

# Programação Orientada por Objetos

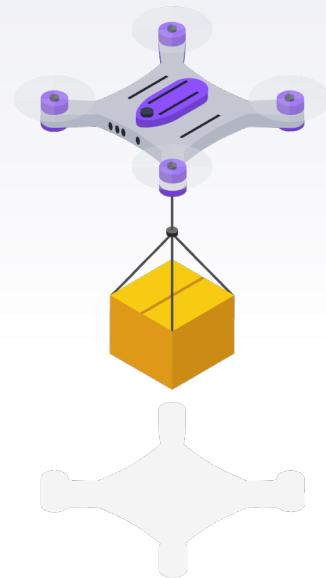
## Entradas e saídas

Prof. Cédric Grueau

Prof. José Sena Pereira

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

2022/2023



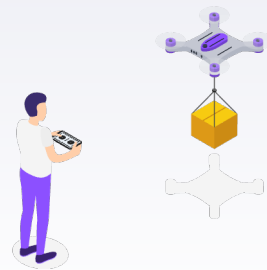
# Sumário

- ▶ Entrada e Saída de Dados
- ▶ Exemplo Prático
- ▶ Serialização



# Entrada e Saída de Dados

- ▶ Entradas e Saídas



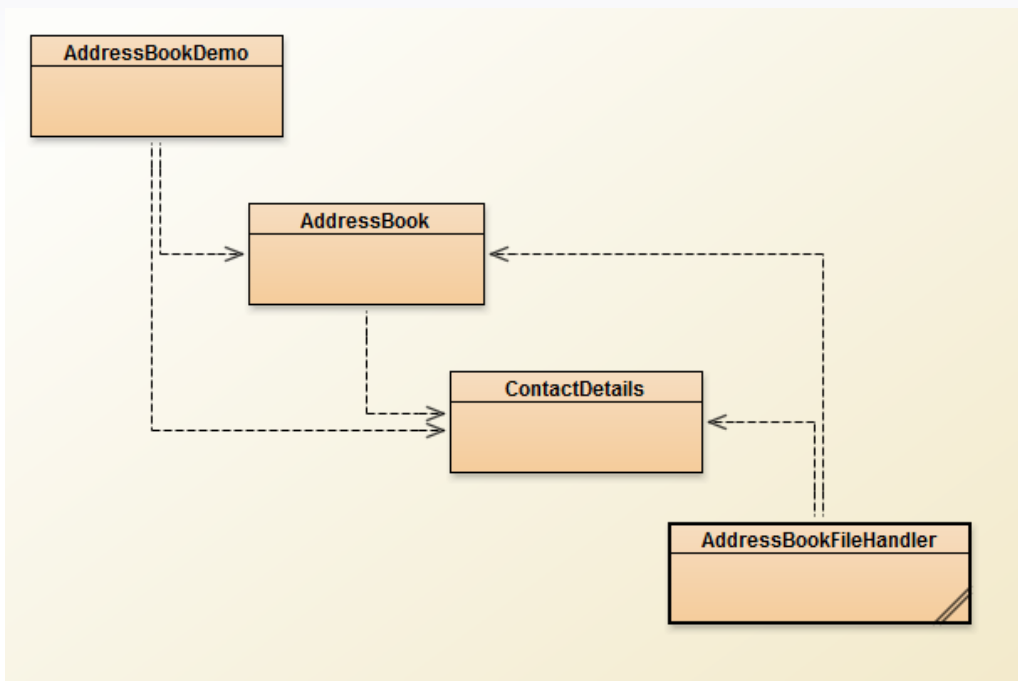
# Exemplo – Address Book

- ▶ 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.
  - ▶ **Guardar os contactos e o resultado das procuras em ficheiro**



