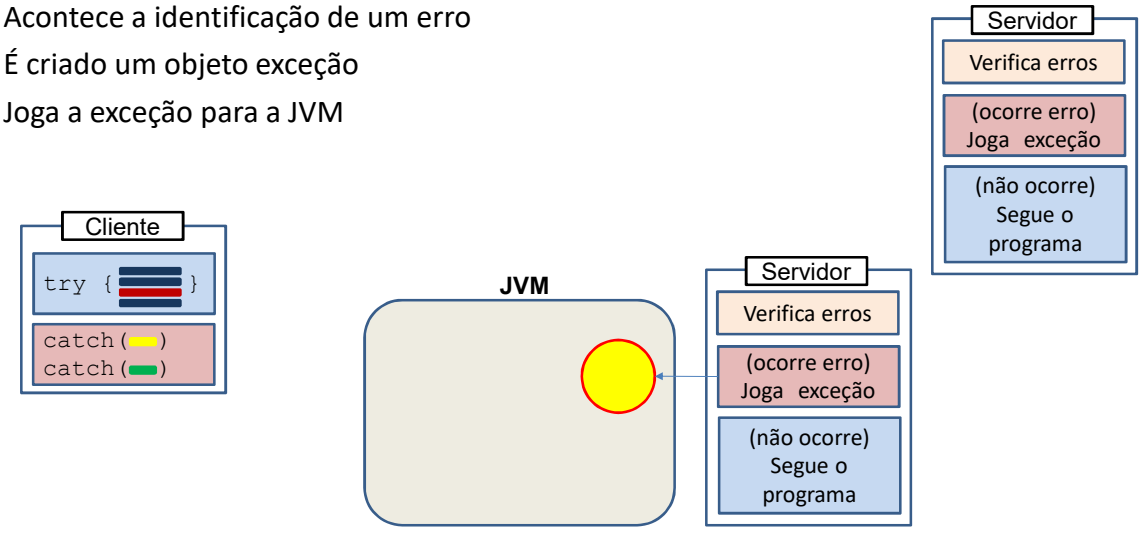
Cliente e servidor

- Ocorre um erro na chamada de um método do bloco `try`.



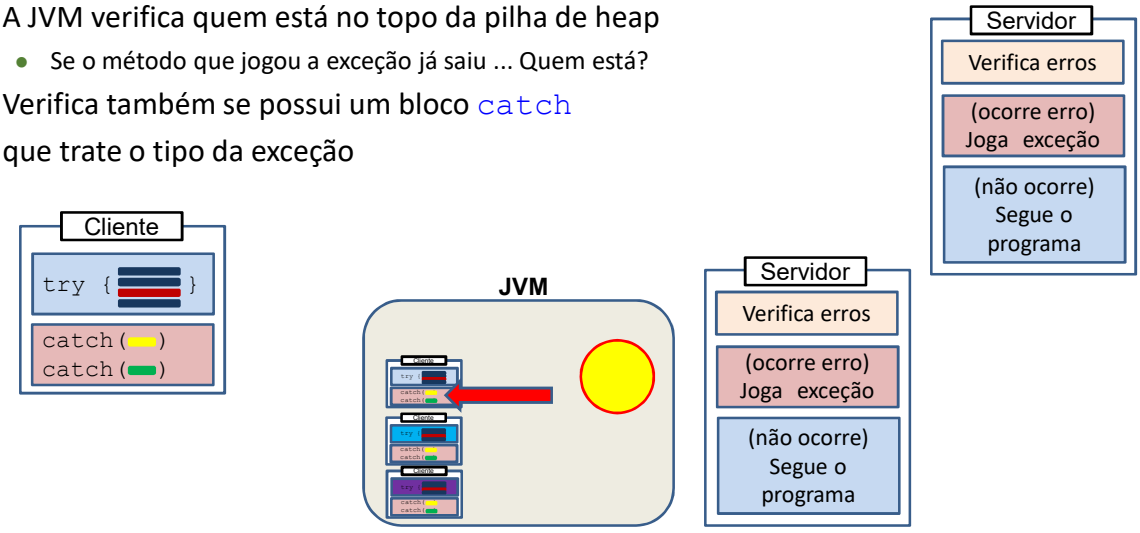
Cliente e servidor

- Acontece a identificação de um erro
- É criado um objeto exceção
- Joga a exceção para a JVM



