



Curso Java Básico

Uma introdução prática usando
BlueJ



Tratando erros

Principais conceitos a serem abordados

- Programação defensiva
 - Antecipar o que pode sair errado
- Lançamento e tratamento de exceção
- Informe de erro
- Processamento de arquivo 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, surgindo devido à extensão de classe.

Nem sempre erro do programador

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

Catálogo de endereços

- Utilizaremos uma aplicação que armazena dados de contatos pessoais:
 - Os dados de cada contato são: nome, endereço e número de telefone;
 - Os detalhes de contato podem ser pesquisados tanto pelo nome quanto pelo número do telefone;

Exercício

- **Catálogo de endereços**

- Inicie o **BlueJ**, abra o projeto *address-book-v1g*, crie uma instância de *AddressBookDemo* e chame o método *showInterface*.
- Interaja com o catálogo de endereços: liste suas entradas, pesquise com o argumento `'08459'` e adicione um novo contato.
- Repita os mesmos passos com o projeto *address-book-v1t*.

Código-fonte: AddressBook

```
public class AddressBook
{
    private TreeMap<String, ContactDetails> book;
    private int numberOfEntries;

    public AddressBook()
    {
        book = new TreeMap<String, ContactDetails>();
        numberOfEntries = 0;
    }
}
```


Código-fonte: AddressBook

```
public ContactDetails getDetails(String key)
{
    return book.get(key) ;
}
```

```
public boolean keyInUse(String key)
{
    return book.containsKey(key) ;
}
```

```
public int getNumberOfEntries()
{
    return numberOfEntries;
}
```

Código-fonte: AddressBook

```
public void addDetails(ContactDetails details)
{
    book.put(details.getName(), details);
    book.put(details.getPhone(), details);
    numberOfEntries++;
}

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

Código-fonte: AddressBook

```
public void changeDetails(String oldKey,  
                           ContactDetails details)  
{  
    removeDetails(oldKey);  
    addDetails(details);  
}  
public String listDetails()  
{  
    StringBuilder allEntries = new StringBuilder();  
    Set<ContactDetails> sortedDetails =  
        new TreeSet<ContactDetails>(book.values());  
    for(ContactDetails details : sortedDetails) {  
        allEntries.append(details);  
        allEntries.append('\n');  
        allEntries.append('\n');  
    }  
    return allEntries.toString();  
}
```

Código-fonte: AddressBook

```
public ContactDetails[] search(String keyPrefix)
{
    List<ContactDetails> matches =
        new LinkedList<ContactDetails>();
    SortedMap<String, ContactDetails> tail =
        book.tailMap(keyPrefix);
    Iterator<String> it = tail.keySet().iterator();
    boolean endOfSearch = false;
    while(!endOfSearch && it.hasNext()) {
        String key = it.next();
        if(key.startsWith(keyPrefix)) {
            matches.add(book.get(key));
        } else {
            endOfSearch = true;
        }
    }
    ContactDetails[] results =
        new ContactDetails[matches.size()];
    matches.toArray(results);
    return results;
}
```

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?

Exercício

- **Catálogo de endereços**

- Edite a classe *AddressBook* e verifique o código do método *removeDetail*. Você identifica algum problema ?
- Crie uma instância de *AddressBook*. Faça uma chamada a *removeDetail* e informe qualquer string para a chave. O que acontece ?

Um exemplo

- Tentar remover de uma coleção um elemento inexistente acarreta um erro em tempo de execução.
- De quem é a ‘falha’?
 - do objeto cliente por chamar o objeto servidor com um argumento inadequado;
 - do objeto servidor por não saber tratar essa situação.
- 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 influem frequentemente no 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--;
    }
}
```

Exercício

- **Catálogo de endereços**

- Edite a classe *AddressBook* e altere o método *removeDetail* para que verifique se o argumento é uma chave em uso.
- Altere os métodos *addDetail* e *search* para que verifiquem se o argumento é não nulo.
- Altere o método *changeDetail* para que verifique se a chave antiga está em uso e se o novo contato não é nulo.
- Compare o estágio atual do projeto em uso com *address-book-v2g*.

