

# Programação Orientada por Objetos

## Exceções



Prof. Cédric Grueau

Prof. José Sena Pereira

Departamento de Sistemas e Informática  
Escola Superior de Tecnologia de Setúbal  
Instituto Politécnico de Setúbal

2022/2023

# Sumário

- ▶ Aplicação Agenda de Endereços
- ▶ Gestão de Erros
- ▶ Mecanismo de Exceções
- ▶ Definição de Exceções



# Agenda de Endereços

- ▶ Exceções



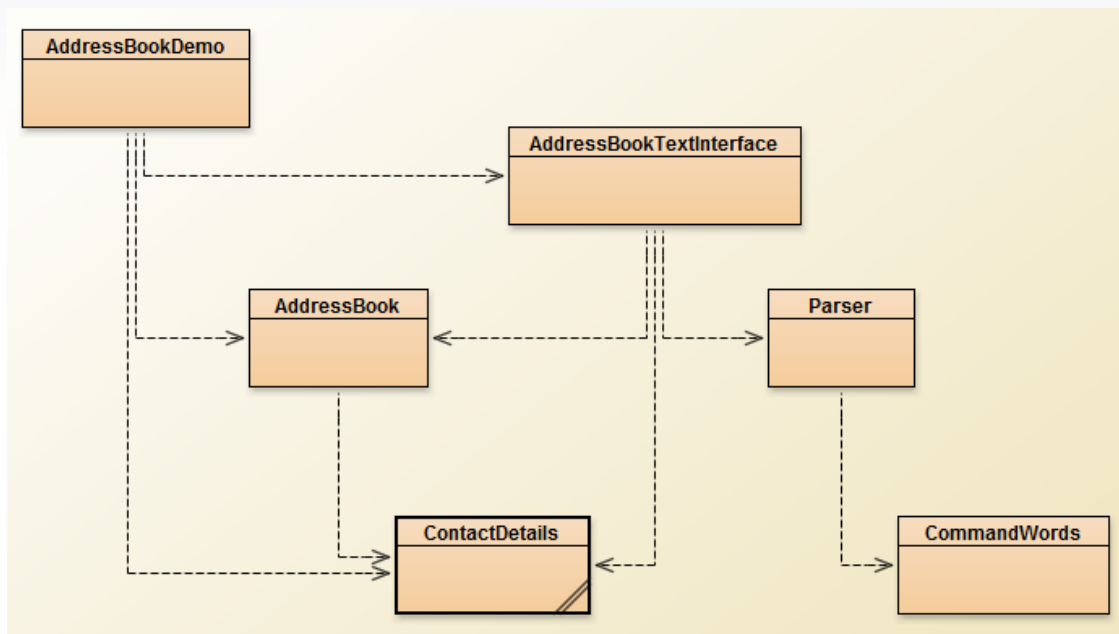
# Exemplo – Address Book

- ▶ Criar uma aplicação para guardar contactos.
  - ▶ Cada contacto regista a informação do nome, telefone e endereço.
  - ▶ Deve ser possível efetuar as operações habituais de criação, listagem, alteração e remoção de contactos (operações CRUD).
  - ▶ Deve existir uma forma de procurar contactos pelo nome ou telefone.
  - ▶ Criar uma interface de consola para a aplicação.



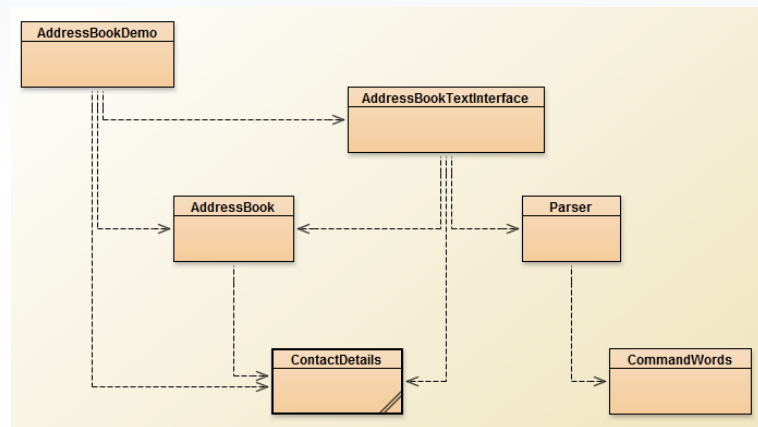
# Exemplo – AddressBook

- ▶ Diagrama de classes da aplicação **Address Book**:



# Exemplo – AddressBook

- ▶ Classes principais da aplicação **AddressBook**:
  - ▶ **ContactDetails** – Informação do contacto.
  - ▶ **AddressBook** – Lista de contactos.
- ▶ Classes da interface
  - ▶ **CommandWords** – Define os comandos que podem ser dados na consola.
  - ▶ **Parser** – Lê a informação da consola e interpreta-a retornando-a como um objeto **Command**.
  - ▶ **AddressBookTextInterface** – A aplicação em ambiente de consola.
  - ▶ **AddressBookTextDemo** – Corre uma demonstração com alguns contactos já definidos.