# Exemplo – AddressBook

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

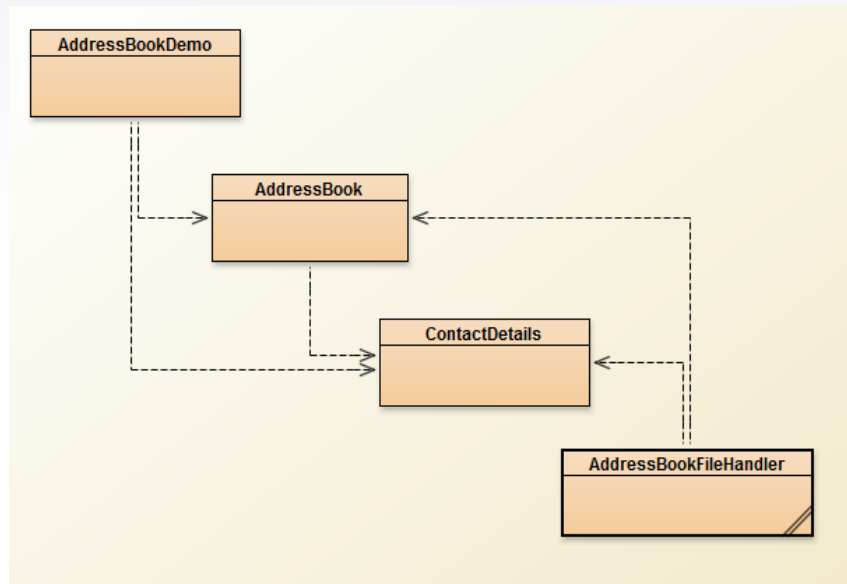


# Exemplo – AddressBook

- ▶ Classes da aplicação

**AddressBook:**

- ▶ **ContactDetails** – Informação do contacto.
- ▶ **AddressBook** – Lista de contactos.
- ▶ **AddressBookDemo** – Cria um livro de contactos com alguns dados para testes.
- ▶ **AddressBookFileHandler** – é responsável pela leitura e escrita em ficheiro da lista de contactos e dos resultados de procura dos contactos.



# Exemplo – AddressBook

## ► Classe **AddressBookDemo**

```
public class AddressBookDemo {  
    private AddressBook book;  
    public AddressBookDemo() {  
        ContactDetails[] sampleDetails = {  
            new ContactDetails("david", "08459 100000", "address 1"),  
            new ContactDetails("michael", "08459 200000", "address 2"),  
            new ContactDetails("john", "08459 300000", "address 3"),  
            new ContactDetails("helen", "08459 400000", "address 4"),  
            new ContactDetails("emma", "08459 500000", "address 5"),  
            new ContactDetails("kate", "08459 600000", "address 6"),  
            new ContactDetails("chris", "08459 700000", "address 7"),  
            new ContactDetails("ruth", "08459 800000", "address 8"),  
        };  
        book = new AddressBook();  
        for(ContactDetails details : sampleDetails) {  
            book.addDetails(details);  
        }  
    }  
    public AddressBook getBook() {  
        return book;  
    }  
}
```

Cria um AddressBook  
com uma lista inicial de  
contactos

# Exemplo – AddressBook

- ▶ Classe **AddressBookFileHandler**

```
public class AddressBookFileHandler {  
    private AddressBook book;  
    private static final String RESULTS_FILE = "results.txt";  
  
    public AddressBookFileHandler(AddressBook book) {  
        this.book = book;  
    }  
    // Continua...
```

Guarda a referência do livro de contactos que irá ser utilizado na escrita e na leitura para ficheiro

Nome do ficheiro que irá guardar o resultado da última procura

Recebe o livro de contactos no construtor

- Antes de vermos a implementação será necessário perceber como funciona a escrita e leitura de ficheiros ...



# Entradas e saídas

- Em Java os **ficheiros** e as **pastas** (ou directórios) são representados pela classe **File**
  - Importa-se como **java.io.File**
  - Em java 7 foram acrescentadas a interface **Path** e as classes **Files** e **Paths** importadas de **java.nio.file** dedicadas igualmente à manipulação de ficheiros.