Informe de erro do servidor

- Como informar argumentos inválidos?
 - Para o usuário?
 - Há usuários humanos?
 - Eles podem resolver o problema?
 - Para o objeto cliente?
 - Retorna um valor de diagnóstico.
 - *Lança uma exceção.*

Retornando um diagnóstico

```
public boolean 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.

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.

Exceção

- O termo *exceção* é a abreviatura da expressão “evento excepcional”.
- Uma exceção é um evento, ocorrido durante a execução de um programa, que interrompe o fluxo normal de instruções do programa.
- As informações sobre a exceção são colocadas em um *objeto exceção*.

Exceção

- Lançar uma *exceção* é a maneira mais eficaz que um objeto servidor tem para indicar que não é capaz de atender a uma solicitação de um cliente.
- Há duas etapas (normalmente combinadas) para lançar uma exceção:
 - criar um objeto exceção
 - lançar o objeto exceção

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 correspondência.
 * @throws NullPointerException se a chave for null.
 */
public ContactDetails getDetails(String key)
{
    if(key == null) {
        throw new NullPointerException(
            "null key in getDetails");
    }
    return book.get(key) ;
}
```

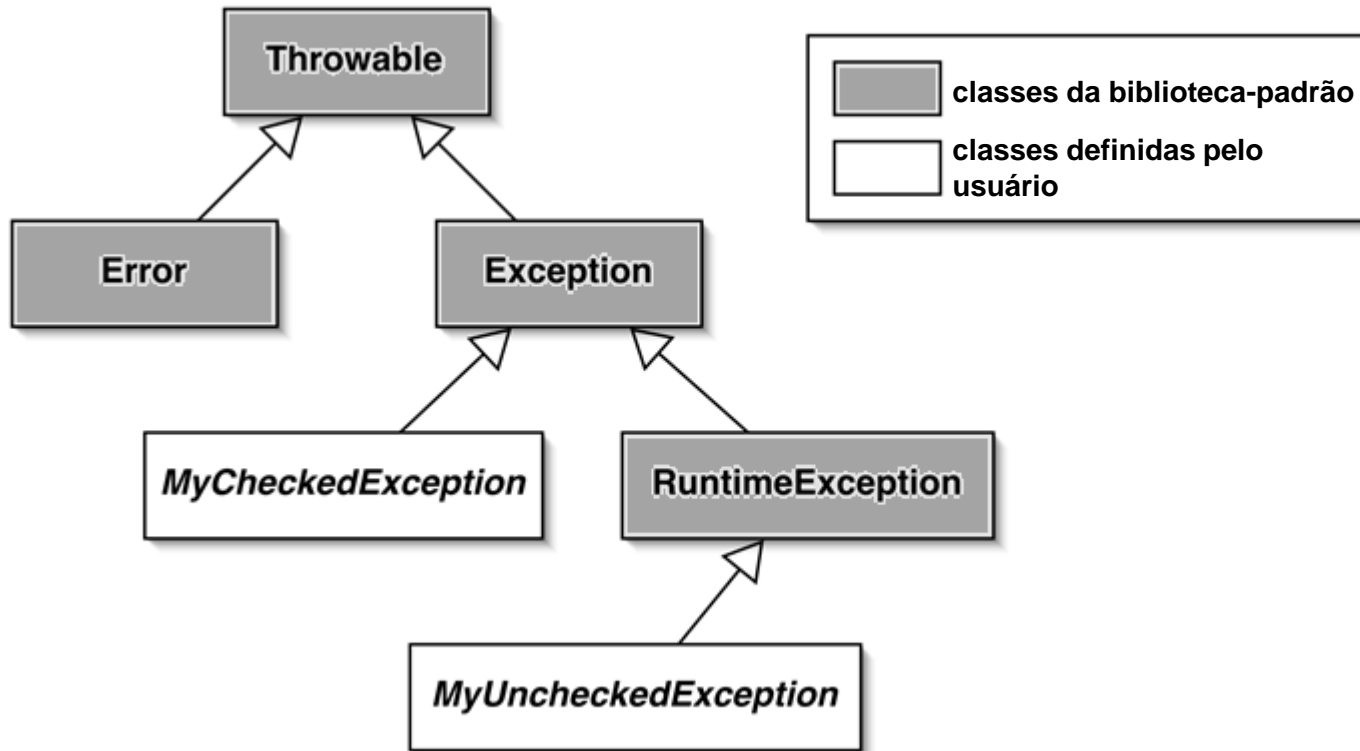
Lançando uma exceção (2)

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

O efeito de uma exceção

- O método chamado (que lança a exceção) termina prematuramente.
- Nenhum valor de retorno é retornado.
- Controle não retorna ao método chamador após o ponto da chamada.
 - Portanto, o cliente não pode prosseguir de qualquer maneira.
- Um cliente pode ‘capturar’ uma exceção.

