Leitura e Escrita em Ficheiros

Mestrado Integrado em Engenharia Informática FCT UNL
http://ctp.di.fct.unl.pt/miei/ip/

Corpo Docente 2019/2020
António Ravara, Artur Miguel Dias, Bernardo Toninho,
Ema Vieira, Inês Fernandes, Margarida Mamede,
Miguel Monteiro, Rui Nóbrega

Estação Meteorológica

Recorde a Estação Meteorológica

- Objectivo
  - Manipular valores de temperaturas ao longo do tempo.
- Descrição
  - Numa estação meteorológica registam-se valores de temperaturas (valores reais).
- Funcionalidades
  - É sempre possível registar um dado valor de temperatura.
  - É necessário saber sempre o número de temperaturas registadas, poder consultar os valores médio, máximo e mínimo das temperaturas registadas.
  - Quando é criada, não existem temperaturas registadas.

Recorde a Estação Meteorológica

- Interface (classe `WeatherStation`)

```
void sampleTemperature(double temp)
```

Registar a amostra `temp` na estação

```
int numberTemperatures()
```

Consultar o número de temperaturas registadas até ao momento

```
double getMaximum()
```

Consultar a máxima temperatura observada até ao momento

```
pre: numberTemperatures() > 0
```

```
double getMinimum()
```

Consultar a mínima temperatura observada até ao momento

```
pre: numberTemperatures() > 0
```

```
double getAverage()
```

Consultar a média das temperaturas observadas até ao momento

```
pre: numberTemperatures() > 0
```

Estação Meteorológica

- Interacção com o utilizador
  - O utilizador pode registar um conjunto de temperaturas. Para finalizar o registo de temperaturas deve escrever um valor não numérico.
  - No final, caso se tenha feito o registo de pelo menos uma temperatura, deve ser apresentado o valor médio, máximo e mínimo das temperaturas registadas.

Estação Meteorológica – Classe `Main`

```
public class Main {
  /**
   * Programa principal
   */
  public static void main(String[] args){
    WeatherStation ws = readFromConsole();
    writeToConsole(ws);
  }

  /**
   * Métodos auxiliares da Main
   * Cria uma estação, recebe dados, gera relatório.
   */
  private static WeatherStation readFromConsole() {...}
  private static void writeToConsole(WeatherStation ws) {...}
}
```

Normalmente será boa ideia irmos criando métodos auxiliares, em vez de colocar todo o código na classe com o programa principal.

Estação Meteorológica – Classe `Main`

```
public class Main {
  /**
   * Programa principal
   */
  public static void main(String[] args){
     writeToConsole(readFromConsole());
  }

  /**
   * Métodos auxiliares da Main
   * Cria uma estação, recebe dados, gera relatório.
   */

  private static WeatherStation readFromConsole() {...}
  private static void writeToConsole(WeatherStation ws) {...}
}
```

Simplificando um pouco… temos código equivalente

Estação Meteorológica – Classe `Main`

```
/**
 * Cria uma estação meteorológica e lê uma sequência de temperaturas,
 * armazenando-as na estação meteorológica. Finalmente, devolve a
 * estação meteorológica com as temperaturas registadas.
 * @return a estação meteorológica, após lidas as temperaturas.
 */
private static WeatherStation readFromConsole() {
  double temp;
  Scanner in = new Scanner(System.in);
  WeatherStation ws = new WeatherStation();


  System.out.print("Temperatura (caracter nao numerico para fim): ");
  while ( in.hasNextDouble() ) {
    temp=in.nextDouble();
    in.nextLine();
    ws.sampleTemperature(temp);
    System.out.print("Temperatura (caracter nao numerico para fim): ");
  }
  in.close();
  return ws;
}
```

Estação Meteorológica – Classe `Main`

```
/**
 * Cria uma estação meteorológica e lê uma sequência de temperaturas,
 * armazenando-as na estação meteorológica. Finalmente, devolve a
 * estação meteorológica com as temperaturas registadas.
 * @return a estação meteorológica, após lidas as temperaturas.
 */
private static WeatherStation readFromConsole() {
  double temp;
  Scanner in = new Scanner(System.in);
  WeatherStation ws = new WeatherStation();


  System.out.print("Temperatura (caracter nao numerico para fim): ");
  while ( in.hasNextDouble() ) {
    temp=in.nextDouble();
    in.nextLine();
    ws.sampleTemperature(temp);
    System.out.print("Temperatura (caracter nao numerico para fim): ");
  }
  in.close();
  return ws;
}
```

O ciclo vai decorrer enquanto o utilizador inserir valores que são aceites como double.