## Classe File:

### ▶ CONSTRUTORES

**File(String caminho)**

construtor de directórios/ficheiros

**File(String caminho&filename)**

construtor com caminho e nome do ficheiro

### ▶ MÉTODOS

**boolean canRead()**

ficheiro/directório pode ser lido

**boolean canWrite()**

pode-se gravar no ficheiro/directório

**boolean delete()**

apaga ficheiro/directório

**boolean exists()**

verifica se ficheiro/directório existem

**boolean isAbsolute()**

verifica se caminho é absoluto

**boolean isDirectory()**

verifica se objecto é directório

**boolean isFile()**

verifica se objecto é ficheiro

**boolean mkdir()**

cria directório do objecto

**boolean mkdirs()**

cria directórios do caminho

**boolean renameTo(String novo)**

muda nome do ficheiro/directório para novo

# Entradas e saídas

- ▶ Exemplo de utilização da classe **File**

```
import java.io.File;
```

```
class FileDemo {
```

```
    public static void main(String[] args) {
```

```
        String filename = "dados.txt";
```

```
        File file = new File(filename);
```

```
        if (file.exists()) {
```

```
            System.out.println(file.getName() + " existente");
```

```
        }
```

```
        else {
```

```
            System.out.println(file.getName() + " não existente");
```

```
        }
```

```
    }
```

```
}
```

Cria a representação do  
ficheiro:  
O objeto File

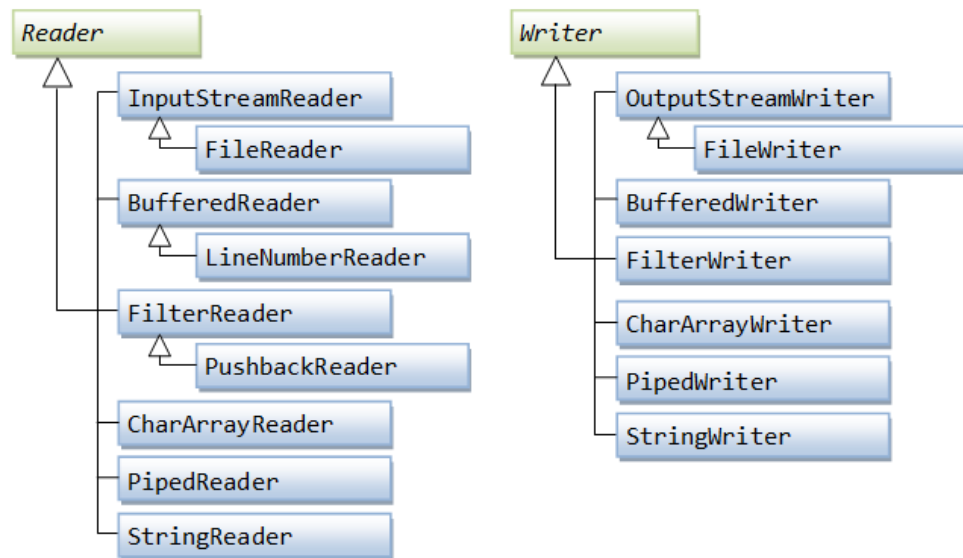
Verifica se o  
ficheiro existe

# Entradas e saídas

- ▶ Para além da representação dos ficheiros o Java inclui classes dedicadas à escrita e à leitura dos ficheiros em disco.
  - ▶ Existem classes independentes para a escrita e para a leitura de ficheiros.
- ▶ De acordo com o tipo de informação que é armazenada, os ficheiros podem ser classificados como **ficheiros de texto** ou **ficheiros binários**
  - ▶ Os **ficheiros de texto** guardam caracteres e podem ser lidos e editados por qualquer aplicação de edição de texto (notepad, word, etc.)
    - ▶ Na realidade a informação é guardada em bytes que representam caracteres de acordo com um determinado standard (ASCII, Unicode, etc.)
    - ▶ Neste caso existe alguma interpretação tanto na leitura como na escrita destes ficheiros de texto
  - ▶ Os **ficheiros binários** guardam a informação em bytes que não é possível interpretar sem a ferramenta adequada. Por exemplo imagens, música, vídeo, etc.
    - ▶ Neste caso não existe qualquer interpretação dos bytes escritos ou lidos.