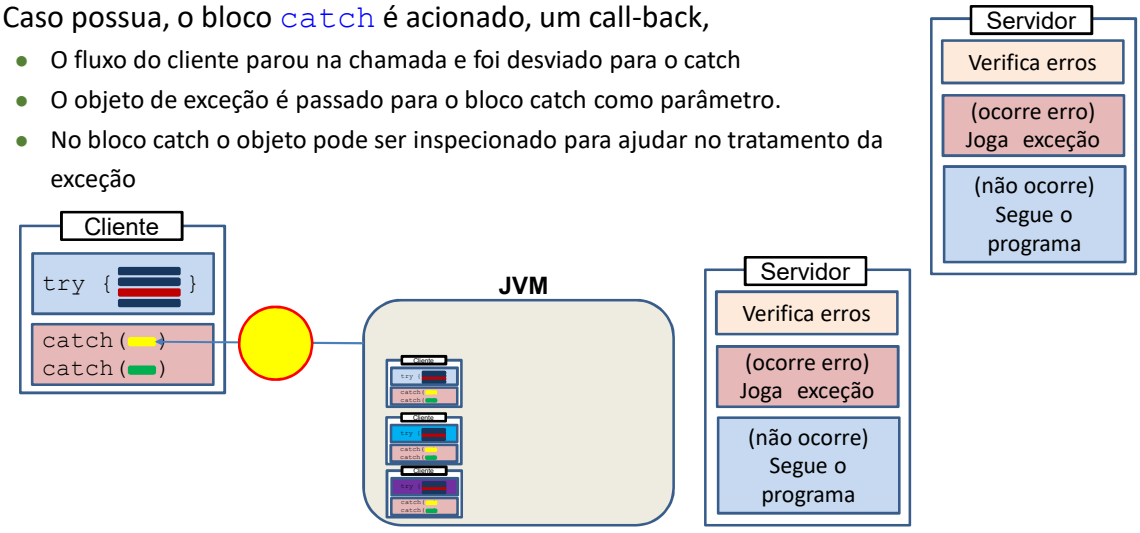
Cliente e servidor

- A JVM verifica quem está no topo da pilha de heap
  - Se o método que jogou a exceção já saiu ... Quem está?
- Verifica também se possui um bloco `catch` que trate o tipo da exceção



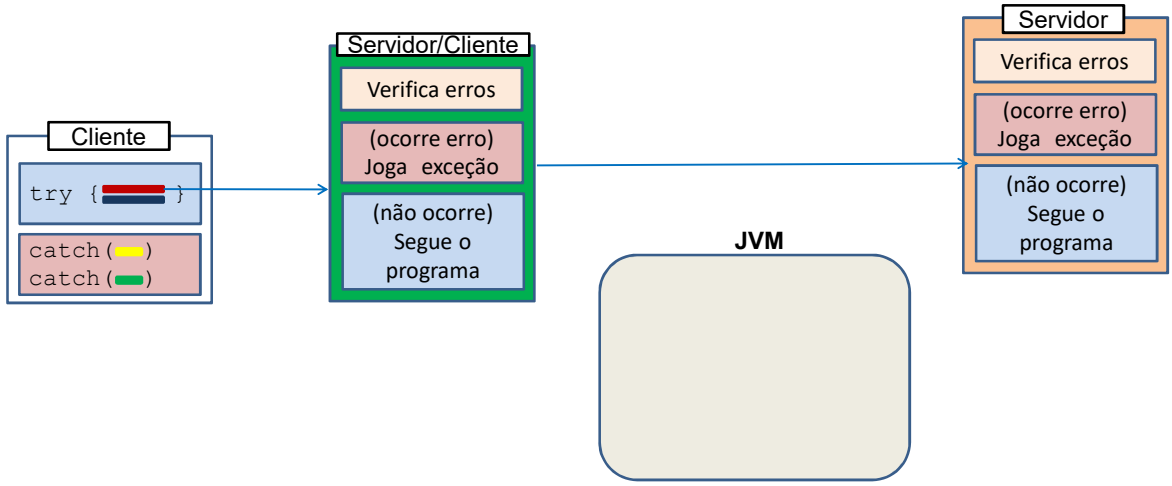
Cliente e servidor

- Caso possua, o bloco `catch` é acionado, um call-back,
  - O fluxo do cliente parou na chamada e foi desviado para o catch
  - O objeto de exceção é passado para o bloco catch como parâmetro.
  - No bloco catch o objeto pode ser inspecionado para ajudar no tratamento da exceção