# Exemplo – AddressBook

## ► Classe **ContactDetails**

```
public class ContactDetails implements Comparable<ContactDetails> {  
    private String name;  
    private String phone;  
    private String address;  
    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();  
    }  
    // Continua...
```

# Exemplo – AddressBook

- ▶ Classe **ContactDetails** – métodos **seletores** e **toString**

```
public String getName() {  
    return name;  
}  
public String getPhone() {  
    return phone;  
}  
public String getAddress() {  
    return address;  
}  
public String toString() {  
    return name + "\n" + phone + "\n" + address;  
}
```



# Exemplo – AddressBook

- ▶ Classe **ContactDetails** – métodos **equals** e **hashCode**

```
public boolean equals(Object other) {
    if (other instanceof ContactDetails) {
        ContactDetails otherDetails = (ContactDetails) other;
        return name.equals(otherDetails.getName()) &&
            phone.equals(otherDetails.getPhone()) &&
            address.equals(otherDetails.getAddress());
    }
    else {
        return false;
    }
}

public int hashCode() {
    int code = 17; code = 37 * code + name.hashCode();
    code = 37 * code + phone.hashCode();
    code = 37 * code + address.hashCode();
    return code;
}
```

# Exemplo – AddressBook

- ▶ Classe **ContactDetails** - método **compareTo**

Implementação da interface  
Comparable<ContactDetails>

```
public int compareTo(ContactDetails otherDetails) {  
    int comparison = name.compareTo(otherDetails.getName());  
    if (comparison != 0) {  
        return comparison;  
    }  
    comparison = phone.compareTo(otherDetails.getPhone());  
    if (comparison != 0) {  
        return comparison;  
    }  
    return address.compareTo(otherDetails.getAddress());  
}
```

Vai permitir a ordenação por:  
1-nome, a seguir  
2-telefone e a seguir  
3-endereço

# Exemplo – AddressBook

- ▶ Classe **AddressBook** e método **addDetails**

```
public class AddressBook {  
  
    private TreeMap<String, ContactDetails> book;  
    private int numberOfEntries;  
  
    public AddressBook() {  
        book = new TreeMap<String, ContactDetails>();  
        numberOfEntries = 0;  
    }  
  
    public void addDetails(ContactDetails details){  
        book.put(details.getName(), details);  
        book.put(details.getPhone(), details);  
        numberOfEntries++;  
    }  
    // Continua...
```

Armazena os contactos  
numa coleção TreeMap

Usa como chaves  
simultaneamente o nome e  
o telefone para facilitar a  
procura

Porquê?

Como tem duas entradas por  
contacto usa separadamente um  
contador de contactos

# Exemplo – AddressBook

- ▶ Classe **AddressBook** – métodos **getDetails**, **KeyInUse** e **changeDetails**

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

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

```
public void changeDetails(String oldKey, ContactDetails details){  
    removeDetails(oldKey);  
    addDetails(details);  
}
```

# Exemplo – AddressBook

- ▶ Classe **AddressBook** – métodos **getNumberOfEntries** e **removeDetails**

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

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

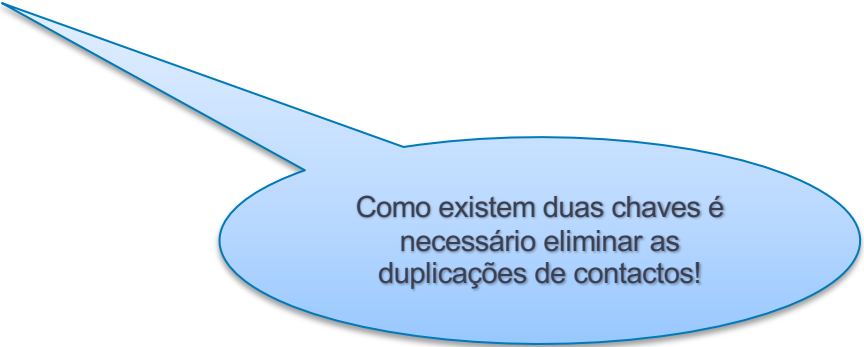


Como funciona?

# Exemplo – AddressBook

- ▶ Classe **AddressBook** - método **listDetails**

```
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();  
}
```



Como existem duas chaves é necessário eliminar as duplicações de contactos!

# Exemplo – AddressBook

- ▶ Classe **AddressBook** – método **search**

```
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;  
}
```

Chaves maiores ou  
iguais ao prefixo

Devolve um array com  
os contactos

# Exemplo – Addressbook

- ▶ Análise da aplicação **AddressBook**
  - ▶ A aplicação está funcional mas estamos a assumir que tudo corre bem ou não devemos assumir isso?
    - ▶ Neste caso um objeto **AddressBook** é um objeto servidor típico.
      - ▶ Não inicia ações, toda a sua atividade é em resposta a pedidos de clientes.
    - ▶ Se pensarmos que o objeto servidor terá clientes que poderão cometer erros inadvertidamente ou mesmo intencionalmente, a implementação nestes casos terá que ser diferente.
    - ▶ A maior **vulnerabilidade** num objeto servidor está nos **argumentos dos métodos**
      - ▶ Os argumentos dos construtores que inicializam o estado do objeto
      - ▶ Os argumentos dos métodos que contribuem para o comportamento do objeto
    - ▶ Devemos colocar as seguintes **questões** para lidar com as vulnerabilidades:
      - ▶ Que verificações devem ser feitas nos métodos do servidor?
      - ▶ Como reportar os erros aos clientes?
      - ▶ Como devem os clientes antecipar problemas e falhas nos pedidos ao servidor?
      - ▶ Como é que os clientes devem lidar com as falhas nos pedidos ao servidor?