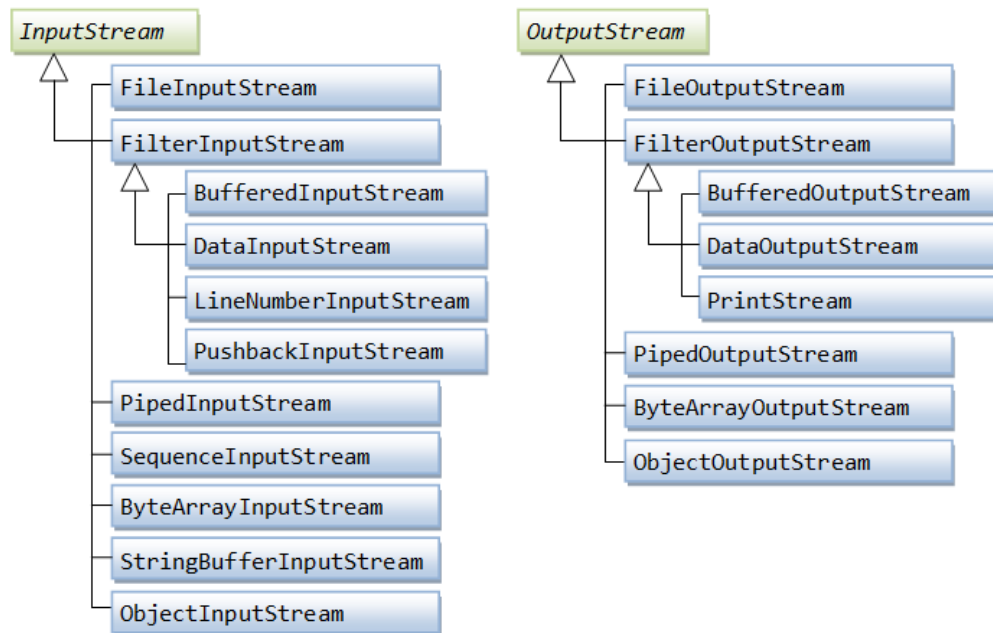
# Entradas e saídas

- Para a **leitura e escrita de ficheiros de texto** existem várias classes que formam uma hierarquia e que derivam respetivamente das classes abstratas **Reader** e **Writer**.



# Entradas e saídas

- Para a **leitura e escrita de ficheiros binários** existem várias classes que formam uma hierarquia e que derivam respetivamente das classes abstratas **InputStream** e **OutputStream**.