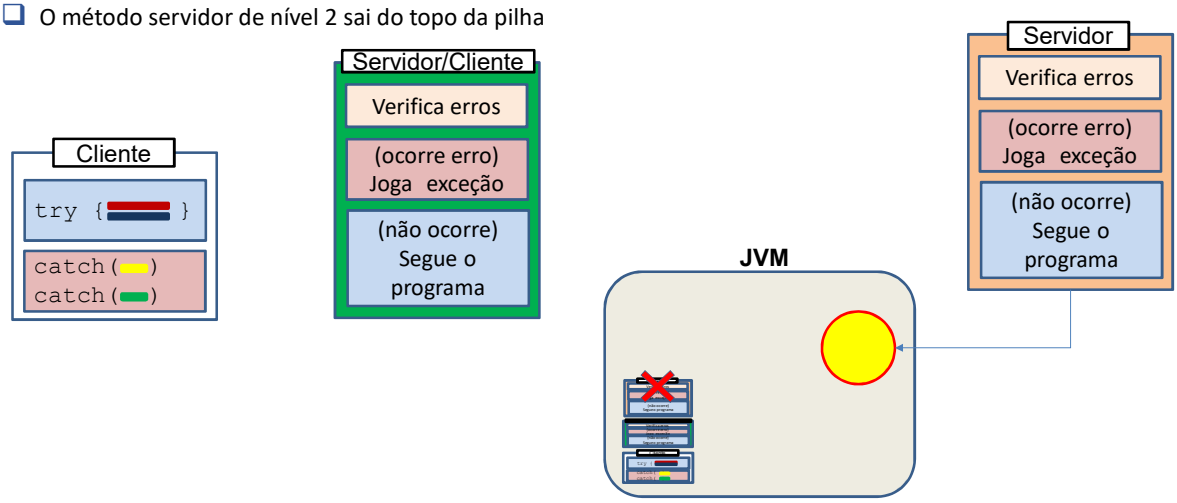
Cliente e servidor

- Situação 2. Cliente chama um método que chama outro método.



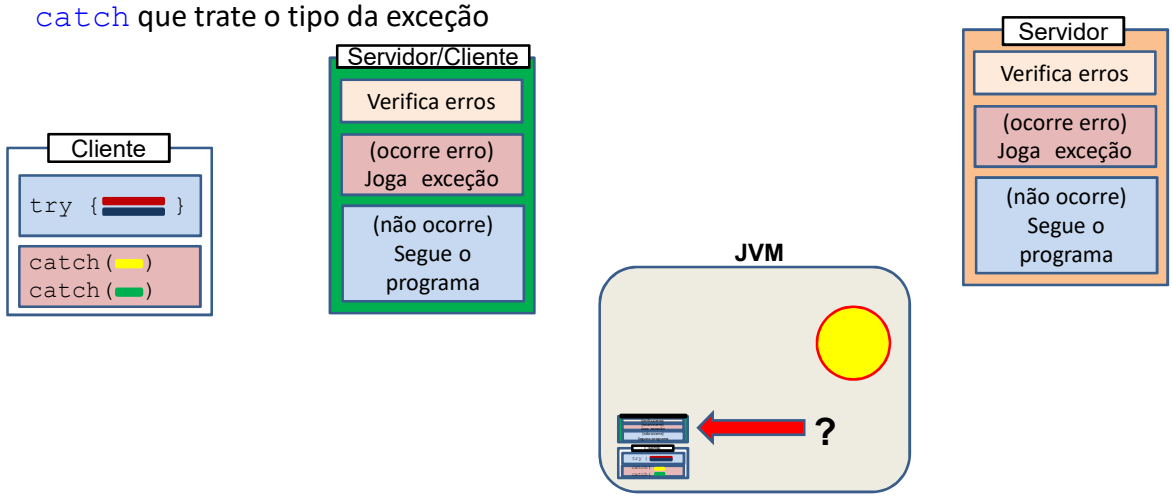
Cliente e servidor

- Ocorre uma exceção e a exceção é jogada para JVM
- O método servidor de nível 2 sai do topo da pilha



