Mecanismo para tratamento de exceções

Algumas causas das situações de erros

- Implementação incorreta.
  - Não atende à especificação.
- Solicitação de objeto inapropriado.
  - Por exemplo, índice inválido.
- Estado do objeto inconsistente ou inadequado.
  - Por exemplo, surgindo devido à extensão de classe.

Nem sempre erro do programador

- Erros surgem freqüentemente do ambiente:
  - URL incorreto inserido; e
  - interrupção da rede.
- Processamento de arquivos é particularmente propenso a erros:
  - arquivos ausentes; e
  - falta de permissões apropriadas.

Questões a serem resolvidas

- Qual é o número de verificações por um servidor nas chamadas de método?
- Como informar erros?
- Como um cliente pode antecipar uma falha?
- Como um cliente deve lidar com uma falha?

Um exemplo

- Crie um objeto `AddressBook`.
- Tente remover uma entrada.
- Resulta em um erro em tempo de execução.
  - De quem é a 'falha'?
- Antecipação e prevenção são preferíveis a apontar um culpado.

Valores dos argumentos

- Argumentos representam uma séria 'vulnerabilidade' para um objeto servidor.
  - Argumentos do construtor inicializam o estado.
  - Argumentos do método contribuem freqüentemente com o comportamento.
- Verificação de argumento é uma das medidas defensivas.

Verificando a chave

```
public void removeDetails(String key)
{
    if(keyInUse(key)) {
        ContactDetails details =
                        (ContactDetails) book.get(key);
        book.remove(details.getName());
        book.remove(details.getPhone());
        numberOfEntries--;
    }
}
```

Informe de erro do servidor

- Como informar argumentos inválidos?
  - No usuário?
    - Há usuários humanos?
    - Eles podem resolver o problema?
  - No objeto cliente?
    - Retorna um valor de diagnóstico.
    - *Lança uma exceção.*

Retornando um diagnóstico

```
boolean public removeDetails(key String)
{
    if(keyInUse(key)) {
        ContactDetails details =
                        (ContactDetails) book.get(key);
        book.remove(details.getName());
        book.remove(details.getPhone());
        numberOfEntries--;
        return true;
    }
    else {
        return false;
    }
}
```

Respostas do cliente

- Testar o valor de retorno.
  - Tente recuperar no erro.
  - Evite a falha do programa.
- Ignorar o valor de retorno.
  - Não pode ser evitado.
  - Possibilidade de levar a uma falha do programa.
- Exceções são preferíveis.

Exceptions

- An exception is an unusual situation that occurs when the program is running.
- Exception Management
  - **Define the error condition**
  - **Enclose code containing possible error (try).**
  - **Alert the system if error occurs (throw).**
  - **Handle error if it is thrown (catch).**

Princípios do lançamento de exceções

- Um recurso especial de linguagem.
- Nenhum valor de retorno ‘especial’ necessário.
- Erros não podem ser ignorados no cliente.
  - O fluxo normal de controle é interrompido.
- Ações específicas de recuperação são encorajadas.

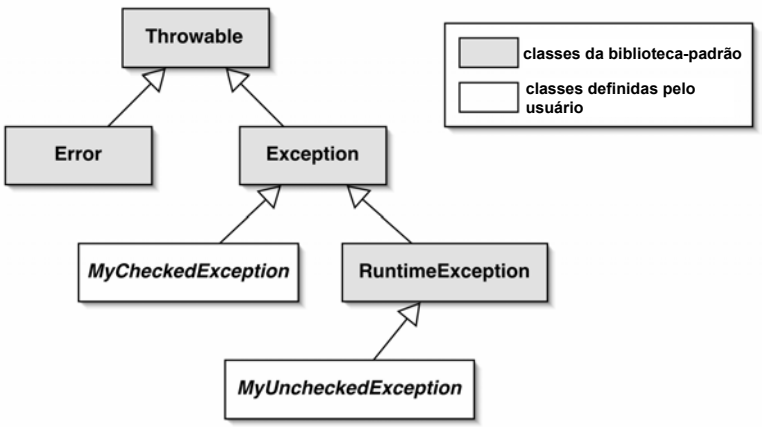
Lançando uma exceção (1)

```
/**
 * Pesquisa um nome ou um número de telefone e retorna
 * os detalhes do contato correspondentes.
 * @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 void getDetails(String key)
{
    if(oldkey == null){
        throw new NullPointerException(
                          "null key in getDetails");
    }
    return (ContactDetails) book.get(key);
}
```

Lançando uma exceção (2)

- Um objeto de exceção é construído:
  - `new ExceptionType("...");`
- O objeto exceção é lançado:
  - `throw ...`
- Documentação Javadoc :
  - `@throws ExceptionType reason`

A hierarquia de classes de exceção



Categorias de exceção

- Exceções verificadas:
  - subclasse de `Exception`;
  - utilizadas para falhas iniciais; e
  - onde a recuperação talvez seja possível.
- Exceções não-verificadas:
  - subclasse de `RuntimeException`;
  - utilizadas para falhas não-antecipadas; e
  - onde a recuperação não é possível.