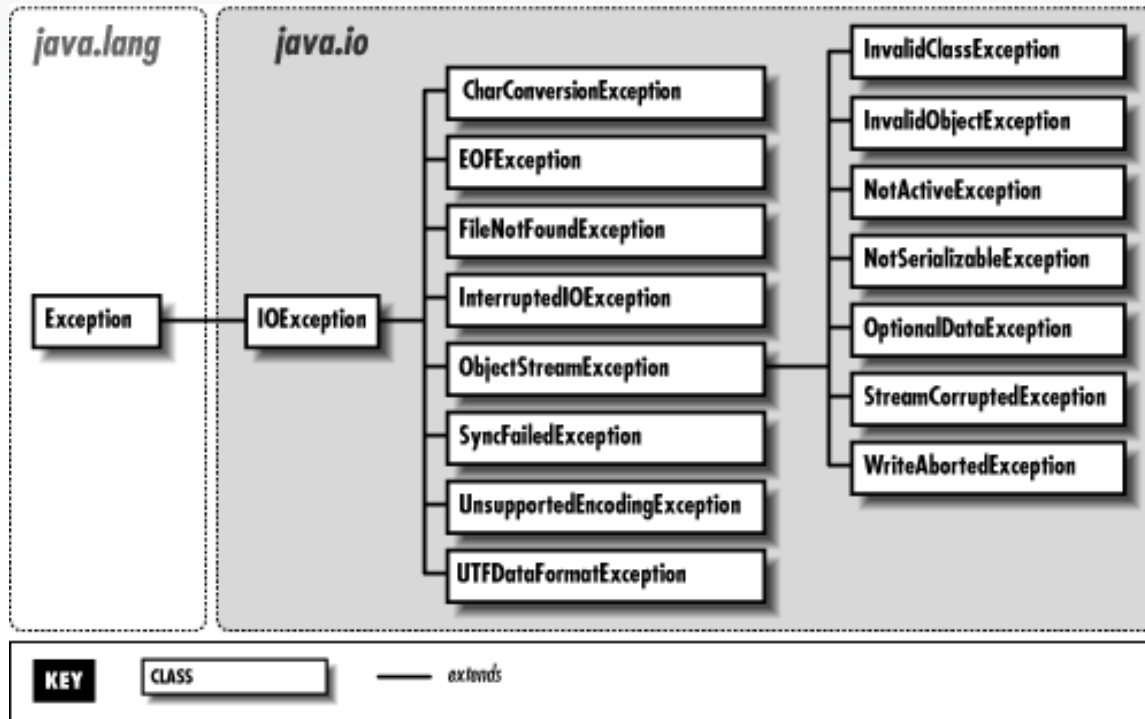


# Entradas e saídas

- ▶ A **leitura e escrita de ficheiros** independentemente de serem de texto ou binários **é feita sequencialmente**.
- ▶ Os **processos** de escrita ou de leitura de ficheiros são sempre feitos em **3 etapas**:
  1. Abrir o ficheiro
  2. Operação de escrita ou de leitura
  3. Fechar o ficheiro.
- ▶ As operações com ficheiros estão também sujeitas a muitos tipos de falhas, por isso a maior parte delas pode levantar **exceções, que são sempre verificadas** (obrigam à utilização de blocos **try-catch**)

# Entradas e saídas

- Hierarquia de **exceções de io** (input/output = entrada/saída) do Java



# Ficheiros de texto

- Exemplo da **escrita** de um ficheiro de Texto usando a classe **FileWriter**.

```
public class TesteFileWriter {  
  
    public static void main(String arg[]) {  
        File ficheiro = new File("textOutput.txt");  
        try { FileWriter fileWriter = new FileWriter(ficheiro);  
            BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);  
            PrintWriter printWriter = new PrintWriter(bufferedWriter);  
            printWriter.println("Saída c/ PrintWriter. Tipos primitivos conv. em strings ");  
            boolean aBoolean = false;  
            int anInt = 1234567;  
            printWriter.println(aBoolean);  
            printWriter.println(anInt);  
            printWriter.flush();  
            printWriter.close();  
        }  
        catch (IOException e) { System.out.println(e.getMessage());  
        }  
    }  
}
```

1. O ficheiro com que se vai trabalhar

2. Criação das classes envolvidas na escrita

3. Escreve-se como se fosse para o ecrã

4. Forçar a escrita (*flush*) e fechar o ficheiro



# Ficheiros de texto

- Exemplo da **leitura** de um ficheiro de Texto usando a classe **FileReader**.

```
public class TesteFileReader {  
    public static void main(String arg[]) {  
        File ficheiro = new File("textOutput.txt");  
        try {  
            FileReader fileReader = new FileReader(ficheiro);  
            BufferedReader bufferedReader = new  
                BufferedReader(fileReader);  
            String line = "";  
            while (line != null) {  
                line = bufferedReader.readLine();  
                System.out.println(line);  
            }  
            bufferedReader.Close();  
        }  
        catch (IOException e) {  
            System.out.println(e.getMessage());  
        }  
    }  
}
```

1. O ficheiro com que se vai trabalhar

2. Criação das classes envolvidas na leitura

3. Lê-se cada linha para uma *string*

4. Fechar o ficheiro

# Ficheiros de texto

- Exemplo da **leitura** de um ficheiro de Texto usando a classe **Scanner**.

```
public class TesteLeituraComScanner {  
    public static void main(String arg[]) {  
        File file = new File("textOutput.txt");  
        try {  
            Scanner scanner = new Scanner(file);  
            // 3. Ler a informação do ficheiro.  
            String primeiraLinha = scanner.nextLine();  
            System.out.println("String lida: " + primeiraLinha);  
            boolean segundaLinha = scanner.nextBoolean();  
            System.out.println("Boolean lido: " + segundaLinha);  
            int terceiraLinha = scanner.nextInt();  
            System.out.println("Inteiro lido: " + terceiraLinha);  
        }  
        catch (InputMismatchException e) {  
            System.out.println("Mismatch exception:" + e);  
        }  
        catch (FileNotFoundException e) {  
            System.out.println("Ficheiro não encontrado!");  
            System.exit(0);  
        }  
    }  
}
```