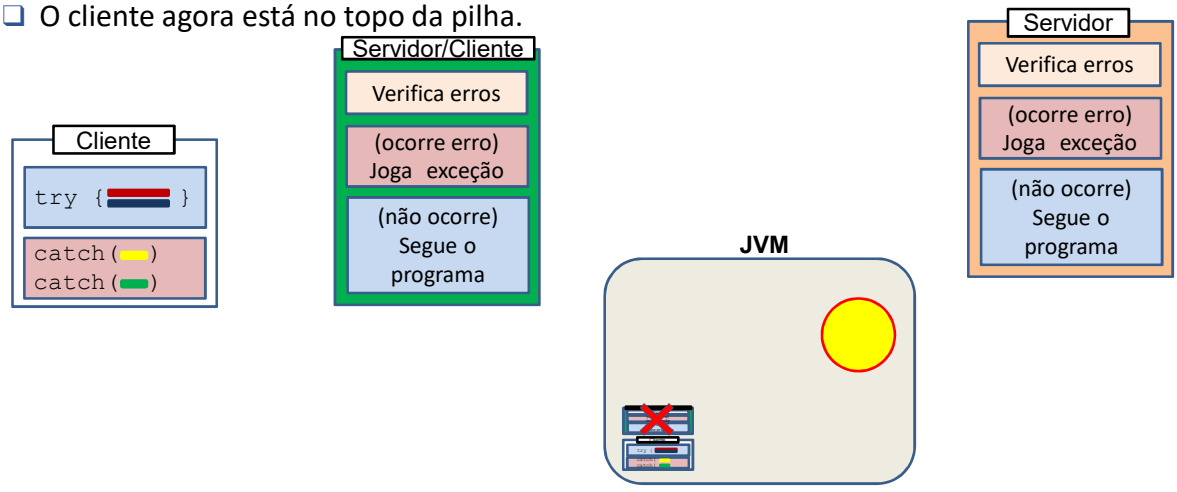
Cliente e servidor

- A JVM verifica se quem está no topo da pilha possui um bloco `catch` que trate o tipo da exceção



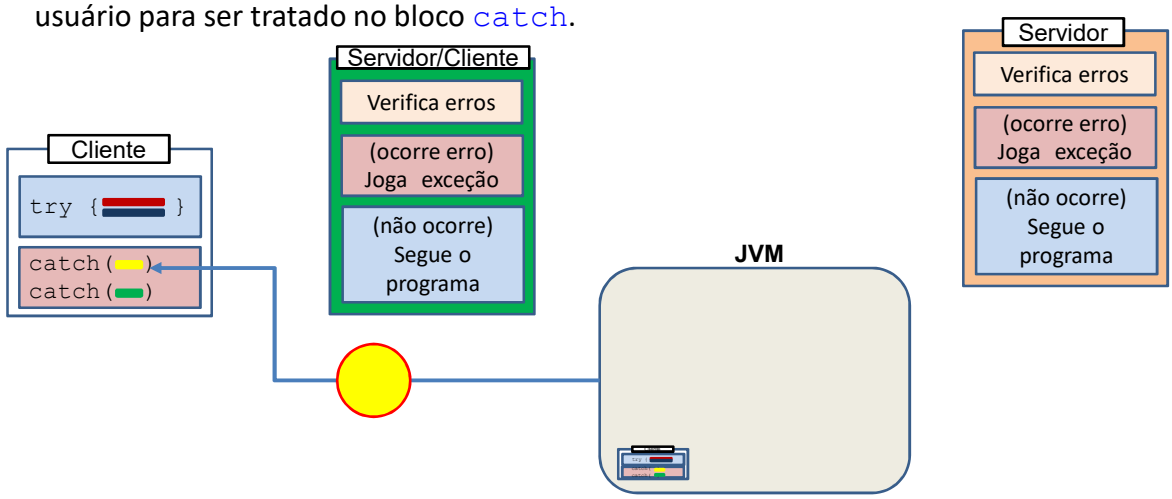
Cliente e servidor

- Como o método não possuía o bloco catch necessário o método é desempilhado.
- O cliente agora está no topo da pilha.

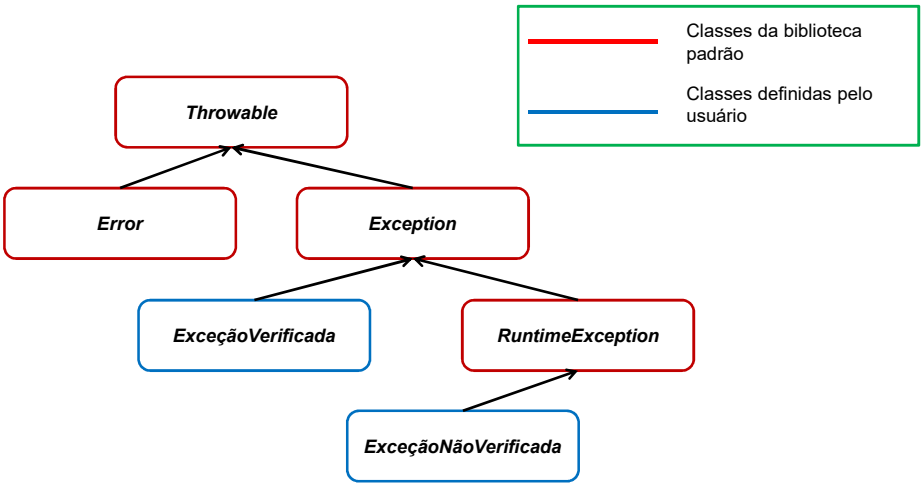


Cliente e servidor

- Caso o cliente consiga tratar, fluxo do programa volta ao usuário para ser tratado no bloco `catch`.