Cada valor inserido é adicionado à estação!

No final, não esquecer de devolver a estação preenchida!

Estação Meteorológica – Classe `Main`

```
/**
 * Escreve na consola algumas estatísticas sobre a
 * estação meteorológica.
 * @param ws A estação meteorológica.
 */
private static void writeToConsole(WeatherStation ws) {
   System.out.println("Estatisticas");
   System.out.println("Temperatura maxima: " + ws.getMaximum());
   System.out.println("Temperatura minima: " + ws.getMinimum());
   System.out.printf("Media: %.2f\n", ws.getAverage());
}
```

A operação `printf` permite escrever uma `String` formatada.
O primeiro argumento tem a estrutura da string a escrever

- Dentro dessa estrutura, existem "locais" especiais a preencher com dados, os chamados "designadores de formato".
- Neste exemplo, queremos escrever um número real com duas casas decimais. `%.2f` vai ser substituído pelo valor de `ws.getAverage()`, apresentando-o com duas casas decimais.
  - Por exemplo, se ws.getAverage() valer `12.648`, a `String` será:
    "`Media: 12.65`"
- `\n` simboliza a mudança de linha, no final da String.

Estação Meteorológica – Classe `Main`

```
/**
 * Escreve na consola algumas estatísticas sobre a
 * estação meteorológica.
 * @param ws A estação meteorológica.
 */
private static void writeToConsole(WeatherStation ws) {
  System.out.println("Estatisticas");
  System.out.println("Temperatura maxima: " + ws.getMaximum());
  System.out.println("Temperatura minima: " + ws.getMinimum());
  System.out.printf("Media: %.2f\n", ws.getAverage());
}
```

E se não houver temperaturas registadas?
É violada a pré-condição de `getAverage`: `numberTemperatures() > 0`
O cálculo da média faz uma divisão por 0 e o programa aborta!
Para cumprir a pré-condição, devemos usar a operação `numberTemperatures()` da `WeatherStation` num `if` para prevenir contra essa situação.

Estação Meteorológica – Classe `Main`

```
/**
 * Escreve na consola algumas estatísticas sobre a
 * estação meteorológica.
 * @param ws A estação meteorológica.
 */
private static void writeToConsole(WeatherStation ws) {
  if (ws.numberTemperatures()>0) {
    System.out.println("Estatisticas");
    System.out.println("Temperatura maxima: " + ws.getMaximum());
    System.out.println("Temperatura minima: " + ws.getMinimum());
    System.out.printf("Media: %.2f\n", ws.getAverage());
  }
  else
    System.out.println("Nao ha temperaturas registadas!");
}
```

E se os dados estivessem num ficheiro ?

Leitura de ficheiros

Escrita em ficheiros

Informação geral sobre ficheiros

- Na generalidade das linguagens de programação existem operações de **abertura** e **fecho** de um ficheiro**,** para se obter um objeto que representa o ficheiro e através do qual esse ficheiro é processado. Esse objecto funciona como um **intermediário** entre o programa e o sistema operativo.

- Há vários modos de abertura de ficheiro:
  - **Para leitura** – apenas se pode ler e o ficheiro não é modificado, o que até permite que o ficheiro possa ser lido por mais do que um programa ao mesmo tempo. A abertura falha se o ficheiro não for encontrado.
  - **Para escrita** – apenas se pode escrever. Caso o ficheiro não exista, é criado vazio. Caso já exista, o conteúdo anterior perde-se.
  - **Para acrescento** – semelhante à escrita, mas o conteúdo anterior não se perde e a escrita é (geralmente) feita a seguir ao conteúdo anterior.
  - **Para escrita e leitura** – modo mais complexo, que permite a um programa comutar entre escrita e leitura, conforme seja conveniente.