# Gestão de Erros

- ▶ Exceções



# Exemplo – Address Book

- ▶ Criar uma aplicação para guardar contactos.
  - ▶ Cada contacto regista a informação do nome, telefone e endereço.
  - ▶ Deve ser possível efetuar as operações habituais de criação, listagem, alteração e remoção de contactos (operações CRUD).
  - ▶ Deve existir uma forma de procurar contactos pelo nome ou telefone.
  - ▶ Criar uma interface de consola para a aplicação.



# Exemplo – Addressbook

- ▶ Análise da aplicação **AddressBook**
  - ▶ A aplicação está funcional mas estamos a assumir que tudo corre bem ou não devemos assumir isso?
    - ▶ A **vulnerabilidade** nos **argumentos dos métodos**
      - ▶ Os argumentos dos construtores que inicializam o estado do objeto
      - ▶ Os argumentos dos métodos que contribuem para o comportamento do objeto
    - ▶ Devemos colocar as seguintes **questões** para lidar com as vulnerabilidades:
      - ▶ Que verificações devem ser feitas nos métodos do servidor?
      - ▶ Como reportar os erros aos clientes?
      - ▶ Como devem os clientes antecipar problemas e falhas nos pedidos ao servidor?
      - ▶ Como é que os clientes devem lidar com as falhas nos pedidos ao servidor?

# Exemplo – Addressbook

## ► Análise da aplicação AddressBook – método **removeDetails**

### ► Exemplo de uma vulnerabilidade:

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

1. Exemplo - chamado com *null*:  
*addressBook.removeDetails(null)*

2. *book.get(key)* retorna *null*:  
*details* fica com o valor *null*

3. O programa termina com uma exceção na chamada a *details.getName()*:  
*java.lang.NullPointerException*  
at java.util.TreeMap.getEntry(TreeMap.java:347)  
at java.util.TreeMap.get(TreeMap.java:278)  
at AddressBook.removeDetails(AddressBook.java:121)

1. Se a chave não existir **book.get(key)** retorna **null**

► Não é aqui que está o problema

2. A seguir **details.getName()** origina um erro durante a execução

► Neste caso **details** tem o valor **null**, a chamada a métodos a partir deste objeto leva a que o programa termine com uma mensagem de exceção

► De quem é a culpa? Do servidor que não verificou o argumento passado ou do cliente que passou um valor errado?

# Exemplo – Addressbook

- ▶ Análise da aplicação **AddressBook**
  - ▶ No exemplo:
    - ▶ Criamos um objeto **AddressBook**.
    - ▶ Removemos um contacto ( **removeDetails** )
    - ▶ A aplicação reporta um erro na execução.
      - ▶ De quem é a culpa deste erro?
    - ▶ É preferível anteciparmos esta situação do que passar por este problema.
  - ▶ A maior vulnerabilidade num objeto servidor está nos argumentos dos métodos
    - ▶ Os argumentos dos construtores que inicializam o estado do objeto
    - ▶ Os argumentos dos métodos que contribuem para o comportamento do objeto
  - ▶ A **verificação dos argumentos** é o que se chama uma **medida defensiva**

# Exemplo – Addressbook

- ▶ Análise da aplicação **AddressBook** – método **removeDetails**
  - ▶ Uma solução simples neste caso é fazer a verificação e não agir se a chave a procurar não existir.

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

# Exemplo – Addressbook

- ▶ Análise da aplicação **AddressBook** – outros métodos
  - ▶ Problemas idênticos existem noutros métodos:
    - ▶ **void** **addDetails**(**ContactDetails** details)
      - ▶ Não verifica se o parâmetro **details** vem com **null**
    - ▶ **void** **changeDetails**(**String** oldKey, **ContactDetails** details)
      - ▶ **Oldkey** devia existir e **details** não deveria ter **null**
    - ▶ **ContactDetails[]** **search**(**String** keyPrefix)
      - ▶ **keyPrefix** não deve vir a **null**
  - ▶ Mesmo que se protejam os métodos com verificações o problema não fica totalmente resolvido. É conveniente em casos destes avisar a aplicação cliente ou mesmo o utilizador. Qual a melhor maneira de o fazer?
    - ▶ A resposta é: depende! Não existe uma solução única.

# Reporte de erros

- ▶ Solução 1: **Notificar o utilizador**
  - ▶ Através duma mensagem de erro escrita no ecrã ou numa janela de alerta.
- ▶ Problemas da solução:
  - ▶ Estamos a assumir que existe um utilizador humano com acesso à mensagem.
    - ▶ Nem sempre é verdade. A aplicação do utilizador pode estar a correr num computador diferente daquele que tem os dados. Pode não existir acesso a um dispositivo de visualização.
  - ▶ Mesmo que o utilizador tenha acesso à informação do erro será que ele pode de alguma forma corrigi-lo?
    - ▶ A maior parte das vezes o utilizador não tem meios para corrigir o erro.
      - ▶ O que pode fazer um utilizador se estiver num terminal de multibanco e receber uma mensagem **NullPointerException**? 😊



# Reporte de erros

- ▶ Solução 2: **Notificar a aplicação cliente** através de um **valor de retorno** do método que está a ser chamado.
  - ▶ No objeto servidor:

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

Removido com **sucesso**

**Falhou** a remoção

# Reporte de erros

- ▶ Solução 2: **Notificar a aplicação cliente** através de um **valor de retorno** do método que está a ser chamado.