Hierarquia de classes de exceção



Exceções verificadas e não verificadas

- Exceções verificadas (*Checked Exceptions*):
  - Subclasse de `Exception`.
  - O compilador força o programador a lidar com elas, é um contrato (se eu jogo, você tem que capturar ... Ou rejogar*)
  - Para erros que podem acontecer, mas não conseguimos antecipar ou controlar ... "Não depende do programador"
  - Exemplos: `ClassNotFoundException`, `IOException`, `FileNotFoundException`.
- Exceções não verificadas (*Unchecked Exceptions*):
  - Subclasse de `RuntimeException`.
  - O compilador não verifica se a exceção é tratada, causam o término do programa se forem jogadas e não forem capturadas
  - O programador não deveria deixar o problema acontecer (discutimos isso no início), dá para antecipar e garantir que o erro não vai ocorrer
  - Exemplos: `NullPointerException`, `NumberFormatoException`, `IndexOutOfBoundsException`.

Servidor: Verificando argumentos e lançando uma exceção

```
/**
* @param key O nome ou número a ser pesquisado.
* @return Os detalhes correspondentes à chave,ou null  se não houver
nenhuma
*correspondência.
* @throws NullPointerException se a chave for null.
*/
public DetalhesContato getDetalhes(String chave){
      if(chave == null){
            throw new NullPointerException( "Chave nula em getDetalhes");
      }
      return (DetalhesContato) livro.get(chave);
}
```

Servidor: Verificando argumentos e lançando uma exceção

Este bloco serve para documentação Javadoc

```
/**
* @param key O nome ou número a ser pesquisado.
* @return Os detalhes correspondentes à chave,ou null se não houver nenhuma
*correspondência.
* @throws NullPointerException se a chave for null.
*/
public DetalhesContato getDetalhes(String chave){
	if(chave == null){
		throw new NullPointerException( "Chave nula em getDetalhes");
	}
	return (DetalhesContato) livro.get(chave);
}
```

Servidor: Verificando argumentos e lançando uma exceção

```
/**
* @param key O nome ou número a ser pesquisado.
* @return Os detalhes correspondentes à chave,ou null  se não houver nenhuma
*correspondência.
* @throws NullPointerException se a chave for null.
*/
public DetalhesContato getDetalhes(String chave){
	if(chave == null){
		throw new NullPointerException( "Chave nula em getDetalhes");
	}
	return (DetalhesContato) livro.get(chave);
}
```

Objeto de exceção é construído

Servidor: Verificando argumentos e lançando uma exceção

- Lançando uma exceção:

```
/**
* @param key O nome ou número a ser pesquisado.
* @return Os detalhes correspondentes à chave,ou null  se não houver nenhuma
*correspondência.
* @throws NullPointerException se a chave for null.
*/
public DetalhesContato getDetalhes(String chave){
      if(chave == null){
            throw new NullPointerException( "Chave nula em getDetalhes");
      }
      return (DetalhesContato) livro.get(chave);
}
```

O objeto exceção é lançado

Evitando a criação de um objeto com argumentos falhos

- Verifica os argumentos
- Ajusta os mesmos
- Se ainda assim não atende aos requisitos, ou seja se é detectado um erro, joga a exceção
- Por que esta classe? Isso é importante ...

```
public DetalhesContato(String nome, String telefone, String endereco){
  if(nome == null) { nome = ""; }
  if(telefone == null) { telefone = ""; }
  if(endereco == null) { endereco = ""; }

  this.nome = nome.trim();
  this.telefone = telefone.trim();
  this.endereco = endereco.trim();

  if(this.nome.length() == 0 && this.telefone.length() == 0) {
    throw new   IllegalStateException("Nome e telefone faltando.");
  }
}
```

Clausula `throws`

- Métodos que lançam uma exceção verificada, devem incluir uma clausula `throws` na assinatura.
- A clausula `throws` obriga aos usuários do método a tratar as exceções declaradas caso não sejam tratadas, haverá erro de compilação. É um contrato.
- Com isso o programador do método servidor deixa clara a informação de que excecções podem ser lançadas e devem ser, no mínimo, capturadas

```
public void savetoFile(String arquivoDestino)throws EOFException,
   FileNotFoundException, IOException {
      // código que pode jogar as exceções
}
```

- Interpretação: o método *saveToFile* pode jogar as exceções *EOFException*, *FileNotFoundException*, *IOException*
- As exceções são confirmadas, e devem ser tratadas pelo cliente chamador. Por isso o registro *throws*.
- A situação específica para que cada uma delas seja lançada devem estar documentado (não é para adivinhar!)