1. O ficheiro com que se vai trabalhar

2. Criação do objeto Scanner a partir do ficheiro (*file*)

3. Lê-se como foi explicado antes



# Exemplo Prático

- ▶ Entradas e Saídas de Dados



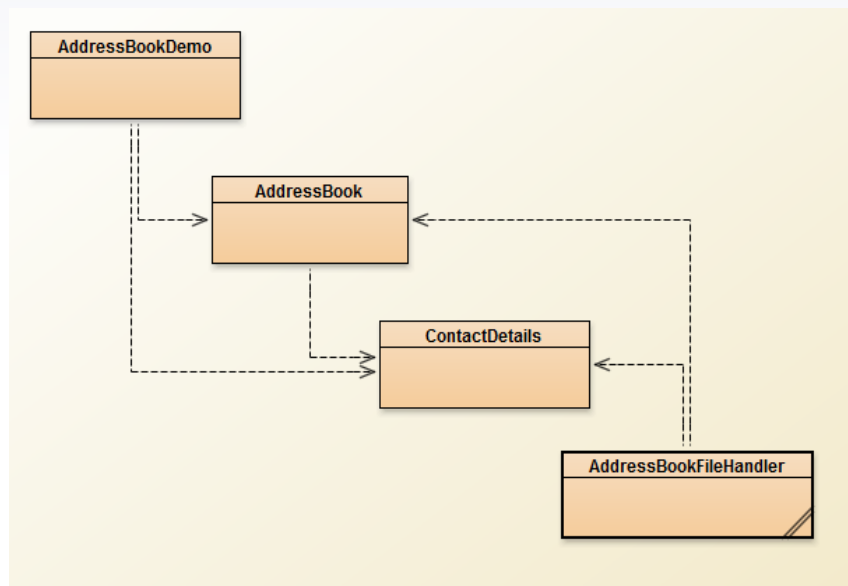
# Exemplo – Address Book

- ▶ 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.
  - ▶ Guardar os contactos e o resultado das procuras em ficheiro



# Exemplo – AddressBook

- ▶ Classes da aplicação **AddressBook**:
  - ▶ **ContactDetails** – Informação do contacto.
  - ▶ **AddressBook** – Lista de contactos.
  - ▶ **AddressBookDemo** – Cria um livro de contactos com alguns dados para testes.
  - ▶ **AddressBookFileHandler** – é responsável pela leitura e escrita em ficheiro da lista de contactos e dos resultados de procura dos contactos.



# Exemplo – AddressBook

- ▶ Classe `AddressBookFileHandler`

```
public class AddressBookFileHandler {  
  
    private AddressBook book;  
    private static final String RESULTS_FILE = "results.txt";  
  
    public AddressBookFileHandler(AddressBook book) {  
        this.book = book;  
    }  
  
    // Continua...
```

Guarda a referência do livro de contactos que irá ser utilizado na escrita e na leitura para ficheiro

Nome do ficheiro que irá guardar o resultado da última procura

Recebe o livro de contactos no construtor

# Exemplo – AddressBook

- ▶ Classe AddressBookFileHandler – métodos `makeAbsoluteFilename` e `getProjectFolder`

```
private File makeAbsoluteFilename(String filename) throws IOException {  
    try {  
        File file = new File(filename);  
        if(!file.isAbsolute()) {  
            file = new File(getProjectFolder(), filename);  
        }  
        return file;  
    }  
    catch(URISyntaxException e) {  
        throw new IOException("Unable to make a valid filename for " + filename);  
    }  
}
```

Cria e devolve a representação do ficheiro no diretório atual do projeto

```
private File getProjectFolder() throws URISyntaxException {  
    String myClassFile = getClass().getName() + ".class";  
    URL url = getClass().getResource(myClassFile);  
    return new File(url.toURI()).getParentFile();  
}
```

Obtém o nome do diretório atual do projeto

# Exemplo – AddressBook

- ▶ Classe **AddressBookFileHandler** - método **saveSearchResults**

```
public void saveSearchResults(String keyPrefix) throws IOException {  
    File resultsFile = makeAbsoluteFilename(RESULTS_FILE);  
    ContactDetails[] results = book.search(keyPrefix);  
    FileWriter writer = new FileWriter(resultsFile);  
    for(ContactDetails details : results) {  
        writer.write(details.toString());  
        writer.write('\n');  
        writer.write('\n');  
    }  
    writer.close();  
}
```