- ▶ No método cliente:

```
if(contacts.removeDetails ( "...")) {  
    // Contacto removido com sucesso.  
    // Continuar normalmente. ...  
}  
else {  
    // Falhou a remoção.  
    // Tentar uma ação para recuperar do  
    problema se for possível  
}
```

Removido com **sucesso**  
Continuar normalmente

**Falhou** a remoção  
Tentar recuperar do problema

# Reporte de erros

- ▶ Solução 2: **Notificar a aplicação cliente** através de um **valor de retorno** do método que está a ser chamado.
- ▶ Problemas da solução:
  - ▶ Por vezes o valor de retorno não permite a utilização dum valor específico para o erro.
    - ▶ Exemplo: um método **double getBalance(int accountNumber)** que retorna o saldo duma conta bancária. Como definir um valor de retorno a reportar que o número da conta está errado?
  - ▶ Mesmo que exista o retorno com a informação do erro como se garante que a aplicação cliente vai verificar e utilizar esse valor de retorno?
    - ▶ A aplicação pode decidir ignorar o valor de retorno do método:

```
contacts.removeDetails ("..."); // Sem teste ao retorno  
// Contacto removido com ou sem sucesso.  
// Continuar normalmente.
```

# mecanismo de Exceções

- ▶ Exceções



# Exemplo – Addressbook

- ▶ Análise da aplicação **AddressBook**
  - ▶ A aplicação está funcional mas estamos a assumir que tudo corre bem ou não devemos assumir isso?
    - ▶ A **vulnerabilidade** nos **argumentos dos métodos**
      - ▶ Os argumentos dos construtores que inicializam o estado do objeto
      - ▶ Os argumentos dos métodos que contribuem para o comportamento do objeto
    - ▶ Devemos colocar as seguintes **questões** para lidar com as vulnerabilidades:
      - ▶ Que verificações devem ser feitas nos métodos do servidor?
      - ▶ Como reportar os erros aos clientes?
      - ▶ Como devem os clientes antecipar problemas e falhas nos pedidos ao servidor?
      - ▶ Como é que os clientes devem lidar com as falhas nos pedidos ao servidor?

# Reporte de erros

- ▶ **Solução 1: Notificar o utilizador**

- ▶ Através duma mensagem de erro escrita no ecrã ou numa janela de alerta.

- ▶ Problemas da solução:

- ▶ Estamos a assumir que existe um utilizador humano com acesso à mensagem.
- ▶ Mesmo que o utilizador tenha acesso à informação do erro será que ele pode de alguma forma corrigi-lo?

- ▶ **Solução 2: Notificar a aplicação cliente** através de um **valor de retorno** do método que está a ser chamado.

- ▶ Problemas da solução:

- ▶ Por vezes o valor de retorno não permite a utilização dum valor específico para o erro.
- ▶ Mesmo que exista o retorno com a informação do erro como se garante que a aplicação cliente vai verificar e utilizar esse valor de retorno?

# Reporte de erros – Exceções

- ▶ **Solução 3:** Notificar a aplicação cliente através da utilização do mecanismo de Exceções.
  - ▶ Em caso de erro é lançada uma exceção a informar do erro e a aplicação cliente é obrigada a tratar esse erro se não quiser evitar que a aplicação termine abruptamente reportando a informação da exceção.
  - ▶ Uma Exceção é um sinal gerado pela máquina virtual de Java em tempo de execução, que é enviado ao programa indicando a ocorrência de um erro recuperável.
  - ▶ **Funcionamento** do mecanismo de exceções:
    - ▶ Quando existe um erro é criado um objeto com a informação do problema.
    - ▶ A aplicação é interrompida e é passado à aplicação o objeto criado com a informação do problema. Diz-se que foi lançada uma Exceção.
    - ▶ O controlo do programa passa depois para um bloco de tratamento do erro caso este tenha sido criado, caso contrário a aplicação é interrompida e é reportada a exceção ocorrida.
    - ▶ Depois do erro tratado a execução da aplicação continua normalmente.

# Reporte de erros

- ▶ Exemplo do lançamento duma exceção:

```
public ContactDetails getDetails(String key) {  
    if (key == null) {  
        throw new IllegalArgumentException("null key in getDetails");  
    }  
    return book.get(key);  
}
```



Lançamento de uma  
exceção

- ▶ Neste caso a exceção é lançada pelo método servidor
- ▶ Também é possível ser a máquina virtual do Java a lançar internamente a exceção
  - ▶ Foi o caso no exemplo anterior com a exceção **NullPointerException** que foi lançada pela JVM