Cliente: Tentando chamar e capturando exceção

- Uso do bloco try-catch

```
try{
  livroEnderecos.saveToFile(filename);
  tentarNovamente = false;
}
catch(IOException e) {
  System.out.println("Erro ao salvar" + filename);
  System.out.println("Dados da exceção" + e);
  tentarNovamente = true;
}
```

Cliente: Tentando chamar e capturando exceção

- Uso do bloco try-catch
- Primeiro chama o método que pode jogar exceções em um bloco try

Sabemos que o método chamado lança IOException ...
Está na documentação

```
try{
  livroEnderecos.saveToFile(filename);
  tentarNovamente = false;
}
catch(IOException e) {
  System.out.println("Erro ao salvar" + filename);
  System.out.println("Dados da exceção" + e);
  tentarNovamente = true;
}
```

Prepara possível recuperação

Cliente: Tentando chamar e capturando exceção

- Depois prepara a captura da exceção
- Pode usar a exceção capturada. Observe que a exceção é capturada como um parâmetro de entrada
  - Então o bloco catch é um método ... Bom ... Quase ... Pode chamar de tratador ou handler ... Ou call-back para exceções.

```
try{
  livroEnderecos.saveToFile(filename);
  tentarNovamente = false;
}
catch(IOException e) {
  System.out.println("Erro ao salvar" + filename);
  System.out.println("Dados da exceção" + e);
  tentarNovamente = true;
}
```

Este bloco catch diz que está prepararado para receber

Mas podem ser lançadas várias exceções

- No método saveToFile a clausula throws informa que podem ser lançadas exeções com classes diferentes ...
- Mas só capturamos Exception ... Isso pode?
  - Pode, pois IOException é superclasse de todas as exceções de IO confimadas ... Não vai dar erro ... Ou seja qualquer uma lançada seria instanceOf IOException == true
  - Mas isso é preguiçoso e leva a código ruim (exemplos no final), pois não se sabe o erro específico (lembre-se do exemplo em C, com consulta ao errno)
- A hierarquia de classes e polimorfismo funciona nas exceções do Java (em ouras linguagens pode não ser assim). Então, vamos usar.

Capturando vários tipos de exceção

- Estrutura similar ao switch-case-default
- Capture as exceções mais específicas
- Depois as mais genéricas
- O *instanceOf* da exceção lançada já indica na cadeia de catch, o que aconteceu ...
- Para detalhar, use a exceção
- Convém receber cada classe de exceção numa variável diferente (depois teste isso)

```
try{
  livroEnderecos.saveToFile(filename);
} catch (EOFException e1){
   // Ação para final de arquivo alcançado
} catch (FileNotFoundException e2){
   // Ação para arquivo não encontrado
}
[...] // captura outras exceções específicas
catch(IOException e3) {
   // Ação para qualquer exceção de IO
   // não capturada
}catch (Exception e4){
   // Ação que Captura qualquer tipo de
   // exceção não tratada previamente
}
```

Recuperação de erro

- Clientes devem estar atentos aos erros.
  - Verificar o valor de retorno
  - **Não ignorar exceções**
  - Incluir código para tentativa de recuperação
    - Geralmente um loop será exigido

Recuperação de erro

- Comentários no exemplo.
- A recuperação está em azul ... Poderia ser melhor ...

```
boolean salvou = false;
int tentativas = 0;
while (!salvou && tentativas < MAX_TENTATIVAS) {
  try {
    enderecos.saveToFile(nomeDoArquivo);
    salvou = true; // só vai executar isso se não der erro ...
  } catch (IOException e){
     System.out.println("Não foi possivel salvar em "+ nomeDoArquivo);
     tentativas++; // ajusta o número de tentativas ...
     if (tentativas < MAX_TENTATIVAS){ // se não atingiu o máximo ...
       nomeDoArquivo = um nome de arquivo diferente;
     }
  }
}
if (!salvou){ // depois das tentativas de recuperação, ainda não deu ...
   //informa o problema e pode optar por desistir;
}
```

Finally

- A cláusula `finally` permite que uma ação seja realizada mesmo que uma exceção seja lançada.

  Ex: Um arquivo foi aberto e, no meio de sua manipulação, uma exceção foi lançada. O programa seria abortado mas o arquivo continuaria aberto. Usa-se `finally` nessas situações.

```
try{
        //Protege uma ou mais instruções aqui.
}
catch(Exception e){
        //Informação da exceção e modo de recuperação aqui.
}
finally{
        //Realize qualquer ações aqui necessárias, seja a exceção lançada ou não.
}
```

Exemplo com finally

```
public class FatorialTryCatchFinally
{
  public static void main(String[] args){
    int n=0;

    try {
      n = Integer.parseInt(args[0]);
    }
    catch(Exception e)  {
      System.out.println("Ocorreu um erro\nExcecao: "+e);
    }
    finally {
      int resultado;
      for(resultado=1; n>0; n--) resultado = resultado*n;

       System.out.println("\n"+n+"! = "+resultado);
    }
  }
}
```

Exemplo com finally

O programa FatorialTryCatchFinally, calcula o fatorial do número passado como argumento, se ocorrer algum erro, ele calcula o fatorial de zero.