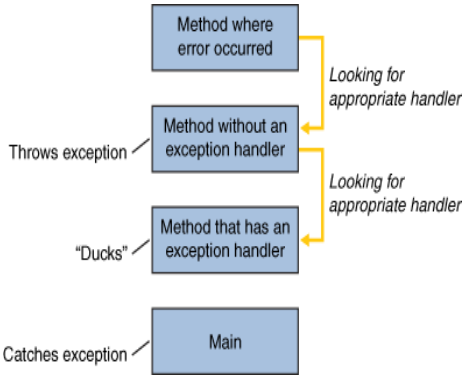
O efeito de uma exceção (i)

- O método
  - de termina prematuramente
  - Cria um objeto que herda de Exception
  - e “joga” este objeto para o runtime (a JVM)
- Nenhum valor de retorno é retornado.
- Controle não retorna ao ponto da chamada do cliente.
  - Portanto, o cliente não pode prosseguir de qualquer maneira.
- Um cliente pode ‘capturar’ uma exceção.

Funcionamento

O efeito de uma exceção (ii)

- O sistema de runtime começa a procurar um tratador para a exceção
- Procura a partir da pilha de chamadas (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 apresentados



Exceções não-verificadas

- A utilização dessas exceções ocorre de forma ‘não-verificada’ pelo compilador.
- Causam o término do programa se não capturadas.
  - Essa é a prática normal.
- `IllegalArgumentException` é um exemplo típico.
- `ArrayIndexOutOfBoundsException` também

Verificação de argumento

```
public void ContactDetails getDetails(String key)
{
    if(key == null) {
        throw new NullPointerException(
                        "null key in getDetails");
    }
    if(key.trim().length() == 0) {
        throw new IllegalArgumentException(
                        "Empty key passed to getDetails");
    }
    return (ContactDetails) book.get(key);
}
```

Evitando a criação de objeto

```
public ContactDetails(String name, String phone, String address)
{
    if(name == null) {
        name = "";
    }
    if(phone == null) {
        phone = "";
    }
    if(address == null) {
        address = "";
    }

    this.name = name.trim();
    this.phone = phone.trim();
    this.address = address.trim();

    if(this.name.length() == 0 && this.phone.length() == 0) {
        throw new IllegalStateException(
                "Either the name or phone must not be blank.");
    }
}
```

Tratamento de exceções

- Exceções verificadas devem ser capturadas.
- O compilador assegura que a utilização dessas exceções seja fortemente controlada.
  - Tanto no servidor como no cliente.
- Se utilizadas apropriadamente, é possível recuperar-se das falhas.

A cláusula *throws*

- Métodos que lançam uma exceção verificada devem incluir uma cláusula throws :

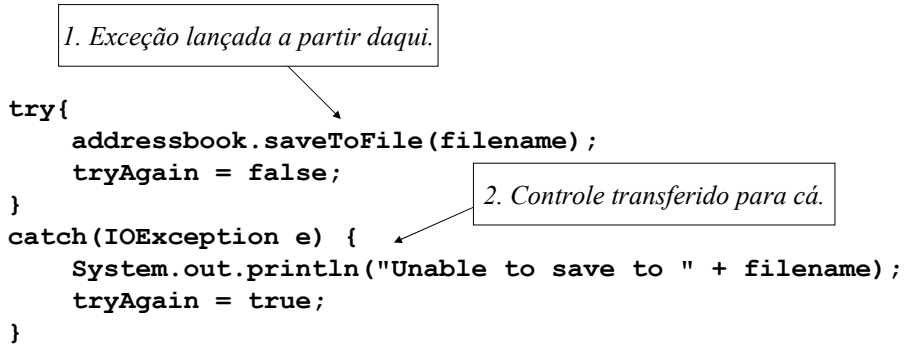
```
public void saveToFile(String destinationFile)
    throws IOException
```