# Lançamento de uma exceção

Palavra reservada  
**throw**

Mensagem de erro  
**null key in getDetails**

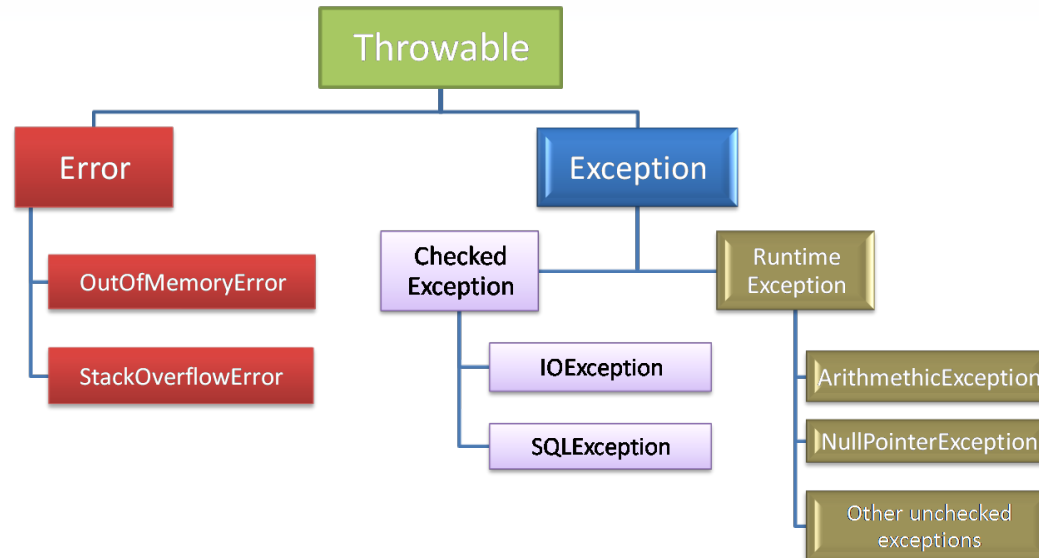
```
throw new IllegalArgumentException("null key in getDetails");
```

Nome da classe da exceção  
**IllegalArgumentException**

- ▶ Lançamento de uma exceção
  - ▶ É criado o objeto com a exceção:
    - ▶ **new ExceptionType("...")**
  - ▶ O objeto da exceção é "lançado" (thrown):
    - ▶ **throw ...**
  - ▶ Na documentação em Javadoc:
    - ▶ **@throws ExceptionType** explicação do motivo ...

# Classe das exceções

- ▶ As exceções lançadas são representadas por classes que formam uma hierarquia com base na classe **Exception**:
  - ▶ Para cada tipo de exceção existe uma classe própria
  - ▶ Todas as classes de exceção têm por convenção o sufixo **Exception**



# Consequências das exceções

- ▶ Quando uma **exceção é lançada** o método onde está termina imediatamente.
  - ▶ Não existe valor de retorno
  - ▶ O controlo da aplicação não volta ao ponto onde o método foi chamado
    - ▶ Neste caso a aplicação não pode continuar ignorando o que aconteceu
  - ▶ Mas a aplicação pode **capturar a exceção**

```
AddressBook book = new AddressBook();
```

```
ContactDetails contact = book.getDetails(null);  
System.out.println("Details: " + contact);
```

Chamada ao método `getDetails`  
passando `null`

O programa **não continua** na  
instrução seguinte

```
public ContactDetails getDetails(String key) {  
    if (key == null) {  
        throw new IllegalArgumentException("null key in getDetails");  
    }  
    return book.get(key);  
}
```

Lançamento da exceção:  
**O método termina imediatamente**  
sem valor de retorno

# Exemplo – AddressBook

- ▶ Através do **lançamento de exceções** podemos proteger os nossos métodos de uma utilização errada

```
public void changeDetails(String oldKey, ContactDetails details){  
    if(details == null) {  
        throw new IllegalArgumentException("Null details passed to changeDetails.");  
    }  
    if(oldKey == null){  
        throw new IllegalArgumentException("Null key passed to changeDetails.");  
    }  
    if(keyInUse(oldKey)){  
        removeDetails(oldKey);  
        addDetails(details);  
    }  
}
```

- ▶ Nestes casos estamos a utilizar as exceções que existem no Java.
  - ▶ É o caso da exceção **IllegalArgumentException**
- ▶ Se a exceção que lançamos não for capturada o programa será terminado.

# Exemplo – AddressBook

- ▶ Através do **lançamento de exceções** podemos igualmente impedir que os objetos sejam criados