- Os Blocos `try` e `catch` funcionam da mesma maneira.
- O bloco associado a instrução `finally` é executado independente de haver ou não uma exceção.

Na tela do console:

```
>java FatorialTryCatchFinally
Ocorreu um erro
Excecao: java.lang.ArrayIndexOutOfBoundsException: 0

0! = 1
Terminou!
```

Criando novas exceções

- Estenda de `Exception` ou `Runtime-Exception`.
- Defina novos tipos para fornecer melhores informações diagnosticadas.

Criando novas exceções

- Ex:

```
public class UsuarioNaoCadastradoException extends Exception{
   private String nome;
   public UsuarioNaoCadastradoException(String nome){
          this.nome = nome;
   }
   public String getNome(){
          return nome;
   }
   public String toString(){
          return ("Usuario: "+this.nome+", nao esta cadastrado no sistema.");
   }
}
```

Implementando nova exceção

- Ex:

```
public class ladoInvalidoException extends Exception{
   private float lado;
   public ladoInvalidoException(float lado){
          this.lado = lado;
   }
   public float getLado(){
          return lado;
   }
   public String toString(){
          return (lado+" Nao e um numero valido para o lado");
   }
}
```

Implementando nova exceção

```
import java.io.*;
public class areaQuadrado{
public static void main(String[] args){
    float lado;
    BufferedReader s = new BufferedReader(new InputStreamReader(System.in));
    while (true){
            try{
                    lado = Float.parseFloat(s.readLine());  //Lê o lado do quadrado
                    if (lado <= 0){
                            throw new ladoInvalidoException(lado);  //Instancia e joga a exceção
                    }
                    System.out.println("Area do quadrado: "+lado*lado);
                    break;
            }catch(ladoInvalidoException e1){       //captura a exceção ladoInvalidoException
                    System.out.println(e1);
            }catch(NumberFormatException e2){        //captura a exceção NumberFormatException
                    System.out.println("Lado deve ser um numero!");
            }catch(IOException e3){        //captura a exceção IOExcepion
                    System.out.println("Nao foi possivel executar essa operação");
            }
    }
}}
```

Criando Exceções

Para criar exceções é necessário criar uma classe que Herde de uma classe de exceção existente.

Exemplo: Classe que trata da exceção de divisão por zero
Uma divisão por zero lança a exceção ArithmeticException, por isso a herança será feita dessa classe.

```
public class ThrowsExemplo1 extends ArithmeticException
{
    throwsExemplo1()
    {
        super("Zero como denominador.");
    }
}
```

Lançando exceção no main()

```
public class ThrowsExemplo2 {
  public static void main(String[] args)throws throwsExemplo1
  {
    double a = Double.parseDouble(args[0]);
    double b = Double.parseDouble(args[1]);

    if(b==0) throw new ThrowsExemplo1();
    else {
      double resultado = a/b;
      System.out.println(a+"/"+b+" = "+resultado);
    }
  }
}
```

Recebe 2 argumentos, e faz a divisão do primeiro pelo segundo.
No console:

```
>java throwsExemplo2 5 0
Exception in thread "main" throwsExemplo1: Zero como denominador.
        at throwsExemplo2.main(throwsExemplo2.java:23)
```

Prevenção de erro

- Clientes podem frequentemente utilizar os métodos de pesquisa do servidor para evitar erros.
  - Ter clientes mais robustos significa que os servidores podem ser mais confiáveis.
  - Exceções não-verificadas podem ser utilizadas.
  - Simplifica a lógica do cliente.
- Pode aumentar o acoplamento cliente/servidor.

Entrada e saída de texto

- Entrada e saída são particularmente propensas a erros.
  - Envolvem interação com o ambiente externo.
- O pacote `java.io` suporta entrada e saída.
- `java.io.IOException` é uma exceção verificada.

Leitores, escritores e fluxos

- Leitores e escritores lidam com entrada textual.
  - Com base no tipo `char`.
- Fluxos lidam com dados binários.
  - Com base no tipo `byte`.
- O projeto "*address-book-io*" ilustra a E/S textual.

Saída de texto

- Utiliza a classe `FileWriter`.
  - Abre um arquivo.
  - Grava no arquivo.
  - Fecha o arquivo.
- Falha em um ponto qualquer resulta em uma `IOException`.