O bloco *try*

- Clientes que capturam uma exceção devem proteger a chamada com um bloco try:

```
try {
    Proteja uma ou mais instruções aqui.
}
catch(Exception e) {
    Informe da exceção e recuperação aqui.
}
```

O bloco *try*




```
try{
    addressbook.saveToFile(filename);
    tryAgain = false;
}
catch(IOException e) {
    System.out.println("Unable to save to " + filename);
    tryAgain = true;
}
```

Capturando vários tipos de exceção

```
try {
    ...
    ref.process();
    ...
}
catch(EOFException e) {
    // Toma a ação apropriada para uma exceção
    // de final de arquivo alcançado.
    ...
}
catch(FileNotFoundException e) {
    // Toma a ação apropriada para uma exceção
    // de final de arquivo alcançado.
    ...
}
```

A cláusula *finally*

```
try {
    Proteja uma ou mais instruções aqui.
}
catch(Exception e) {
    Informe e recupere a partir da exceção aqui.
}
finally {
    Realize quaisquer ações aqui comuns quer ou não
    uma exceção seja lançada.
}
```

A cláusula *finally*

- Uma clásula finally é executada mesmo se uma instrução de retorno for executada nas cláusulas try ou catch.
- Ainda há uma exceção não-capturada ou propagada via a cláusula finally.

Definindo novas classes de exceção

- Estenda `Exception` ou `Runtime-Exception`.
- Defina novos tipos para fornecer melhores informações diagnósticas.
  - Inclua informações sobre a notificação e/ou recuperação.

```
public class NoMatchingDetailsException extends Exception
{
    private String key;

    public NoMatchingDetailsException(String key)
    {
        this.key = key;
    }

    public String getKey()
    {
        return key;
    }

    public String toString()
    {
        return "No details matching '" + key +
               "' were found.";
    }
}
```

Recuperação após erro

- Clientes devem tomar nota dos informes de erros.
  - Verifique o valor de retorno.
  - Não 'ignore' exceções.
- Inclua o código para a tentativa de recuperação.
  - Freqüentemente isso exigirá um loop.

Tentativa de recuperação

```
// Tenta salvar o catálogo de endereços.
boolean successful = false;
int attempts = 0;
do {
    try {
        addressbook.saveToFile(filename);
        successful = true;
    }
    catch(IOException e) {
        System.out.println("Unable to save to " + filename);
        attempts++;
        if(attempts < MAX_ATTEMPTS) {
            filename = um nome de arquivo alternativo;
        }
    }
} while(!successful && attempts < MAX_ATTEMPTS);
if(!successful) {
    Informa o problema e desiste;
}
```

Prevenção de erro

- Clientes podem freqüentemente 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
}
```

Mecanismo para tratamento de exceções e erros (de novo!)

- Java oferece um mecanismo para a programação explícita de tratamento de erros e exceções, de forma elegante e encapsulada, através da estrutura

  try - catch - finally

- O objetivo é separar as rotinas de tratamento de erro do código do programa, tornando o tratamento de exceções parte visível da interface do objeto.
- A ocorrência desses erros serão sinalizadas pelas ***Exceções*** , que são objetos que contém informações sobre o erro detectado.
- Existem diversas categorias de exceções e cada pacote utilizado pelo programa traz exceções específicas.
- Além disso podemos desenvolver nossas próprias exceções.

Exemplo 1

```
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);
    }
  }
}
```

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.

Programa Area usando try catch

```
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);
        }

    }
}
```

Programa Area 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.

Programa Area usando try catch

```
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.\nExcecao: "+e);
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
             System.out.println("Numero insufuciente de
                                  argumentos.\nExcecao: "+e);
        }
    }
}
```

Como as exceções são tratadas pelo programa

- 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"
```

Estrutura try catch 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);
          }
    }
}
```

Estrutura try catch 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!
```

Propagando exceções: cláusula `throws`

A cláusula throws, captura a exceção e lança 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 lançar a exceção através do `throws` ou tratá-la com `try - catch`.

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.");
    }
}
```

```
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);
        }
    }
}
```

Programa que recebe 2 argumentos, e faz a dvisão do primeiro pelo segundo.
No console:

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

Detectando o <ENTER>

A comparação com "" funciona usando o método:

compareTo da String, exemplo:

```
if(nome.CompareTo("") == 0) ou
nome.equals("")    ou ainda
nome.equalsIgnoreCase("");
```

... Na verdade temos uma String com conteúdo ""

A comparação que não funcina é `(nome=="")`