Faz uma procura e  
guarda os resultados  
num array de  
contactos: results

Escreve para disco  
os vários contactos  
do array results

Este método terá de ser  
chamado dentro de um  
bloco try-catch



# Exemplo – AddressBook

## ► Classe

**AddressBookFileHandler**

- método

**showSearchResults**

Lê do ficheiro os resultados da busca e mostra-os no ecrã

```
public void showSearchResults() {
    BufferedReader reader = null;
    try {
        File resultsFile = makeAbsoluteFilename(RESULTS_FILE);
        reader = new BufferedReader(new FileReader(resultsFile));
        System.out.println("Results ...");
        String line; line = reader.readLine();
        while(line != null) {
            System.out.println(line);
            line = reader.readLine();
        }
        System.out.println();
    }
    catch(FileNotFoundException e) {
        System.out.println("Unable to find the file: " + RESULTS_FILE);
    }
    catch(IOException e) {
        System.out.println("Error encountered reading the file: " +
            RESULTS_FILE);
    }
    finally {
        if(reader != null) {
            try { reader.close(); }
        }
        catch(IOException e) {
            System.out.println("Error on closing: " + RESULTS_FILE);
        }
    }
}
```

# Exemplo – AddressBook

- ▶ Classe **AddressBookFileHandler** – método **addEntriesFromFile**

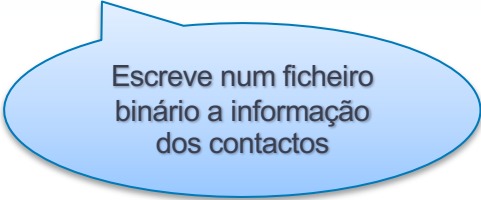
```
public void addEntriesFromFile(String filename) throws IOException {
    URL resource = getClass().getResource(filename);
    if(resource == null) {
        throw new FileNotFoundException(filename);
    }
    filename = resource.getFile();
    BufferedReader reader = new BufferedReader( new FileReader(filename));
    String name;
    name = reader.readLine();
    while(name != null) {
        String phone = reader.readLine();
        String address = reader.readLine();
        // Discard the separating blank line.
        reader.readLine();
        book.addDetails(new ContactDetails(name, phone, address));
        name = reader.readLine();
    }
    reader.close();
}
```

Lê um ficheiro de texto com contactos com a informação do nome, telefone e endereço separada por linhas

# Exemplo – AddressBook

- ▶ Classe **AddressBookFileHandler** – método **saveToFile**

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



Escreve num ficheiro  
binário a informação  
dos contactos

- ▶ Neste caso a escrita para ficheiro no modo binário é feita utilizando a classe **ObjectOutputStream**
  - ▶ O procedimento é semelhante à escrita em modo de texto mas as classes envolvidas são diferentes e também estamos a escrever um objeto completo para o disco.
  - ▶ É necessário igualmente usar este método dentro de um bloco **try-catch**

# Exemplo – AddressBook

- ▶ Classe **AddressBookFileHandler** – método **readFromFile**

```
public AddressBook readFromFile(String sourceFile) throws IOException, ClassNotFoundException {
    URL resource = getClass().getResource(sourceFile);
    if(resource == null) {
        throw new FileNotFoundException(sourceFile);
    }
    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 valid filename for " + sourceFile);
    }
}
```

Lê de um ficheiro  
binário a informação  
dos contactos

Efetua a leitura do ficheiro binário dos contactos escritos anteriormente.

# Serialização

- ▶ Entradas e Saídas de Dados



# Serialização

- ▶ Além das formas tradicionais de escrita e leitura para ficheiro existe ainda uma outra forma chamada **serialização de objetos**
  - ▶ Neste caso é escrito em modo binário para o ficheiro um objeto de uma determinada classe incluindo os objetos que são referenciados nos seus atributos
- ▶ O processo de leitura é chamado **desserialização** de objetos
  - ▶ Agora são restaurados a partir do ficheiro os objetos que anteriormente foram serializados pela mesma ordem em que foram guardados
- ▶ Em Java o algoritmo de serialização de dados garante:
  - ▶ Que quando os dados venham a ser lidos de um ficheiro serializado, todos os objetos com os respetivos atributos serão reconstruídos no estado em que estavam aquando da sua gravação.