A hierarquia de classes de exceção



Error

- `Error` e suas subclasses indicam a ocorrência de eventos excepcionais que não podem ser antecipados ou recuperados.
- `IOError` e `OutOfMemoryError` são exemplos típicos.

Categorias de exceção

- Exceções verificadas:
 - subclasse de `Exception`;
 - utilizadas para falhas antecipáveis; 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.

Categorias de exceção

- Exceções verificadas:
 - Concebidas para os casos em que o cliente deve esperar que a operação possa falhar eventualmente.
- Exceções não-verificadas:
 - Concebidas para os casos em que nunca deve ocorrer uma falha em uma operação normal - geralmente elas indicam um descuido do programador.

Exceções não-verificadas

- O compilador não aplica verificações especiais no método em que uma exceção não-verificada é lançada ou no método que o chamou.
- Causam o término do programa se não capturadas.
 - Essa é a prática normal.
- `IllegalArgumentException` e `NullPointerException` são exemplos típicos.

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.");
    }
}
```

Exercício

- **Catálogo de endereços**

- Feche o projeto anterior e abra o projeto *address-book-v2t*.
- Altere os métodos *addDetails*, *changeDetails* e *removeDetails* de *AddressBook* para que lancem uma *IllegalArgumentException* quando recebem argumentos nulos.
- Compare o estágio atual do projeto em uso com *address-book-v3t*.

Exceções verificadas

- Exceções verificadas devem ser capturadas (ou propagadas).
- 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 podem lançar exceções verificadas devem declarar isso incluindo no seu cabeçalho uma cláusula `throws`:

```
public void saveToFile(String destinationFile)  
    throws IOException { ... }
```

Lidando com exceções verificadas

- Clientes que chamam um método que pode lançar exceções verificadas devem escolher entre:
 - propagar as exceções incluindo em seu cabeçalho uma cláusula `throws` igual ao do método chamado, ou
 - proteger a chamada e capturar as exceções com uma instrução `try`.

A instrução *try*

- Clientes que chamam um método que pode lançar exceções verificadas devem proteger a chamada com uma instrução `try`:

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


A instrução *try*

- A instrução `try` protege as chamadas a métodos que podem lançar exceções verificadas, capturando e tratando estas exceções:

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

A instrução *try*

- A instrução `try` tem três partes:
 - a cláusula `try`
 - a cláusula `catch`
 - a cláusula `finally`
- Sendo que:
 - deve ter ao menos uma cláusula `catch` ou `finally` para tratar exceções
 - pode ter várias cláusulas `catch`
 - pode ter apenas uma cláusula `finally`

A instrução *try*

```
try
{
    // Proteja uma ou mais instruções aqui.
}
catch (Exception exception parameter e)
{
    // Informe e recupere a exceções aqui.
}
finally
{
    // Especifique quaisquer ações comuns aqui,
    // quer uma exceção seja lançada ou não.
}
```

Diagram illustrating the structure of a `try` statement with annotations:

- `try`: The start of the try block.
- `{`: The opening brace of the `try` block, labeled **bloco try**.
- `// Proteja uma ou mais instruções aqui.`: Comment indicating the protected code block.
- `catch (Exception e)`: The catch clause, where `e` is the *exception parameter*.
- `{`: The opening brace of the `catch` block, labeled **bloco catch**.
- `// Informe e recupere a exceções aqui.`: Comment indicating the recovery code.
- `finally`: The finally block, used for cleanup or common actions.
- `{`: The opening brace of the `finally` block, labeled **bloco finally**.
- `// Especifique quaisquer ações comuns aqui,`
`// quer uma exceção seja lançada ou não.`: Comments indicating the common actions performed regardless of whether an exception was thrown.
- `}`: The closing brace of the `finally` block.

Capturando uma exceção

1. Exceção lançada a partir daqui.

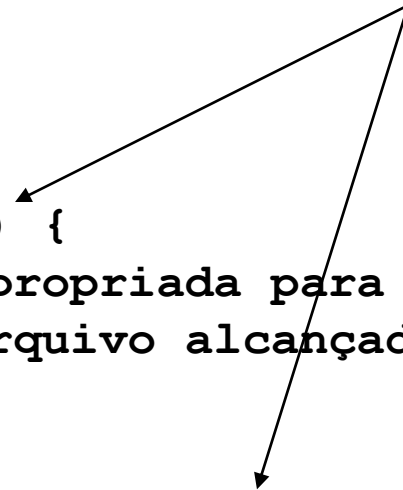
```
try{  
    addressbook.saveToFile(filename) ;  
    ... ← Atenção: demais instruções não serão executadas !  
}  
catch(IOException e) {  
    System.out.println("Unable to save to " + filename);  
}
```

2. Controle transferido para cá.

Capturando múltiplas exceções