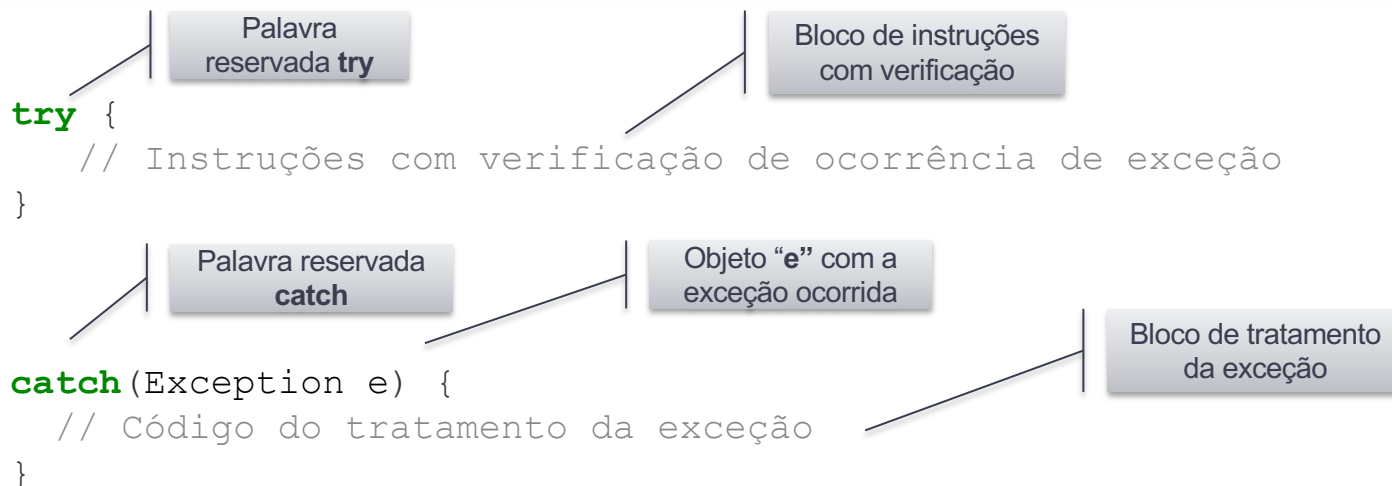
```
public ContactDetails(String name, String phone, String address) {  
    if(name == null) {  
        name = "";  
    }  
    if(phone == null) {  
        phone = "";  
    }  
    if(address == null) {  
        address = "";  
    }  
    this.name = name.trim();  
    this.phone = phone.trim();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

Lançamento da  
exceção:  
**O objeto não é criado**

Exceção do tipo `IllegalStateException`:  
Exceção do Java **não verificada**

# Tratamento de exceções

- ▶ Se quisermos impedir que o programa termine temos de **capturar a exceção** que ocorreu. Neste caso estamos a tratar a exceção.
- ▶ O **tratamento de exceções** é feito usando um bloco **try-catch**
  - ▶ O código que é verificado quanto à ocorrência de exceções está dentro dum bloco **try**
  - ▶ O código que é executado quando a exceção ocorre está dentro de um bloco **catch**



# Tratamento de exceções

- Exemplo do **tratamento** **duma exceção**:

```
AddressBook book = new AddressBook();  
try {  
    ContactDetails contact = book.getDetails(null);  
    System.out.println("Details: " + contact);  
}  
catch(IllegalArgumentException e) {  
    System.out.println("A chave deu problemas: " + e.getMessage());  
    // ...  
}
```

1. Exceção lançada daqui

2. Execução transferida para aqui

e - vem com o objeto de exceção criado em `getDetails`

O método `getMessage()` retorna a mensagem de erro que foi criada com a exceção

# Tratamento de exceções

- ▶ Podem existir vários blocos **catch** de tratamento de exceções
  - ▶ Cada bloco processa o tratamento de um tipo de exceção
  - ▶ Apenas um dos blocos é executado quando ocorre uma exceção dentro do bloco **try**
  - ▶ O primeiro que for do tipo da exceção que foi lançada será o que é executado
  - ▶ Como existe uma hierarquia de exceções, as exceções mais genéricas devem ser as últimas a ser capturadas

```
try {  
    ...  
    ref.process();  
    ...  
}  
catch (EOFException e) {  
    // código de tratamento da exceção end-of-file  
    ...  
}  
catch (FileNotFoundException e) {  
    // código de tratamento da exceção file-not-found  
    ...  
}  
catch (Exception e) {  
    // código de tratamento da exceção genérica (Exception)  
    ...  
}
```

Exceção mais genérica



# Tratamento de exceções

- ▶ Podem existir vários blocos **catch** de tratamento de exceções
  - ▶ **Multi-catch** (a partir do Java 7)
    - ▶ Permite usar o mesmo bloco de **catch** para mais do que um tipo de exceção:

```
try {  
    ...  
    ref.process();  
    ...  
}  
catch (EOFException | FileNotFoundException e) {  
    // código de tratamento para ambos os tipos de exceção  
    ...  
}
```

# Tratamento de exceções

- ▶ A clausula **finally**
  - ▶ A clausula **finally** é um bloco de código que aparece depois do(s) bloco(s) **catch** e que é **executado sempre**, exista ou não exista exceção

```
try {  
    // Instruções com verificação de ocorrência de exceção  
}  
catch(Exception e){  
    // Código do tratamento da exceção  
}  
finally{  
    // Ações comuns que devem ser executadas haja ou não haja exceção.  
}
```

# Exceções – Procura Ascendente de um Catch

- Quando o método que lança a exceção não tem um **catch** para a exceção lançada, a procura do bloco **catch** adequado propaga-se pelos métodos clientes até se encontrar um **catch** para essa exceção ou se atingir o método **main** e terminar o programa.

```
public void static main(int[] args) {  
    metodo1();  
}
```

```
metodo1() {  
    try {  
        metodo2();  
    }  
    // tem catch para exceção "e"  
}
```

Procura o **catch** para "e": Encontra e executa

```
metodo2() {  
    metodo3();  
    // não tem catch para exceção "e"  
}
```

Procura o **catch** para "e": Não encontra vai procurar no método cliente