# Serialização

- ▶ A serialização é aplicável apenas a instâncias de classes que implementem a interface **Serializable**

```
public class Date implements Serializable {  
    private int year;  
    private int month;  
    private int day;  
    ...  
}
```

# Serialização

- ▶ Ao declarar que uma classe implementa a interface **Serializable**, o compilador gera dois métodos privados para essa classe:
  - ▶ `void writeObject ( ObjectOutputStream out ) throws IOException`
  - ▶ `Object readObject ( ObjectInputStream in ) throws IOException, ClassNotFoundException`
- Um objeto "**ObjectOutputStream**" representa um canal binário que trabalha diretamente sobre um ficheiro e que armazena objetos e valores simples, usando o método `writeObject()` o qual implementa um algoritmo de serialização (**serialize**).
- Um objeto "**ObjectInputStream**" representa um canal especial que trabalha diretamente sobre um ficheiro e lê objetos e valores simples, usando o método `readObject()` o qual implementa um algoritmo de desserialização (**deserialize**).



# Serialização – Gravação em Ficheiro

- ▶ Considerando que
  - ▶ A classe **Persons** inclui um array de objetos da classe **Person**
  - ▶ E que a classe **Person** possui um atributo **yearOfBirth** da classe **Date**
  - ▶ O método abaixo vai gravar num ficheiro binário
    - ▶ um objeto da classe **Persons**,
    - ▶ com todos os elementos do array de objetos (com todos os objetos da classe **Person**)
    - ▶ e para cada objeto da classe **Person**, o nome e a respectiva data de nascimento
    - ▶ Num ficheiro binário serializado a partir do qual será possível reconstituir completamente o objeto da classe **Persons**

```
public static void saveFile(Persons listOfPersons, String filename) {  
    try {  
        ObjectOutputStream oos = new ObjectOutputStream( new FileOutputStream(filename));  
        oos.writeObject(listOfPersons);  
        oos.flush(); oos.close();  
    }  
    catch (IOException e) {  
        System.out.println(e.getMessage());  
    }  
}
```

# Serialização – Leitura de Ficheiro

- ▶ Considerando que
  - ▶ Foi gravado através de uma **ObjectOutputStream** num qualquer ficheiro binário, um objeto da classe **Persons**
  - ▶ É possível lê-lo do ficheiro reconstituindo completamente o seu estado no momento da gravação através do método abaixo:

```
public static Persons readSerializedFile(String filename) {
    Persons listOfPersons;
    try {
        ObjectInputStream ois = new ObjectInputStream( new FileInputStream(filename));
        listOfPersons = (Persons) ois.readObject();
        ois.close();
    }
    catch (IOException e) {
        System.out.println(e.getMessage());
        listOfPersons = new Persons(10);
    }
    catch (ClassNotFoundException e) {
        System.out.println(e.getMessage());
        listOfPersons = new Persons(10);
    }
    return listOfPersons;
}
```

# Serialização – modificador transient

- ▶ Na serialização o Java escreve no ficheiro todos os atributos, não **static**, da classe que implementa a interface **java.io.Serializable**.
- ▶ Podemos indicar que não pretendemos que um atributo seja escrito (eventualmente porque o seu tipo é de uma classe que não implementa a interface **Serializable**) desde que utilizemos o modificador **transient**:
  - ▶ **private transient Color cor; //Color não é Serializable**
- ▶ Se for importante a informação do atributo teremos que implementar as nossas versões dos métodos que fazem a escrita e leitura da informação:
  - ▶ **private void writeObject(java.io.ObjectOutputStream oos) throws IOException**
  - ▶ **private void readObject(ObjectInputStream ois) throws ClassNotFoundException, IOException**
  - ▶ Estes métodos devem chamar, normalmente no seu início, o comportamento por omissão:
    - ▶ **oos.defaultWriteObject();**
    - ▶ **ois.defaultReadObject();**

# Bibliografia

- ▶ Objects First with Java (6th Edition),  
David Barnes & Michael Kölling,  
Pearson Education Limited, 2016
  - ▶ Capítulo 14 (14.9 e 14.10)

