

TRATAMENTO DE EXCEÇÕES E I/O DE ARQUIVOS

Mário Meireles Teixeira
mario@deinf.ufma.br

Principais conceitos

- Programação defensiva
 - Antecipar o que pode sair errado
- Lançamento e tratamento de exceções
- Informe de erro
- Processamento de arquivos simples

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, devido à extensão de uma classe

Nem sempre erro do programador

- Erros surgem frequentemente do ambiente:
 - URL incorreto inserido
 - interrupção da rede
- Processamento de arquivos é particularmente propenso a erros:
 - arquivos ausentes
 - falta de permissões apropriadas
 - disco cheio

Programação defensiva

- Interação cliente-servidor:
 - Um servidor deve assumir que os clientes são bem-comportados?
 - Ou ele deve assumir que os clientes são potencialmente hostis?
- Diferenças significativas na implementação são requeridas

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 de erros são preferíveis a apontar um culpado

Valores dos argumentos

- Argumentos representam uma séria 'vulnerabilidade' para um objeto servidor
 - Argumentos do construtor inicializam o estado do objeto
 - Argumentos do método alteram freqüentemente o comportamento do objeto
- Verificação de argumento é uma das medidas defensivas que podem ser tomadas pelo programa

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 código de erro/status (típico de programação estruturada)
 - *Lança uma exceção – Java/OO*

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 do erro
 - Evite a falha do programa
- Ignorar o valor de retorno
 - Não há como evitar
 - Possibilidade de levar a uma falha do programa
- Exceções são preferíveis

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

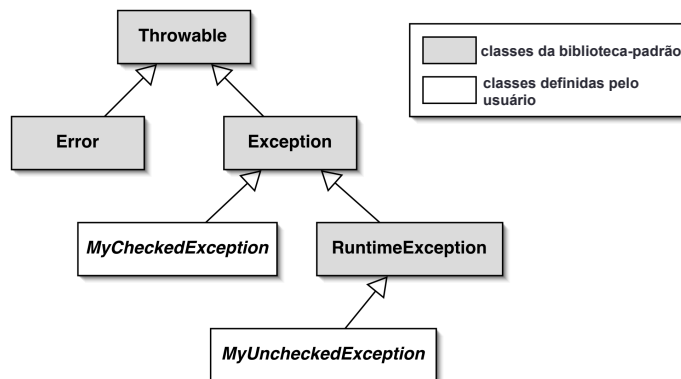
Lançando uma exceção (1)

```
/**
 * 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(key == null){
        throw new NullPointerException(
            "chave nula em 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
 - onde a recuperação talvez seja possível
- Exceções não-verificadas:
 - subclasse de `RuntimeException`
 - utilizadas para falhas não-antecipadas
 - onde a recuperação não é possível

O efeito de uma exceção

- O método de lançamento da exceção termina prematuramente
- Nenhum valor de retorno é retornado
- Controle não retorna ao ponto da chamada do cliente
 - Portanto, o cliente não pode prosseguir de qualquer maneira, pois o programa é interrompido
- Um cliente pode 'capturar' uma exceção, para tentar recuperar-se do erro ou enviar uma mensagem mais amigável ao usuário/cliente

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

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);
}
```

- É importante verificar se os argumentos estão corretos antes de prosseguir com o propósito principal do método

Exceções em construtores

```
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.");
    }
}
```

- Exceções também podem ser lançadas a partir de construtores
- Neste caso, deseja-se evitar criar uma entrada que não se possa indexar

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 ao programa 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
}
```

- Qualquer número de instruções pode ser incluído em um bloco try
- A cláusula catch tentará capturar exceções geradas por qualquer instrução dentro do try precedente

O bloco try

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

1. Exceção lançada a partir daqui

2. Controle transferido para cá

- As instruções no bloco try são denominadas *protegidas*. Se nenhuma exceção ocorrer, elas serão executadas normalmente e o catch será ignorado
- Uma exceção impede que o fluxo normal de controle continue, retomando-se a execução a partir da cláusula catch
- Depois que o catch for completado, a execução *não retornará* para o try

Múltiplas exceções

```
try {
    ...
    ref.processar();
    ...
}
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 arquivo não encontrado
    ...
}
```

- As cláusulas catch são verificadas na ordem em que são escritas
- Uma cláusula catch para um tipo particular de exceção não deve seguir a outro catch que se refira a um dos seus supertipos – o supertipo capturará a exceção antes do catch do subtipo (polimorfismo)
- Porém, pode-se aproveitar esta característica e fazer um catch único para todas as exceções (tipos e subtipos)
- Uma exceção pode ser *propagada* do método chamado para o chamador

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 quer ou não  
    uma exceção seja lançada  
}
```

- A cláusula finally é opcional e é sempre executada, mesmo que não ocorra uma exceção
- Se uma exceção for lançada no try, mas não capturada, o finally ainda assim é executado – p. ex., uma exceção propagada a partir de um método
- Na prática, finally pode ser usado para fechar arquivos, conexões de rede, de banco de dados...

Definindo novas classes de exceção

- Estenda `Exception` ou `RuntimeException`
- 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.";
    }
}

```

- A nova exceção **NoMatchingDetailsException**, específica da aplicação, permite que o valor de **key** seja disponibilizado para as aplicações que queiram se recuperar do erro.

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.
 - Frequentemente isso exigirá um loop
- Um exemplo equivocado de tratamento de exceções:

```

AddressDetails details = null;
try {
    details = addresses.getDetails(...);
}
catch (Exception e) {
    System.out.println("Error: " + e);
}
...

```

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:
 - Arquivo corrompido
 - Arquivo não existe
 - Disco cheio / quota excedida
 - Permissões insuficientes...
- 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 exemplo a seguir (*address-book-io*) ilustra a E/S textual
- O projeto também inclui métodos para ler/gravar versões binárias dos objetos `AddressBook` e `ContactDetails` -- serialização
- Mais informações sobre I/O: <http://java.sun.com/docs/books/tutorial/essential/index.html>

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`
- Arquivo:
 - `AddressBookFileHandler.java`, do projeto *address-book-io*

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`
- 'Empacota' com `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
}
```

Revisão (1)

- Erros em tempo de execução surgem por várias razões:
 - Uma chamada cliente inadequada a um objeto servidor
 - Um servidor incapaz de atender a uma solicitação
 - Erro de programação no cliente e/ou servidor

Revisão (2)

- Erros em tempo de execução freqüentemente levam a falhas de programa
- Programação defensiva antecipa erros — tanto no cliente como no servidor
- Exceções fornecem um mecanismo de informe e recuperação de erros