```
metodo3() {  
    // método onde ocorre ou que lança uma exceção "e"  
}
```

# Definição de Exceções

- ▶ Exceções



# Categorias de exceções

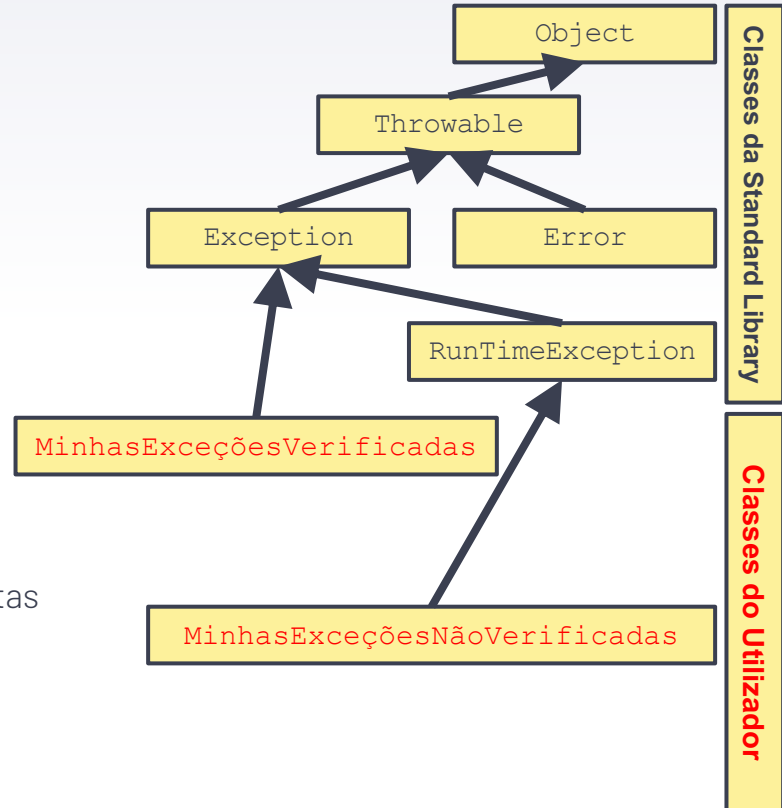
- ▶ Existem duas **categorias de exceções**:

- ▶ Exceções **verificadas**

- ▶ Subclasses de **Exception**
    - ▶ Usadas para falhas que se preveem
    - ▶ A recuperação poderá ser possível

- ▶ Exceções **não verificadas**

- ▶ Subclasses de **RuntimeException**
    - ▶ Usadas para falhas que não são previstas
    - ▶ A recuperação é pouco provável



# Exceções verificadas

- ▶ As exceções **não verificadas** podem ou não ser capturadas e tratadas pela aplicação
  - ▶ Se acontecerem e não forem tratadas a aplicação termina
- ▶ As exceções **verificadas** são feitas para serem capturadas e tratadas pela aplicação
  - ▶ O compilador assegura que são capturadas dando erro de compilação se isso não acontecer.
  - ▶ Estas exceções se forem convenientemente tratadas as falhas serão recuperáveis
  - ▶ Os métodos que lançarem exceções verificadas devem incluir na assinatura do método a palavra reservada **throws** seguida da exceção ou exceções que lançam (separadas por vírgulas)
    - ▶ Exemplo:

```
public void saveToFile(String destinationFile) throws IOException {  
  
    // código do método  
  
}
```

**IOException:**  
É uma **exceção verificada** do Java  
que pode ser lançada neste método

# Exceções – Definição de novas exceções

- ▶ Para definir novas exceções:
  - ▶ Cria-se uma classe derivada de **RuntimeException** para criar uma nova exceção não verificada pelo compilador
  - ▶ Cria-se uma classe derivada de **Exception** para uma exceção verificada pelo compilador.
- ▶ A definição de novas exceções oferece uma informação mais precisa do problema.

```
public class NoMatchingDetailsException extends Exception {  
    public NoMatchingDetailsException(String message){  
        super(message);  
    }  
    public String toString(){  
        return "No details matching the key were found.";  
    }  
}
```

- ▶ No método **servidor**:

```
public ContactDetails getDetails(String key) {  
    if (key == null){  
        throw new NoMatchingDetailsException("null key in getDetails");  
    }  
    return book.get(key);  
}
```

# Exceções – Definição de novas exceções

```
public class NoMatchingDetailsException extends Exception {  
    public NoMatchingDetailsException(String message) {  
        super(message);  
    }  
    public String toString(){  
        return "No details matching the key were found.";  
    }  
}
```

- ▶ No método servidor:

```
public ContactDetails getDetails(String key) {  
    if (key == null) {  
        throw new NoMatchingDetailsException("null key in getDetails");  
    }  
    return book.get(key);  
}
```

- ▶ No método cliente:

```
AddressBook book = new AddressBook();  
try {  
    ContactDetails contact = book.getDetails(null);  
    System.out.println("Details: " + contact);  
}  
catch(Exception e) {  
    System.out.println("A chave deu problemas: " + e.getMessage());  
}
```



# Exceções – Definição de novas exceções

- ▶ A definição de novas exceções pode oferecer mais informação sobre o problema se forem adicionados atributos com essa informação.