```
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 arquivo não encontrado.  
    ...  
}
```

Atenção: as cláusulas
catch são verificadas
na ordem especificada



O bloco *catch*

- Cada bloco `catch` só é executado se for lançado um objeto exceção do mesmo tipo do parâmetro especificado após a palavra-chave `catch`.

O bloco *finally*

- O bloco `finally` (quase sempre omitido) é sempre executado, sendo ou não lançada uma exceção.

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.

Definindo novas classes de exceção

```
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.";
    }
}
```

Exercício

- **Catálogo de endereços**

- Feche o projeto anterior e abra o projeto *address-book-v3t*.
- Observe que foi definida a classe *NoMatchingDetailsException*. Ela é uma exceção verificada ou não-verificada ?
- Altere os métodos *removeDetails* e *addDetails* para que lancem uma *NoMatchingDetailsException* caso o contato informado seja nulo.

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.

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;
}
```

A instrução *assert*

- A instrução `assert` realiza testes internos de consistência.
- A palavra-chave `assert` é seguida por uma expressão booleana que deve ser verdadeira (caso contrário será lançado um `AssertionError`) e um `string` (opcional) :

```
assert boolean_expression : string_expression;
```

A instrução *assert*


- Por padrão, `assert` vem desabilitado. Para usar o recurso é necessário ativá-lo.
 - compilando:
`javac -source 1.5 classe.java`
 - habilitando:
`java ea classe ou`
`java enableassertions classe`
 - desabilitando:
`java da classe ou`
`java disableassertions classe`

A instrução *assert*

- A instrução `assert` serve para dois propósitos:
 - detecta erros de programação, e
 - declara explicitamente nossas premissas, aumentando a legibilidade do código.
- Atenção: como pode estar desabilitada, não deve ser usada para executar algo necessário para a correta operação do método (por exemplo, para verificar argumentos de métodos públicos).

Utilizando assertivas

```
public void removeDetails(String key)
{
    if(key == null){
        throw new IllegalArgumentException();
    }
    if(keyInUse(key)) {
        ContactDetails details = book.get(key);
        book.remove(details.getName());
        book.remove(details.getPhone());
        numberOfEntries--;
    }
    assert !keyInUse(key);
    assert consistentSize()
        : "Inconsistent book size in removeDetails";
}
```



Mensagem informativa passada para o construtor de `AssertionError`

Utilizando assertivas

```
public class AddressBook
{
    private TreeMap<String, ContactDetails> book;
    private int numberOfEntries;
    ...
    private boolean consistentSize(String key)
    {
        Collection<ContactDetails> allEntries =
            book.values();
        Set Collection<ContactDetails> uniqueEntries =
            new HashSet<ContactDetails>(allEntries);
        int actualCount = uniqueEntries.size();
        return numberOfEntries == actualCount;
    }
    ...
}
```

Exercício

- **Catálogo de endereços**

- Feche o projeto anterior, abra o projeto *address-book-assert* e crie uma instância de *AddressBookDemo*.
- Edite a classe *AddressBookDemo*, examine seus vários métodos de teste.
- Depois de entender seu funcionamento, execute estes métodos. Algum erro de assertiva é gerado ? Porque ?

Exercício

- **Catálogo de endereços**

- Edite a classe *AddressBook* e examine todas as instruções *assert*.
- Altere o método *changeDetails* para incluir uma assertiva de que o catálogo de endereços deve conter no fim do método a mesma quantidade de entradas do início.

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 dados textuais (legíveis por humanos).
 - Com base no tipo `char`.
- Fluxos lidam com dados binários.
 - Com base no tipo `byte`.

Saída de texto

- Utiliza a classe `FileWriter`, combinada com `PrintWriter` (para saída baseada em linha).
 - Abrir o arquivo (construtor `FileWriter`).
 - Gravar no arquivo (método `println`).
 - Fechar o arquivo (método `close`).
- Falha em um ponto qualquer resulta em uma `IOException`.