Saída de texto

```
try {
    FileWriter writer = new FileWriter(nome do arquivo);
    while(há mais texto para escrever) {
        ...
        writer.write(próxima parte do texto);
        ...
    }
    writer.close();
}
catch(IOException e) {
    algo saiu errado ao acessar o arquivo
}
```

Entrada de texto

- Utiliza a classe `FileReader` como infra-estrutura de `BufferedReader` para entrada baseada em linha.
  - Abre um arquivo.
  - Lê do arquivo.
  - Fecha o arquivo.
- Falha em um ponto qualquer resulta em uma `IOException`.

Entrada de texto

```
try {
    BufferedReader reader = new BufferedReader(
                            new FileReader("nome do arquivo "));
    String line = reader.readLine();
    while(line != null) {
    faça algo com a linha
        line = reader.readLine();
    }
    reader.close();
}
catch(FileNotFoundException e) {
    o arquivo específico não pode ser localizado
}
catch(IOException e) {
    algo saiu errado com a leitura ou fechamento
}
```

Exemplo extra 1

- No programa Area , foi colocado um if para solucionar o problema da quantidade de argumentos. Mas se o argumento passado for uma letra ao invés de um número, acontecerá o lançamento de uma exceção: NumberFormatException
- Uma solução seria criar funções para testar os argumentos que estão sendo passados. Mas a estrutura try catch resolve o problema de maneira mais vantajosa.

```
public class Area
{
  public static void main(String[] args)
  {
    if(args.length==2)
    {
      double a=Double.parseDouble(args[0]);
      double b=Double.parseDouble(args[1]);
      double area = a * b;
      System.out.println(”Area = ” +area);
    }
  }
}
```

Exemplo extra 1 usando try-catch

- Por que caputamos Exception? Preguiça?
- Por que a mensagem de print como esta?

```
public class AreaTryCatch
{
  public static void main(String[] args)
  {
    try
    {
       double a = Double.parseDouble(args[0]);
       double b = Double.parseDouble(args[1]);
       double area = a*b;
       System.out.println("Area = "+area);
    }
    catch(Exception erro)
    {
        System.out.println("Nao foi fornecido um numero de
              argumentos suficientes, " + "\n ou um dos
              valores é invalido.\n Excecao: “ + erro);
    }
  }
}
```

Exemplo extra 1 usando try-catch

- Na chamada baixo, apenas um argumento é passado:

```
>java AreaTryCatch 2
Nao foi fornecido um numero de argumentos
suficientes,
ou um dos valores é invalido.
Excecao: java.lang.ArrayIndexOutOfBoundsException: 1
```

- Nesta outra chamada, é passado um caracter que não é número:

```
>java AreaTryCatch k 2.5
Nao foi fornecido um numero de argumentos
suficientes,
ou um dos valores é invalido.
Excecao: java.lang.NumberFormatException: For input
string: "k"
```

Analisando as partes

`try{`
Inicia o bloco no qual o conteúdo é tratado pela cláusula catch.

`catch(Exception e){`
Aqui utilizamos uma exceção genérica `Exception`, qualquer erro surgido no trecho de código delimitado pelo bloco try é tratado pela rotina delimitada pela cláusula catch.

Ao invés de Exception podemos colocar vários catch com exceções mais específicas, assim cada tipo de exceção pode receber um tratamento específico.

Exemplo extra 1 usando try-catch

- Agora melhoramos o tratamento de exceção, capturando exceções Específicas?
- E como adivinhar quais são? Não é para adivinhar!!!

```
public class AreaTryCatch2
{
  public static void main(String[] args)
   {
      try
       {
           double a = Double.parseDouble(args[0]);
           double b = Double.parseDouble(args[1]);
           double area = a*b;
           System.out.println("Area = "+area);
      }
      catch(NumberFormatException e) {
           System.out.println("Pelo menos um dos valores é
                                invalido.\n Excecao: "+e);
      }
      catch(ArrayIndexOutOfBoundsException e) {
           System.out.println("Numero insufuciente de
                                argumentos.\n Excecao: "+e);
      }
   }
}
```

Analisando

- Na chamada abaixo, apenas um argumento é passado:

```
>java AreaTryCatch2 2.5
Numero insufuciente de argumentos.
Excecao: java.lang.ArrayIndexOutOfBoundsException:
1
```

- Nesta outra chamada, é passado um caracter que não é número:

```
>java AreaTryCatch2 2.5 j
Pelo menos um dos valores é invalido.
Excecao: java.lang.NumberFormatException: For input
string: "j"
```

Propagando exceções: cláusula `throws`

A cláusula throws, captura a exceção e relança ela para uma outra classe tratá-la. O método que usa o throws, "sabe" que pode ocorrer uma exceção mas não se preocupa com o tratamento.

Declaração:

```
void metodoQueNaoTrataExcecao() throws Exception
{
   ...
}
```

Como exemplo nos programas que se faz uso do pacote java.io.*; é necessário relançar a exceção através do `throws` ou tratá-la com `try - catch`.