```
public class NoMatchingDetailsException extends Exception {  
    private String key;  
    public NoMatchingDetailsException(String message, String key){  
        super(message);  
        this.key = key;  
    }  
    public String getKey(){  
        return key;  
    }  
    public String toString(){  
        return "No details matching '" + key + "' were found.";  
    }  
}
```

Guarda a  
informação sobre a  
chave que originou  
o erro

Vai obter-se a  
key específica  
que originou o  
erro

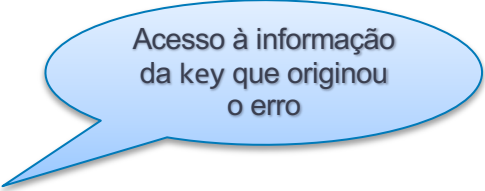
# Exceções – Definição de Novas Exceções

- ▶ Neste exemplo adiciona-se no lançamento da exceção a informação específica da exceção:

```
public ContactDetails getDetails(String key) throws NoMatchingDetailsException {  
    ...  
    throw new NoMatchingDetailsException("Chave com problemas", key);  
    ...  
}
```

- ▶ No caso da exceção ser gerada, temos acesso, através da variável "**e**" (declarada como parâmetro do **catch**), à chave que originou o erro:

```
...  
try {  
    ...  
    contact = agenda.getDetails("Maria");  
    ...  
}  
catch (NoMatchingDetailsException e) {  
    System.out.println("A chave " + e.getKey() + " deu problemas");  
    ...  
}
```



Acesso à informação  
da key que originou  
o erro

# Exceções – Recuperação de Erros

- ▶ Os clientes devem tratar as notificações de erro.
  - ▶ Verifique os valores retornados.
  - ▶ Não ignore as exceções.
- ▶ Inclua código para tentar recuperar do erro.
  - ▶ Frequentemente isso implica um ciclo.

```
// Tenta guardar a agenda telefónica
boolean success = false;
int attempts = 0;
do {
    try {
        addressbook.saveToFile(fileName);
        success = true;
    }
    catch (IOException e) {
        System.out.println("Incapaz de guardar o
            ficheiro " + nomeFicheiro );
        attempts++;
        if (attempts < MAX_TENTATIVAS) {
            fileName = alternativeName;
        }
    }
} while (!success && attempts < MAX_ATTEMPTS);
if (!success) {
    //Relate o problema e desista;
}
```

# Exceções – Mais Comuns

- ▶ **ArithmeticException**  
Indica falhas no processamento aritmético, tal como uma divisão inteira por 0.
- ▶ **ArrayIndexOutOfBoundsException**  
Indica a tentativa de acesso a um elemento de um array fora dos seus limites: ou o índice é negativo ou maior ou igual ao tamanho do array.
- ▶ **IndexOutOfBoundsException**  
Indica a tentativa de usar um índice fora dos limite de uma tabela.
- ▶ **ArrayStoreException**  
Indica a tentativa de armazenamento de um objeto inválido numa tabela.
- ▶ **NegativeArraySizeException**  
Indica a tentativa de criar uma tabela com dimensão negativa.
- ▶ **StringIndexOutOfBoundsException**  
Indica a tentativa de usar um índice numa string fora dos seus limites
- ▶ **NumberFormatException**  
Indica a tentativa de conversão de uma string para um formato numérico, mas que o seu conteúdo não representava um número para aquele formato.
- ▶ **NullPointerException**  
Indica que a instrução tentou usar null onde era necessária uma referência a um objeto
- ▶ **IllegalArgumentException**  
Quando o argumento do método tem um valor impossível
- ▶ **IOException**  
Indica a ocorrência de qualquer tipo de falha em operações de entrada e saída.

# Resumindo

- ▶ A linguagem Java permite o tratamento de situações de exceção de uma forma normalizada através da utilização de 5 palavras chave correspondentes a cláusulas especiais, a saber:

- ▶ **throws**

- ▶ **throw**

- ▶ **try**

- ▶ **catch**

- ▶ **finally**

```
public void method() throws ExcecaoVerificadaHerdaDeException {  
    // [...] implementação do método  
    throw new ExcecaoVerificadaHerdaDeException(...);  
    // [...] mais implementação do método (que não será executada  
    // em caso de ser lançada a exceção)  
}  
// Na chamada do método:  
try{  
    // [...] instruções com chamada ao metodo()  
}  
catch (ExcecaoVerificadaHerdaDeException e) {  
    // Tratamento da exceção "e" do tipo  
    // ExcecaoVerificadaHerdaDeException  
}  
catch (RuntimeException e) {  
    // Tratamento da exceção "e" do tipo RuntimeException  
}  
finally{  
    // Bloco opcional se existir é sempre executado  
}  
}
```

# Resumindo

O mecanismo de exceções é a forma indicada em programação orientada por objetos para lidar com os erros.

- ▶ **throws**
- ▶ **throw**
- ▶ **try**
- ▶ **catch**
- ▶ **finally**

```
public void method() throws ExcecaoVerificadaHerdaDeException {  
    // [...] implementação do método  
    throw new ExcecaoVerificadaHerdaDeException(...);  
    // [...] mais implementação do método (que não será executada  
    // em caso de ser lançada a exceção)  
}  
// Na chamada do método:  
try{  
    // [...] instruções com chamada ao metodo()  
}  
catch (ExcecaoVerificadaHerdaDeException e) {  
    // Tratamento da exceção "e" do tipo  
    // ExcecaoVerificadaHerdaDeException  
}  
catch (RuntimeException e) {  
    // Tratamento da exceção "e" do tipo RuntimeException  
}  
finally{  
    // Bloco opcional se existir é sempre executado  
}  
}
```

# Bibliografia

- ▶ Objects First with Java (6th Edition), David Barnes & Michael Kölling, Pearson Education Limited, 2016
  - ▶ Capítulo 14 (14.1 a 14.6)