Saída de texto

```
try {  
    PrintWriter writer = new PrintWriter(  
        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  
}
```

Código-fonte: Database

```
public void writeBasicDetails(String filename)
{
    try {
        PrintWriter writer = new PrintWriter(
            new FileWriter(filename));
        for(Item item : items) {
            writer.println(item.getBasicDetails());
        }
        writer.close();
    }
    catch(IOException e) {
        System.err.println("Failed to save to "
            + filename);
    }
}
```


Exercício

- **Database multimídia**

- Feche o projeto anterior e abra o projeto *dome-simple-output* e crie uma instância de *Database*. Crie algumas instâncias de *CD* e de *DVD* e as adicione ao *Database*.
- Chame o método *writeBasicDetails* para gravar os detalhes básicos de seus itens. Edite o arquivo e verifique o seu conteúdo.

Entrada de texto

- Utiliza a classe `FileReader`, combinada com `BufferedReader` (para entrada baseada em linha).
 - Abrir o arquivo (construtor `FileReader`).
 - Ler do arquivo (método `readLine`).
 - Fechar o arquivo (método `close`).
- 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) {      ← Linha nula representa EOF  
        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  
}
```

Código-fonte: Responder

```
private void fillDefaultResponses()  
{  
    try {  
        BufferedReader reader = new BufferedReader(  
            new FileReader(RESPONSES));  
        String response = reader.readLine();  
        while(response != null) {  
            defaultResponses.add(response);  
            response = reader.readLine();  
        }  
        reader.close();  
    }  
    catch(IOException e) {  
        System.err.println("A problem was encountered in"  
            + RESPONSES);  
    }  
}
```

Exercício

- **Sistema de suporte técnico**

- Feche o projeto anterior e abra o projeto *tech-support-io*.
- Edite a classe *Responder* e examine o método *fillDefaultResponses*.
- Depois de entender seu funcionamento, crie um Sistema de Suporte e teste-o.
- Crie um arquivo texto com respostas para problemas e altere a classe *Responder* para que o use. Teste o funcionamento do Sistema de Suporte.

Serialização de objetos

- A serialização permite que um objeto seja:
 - gravado em um arquivo em uma única operação
 - reconstruído em uma única operação a partir do arquivo em que foi gravado
- Para isto, o objeto deve implementar a interface `Serializable` (que não define nenhum método).

Código-fonte: AddressBookFileHandler

```
public void saveToFile(String destinationFile)
    throws IOException
{
    File destination =
        makeAbsoluteFilename(destinationFile);
    ObjectOutputStream os = new ObjectOutputStream(
        new FileOutputStream(destination));
    os.writeObject(book);
    os.close();
}
```


Código-fonte: AddressBookFileHandler

```
public AddressBook readFromFile(String sourceFile)
    throws IOException, ClassNotFoundException
{
    ...
    try {
        File source = new File(resource.toURI());
        ObjectInputStream is = new ObjectInputStream(
            new FileInputStream(source));
        AddressBook savedBook =
            (AddressBook) is.readObject();
        is.close();
        return savedBook;
    }
    catch (URISyntaxException e) {
        throw new IOException(
            "Unable to make a filename for " + sourceFile);
    }
}
```


Exercício

- **Catálogo de endereços**

- Feche o projeto anterior e abra o projeto *address-book-io*.
- Crie uma instância de *AddressBookDemo* e uma de *AddressBookFileHandler*.
- Chame o método *saveToFile* e informe um nome de arquivo. Edite o arquivo.
- Chame o método *readFromFile* e informe o nome do arquivo acima.
- Edite a classe *AddressBook*, comente a implementação de *Serializable* e chame o método *saveToFile*. O que ocorre ?

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 frequentemente levam a falhas de programa.
- Programação defensiva antecipa erros — tanto no cliente como no servidor.
- Exceções fornecem um mecanismo de informe e recuperação de falhas.
- Assertivas fornecem um mecanismo de teste de consistência.

Revisão (3)

- Entrada e saída são particularmente propensas a erros.
- Processamento de arquivos envolve:
 - Abrir o arquivo;
 - Processar os registros (ler ou gravar);
 - Fechar o arquivo.
- Falha em um ponto qualquer resulta em uma `IOException`.
- O pacote `java.io` suporta entrada e saída.



Contatos

Câmara dos Deputados
CENIN - Centro de Informática

Carlos Renato S. Ramos

carlosrenato.ramos@camara.gov.br