Ficheiros de texto

- São ficheiros cujo conteúdo, se for directamente escrito na consola (ou para uma impressora), será entendido por humanos.
- A organização em linhas (*new line*) é própria dos ficheiros de texto
- Não possuem qualquer formatação adicional
- Para teste, há várias maneiras fáceis de criar ficheiros de texto:
  - com um editor de texto simples, e.g. Bloco de Notas, Notepad++
  - como saída dos seus programas
- Por exemplo, no Eclipse podemos criar ficheiros de texto seleccionando o projecto no Package Explorer e fazendo:
  - File->New->Untitled text file
- Tipicamente, os ficheiros de texto são gravados com a extensão txt
- NUNCA use **processadores de texto** (e.g. Word) para criar ficheiros de texto simples.
  - Introduzem informação (que pode não ser visível) para além do texto, para dar suporte aos detalhes de formatação dos documentos.

Informação geral sobre ficheiros

- Na generalidade das linguagens de programação, a leitura e escrita de/para a consola usa a metáfora de **ficheiros de texto** – mesmo quando a linguagem fornece instruções próprias para os usar.
  - Leitura da consola é feita como a de um ficheiro de texto aberto para leitura
  - Escrita para a consola é feita como a de um ficheiro de texto aberto para escrita
- Em IP, o processamento de ficheiros foca-se nos ficheiros de texto. São considerados apenas os modos de **leitura** e de **escrita** (o conteúdo anterior perde-se e o ficheiro é criado vazio, caso não exista).
- A linguagem Java usa classes diferentes para representar ficheiros abertos para leitura (e.g., **`FileReader`**) e escrita (e.g., **`PrintWriter`**). Serão essas as classes usadas – em conjunto com a classe **`Scanner`** nos casos de leitura.
- Em linguagens OO como Java, a criação do objecto que representa o ficheiro (i.e., a chamada ao construtor) efectua a abertura do ficheiro.

Ler ficheiros de texto

- `import java.io.*;` deve ser colocado no início.
- A forma mais simples de ler texto envolve o uso dum `Scanner`
- Para ler de um ficheiro do disco, cria-se um objecto da classe `FileReader`. Depois, usa-se o `FileReader` para construir um objecto `Scanner`

```
FileReader reader = new FileReader("input.txt");
Scanner in = new Scanner(reader);
...
in.close();
```

- Utilizam-se os já nossos conhecidos métodos de `Scanner` para ler dados:
  - `next`, `nextLine`, `nextInt`, e `nextDouble`

– Simplificando:

```
Scanner in = new Scanner(new FileReader("input.txt"));
...
in.close();
```

Escrever ficheiros de texto

- Para escrever num ficheiro, constrói-se um objecto da classe `PrintWriter`

```
PrintWriter out = new PrintWriter("output.txt");
```

- Se o ficheiro já existe, o seu conteúdo é apagado antes das novas acções de escrita. Se o ficheiro não existir, é criado um ficheiro vazio
- Usam-se `print`, `println` e `printf` para escrever num objecto `PrintWriter`:

```
out.println(29.95);
out.println(new Rectangle(5, 10, 15, 25));
out.println("Hello, World!");
```

- Terminado o processamento do ficheiro, este deve ser fechado: `out.close();` Senão, existe o risco do output não ser todo escrito no ficheiro

Programa principal - `Main`

```
public class Main {

  private static WeatherStation readFrom(Scanner in) {...}
  private static void writeToFile(WeatherStation ws) {...}

  /**
   * Programa principal
   */
  public static void main(String[] args) {
    Scanner in = new Scanner(new FileReader("sample.txt"));
    writeToFile(readFrom(in));
    in.close();
  }
}
```

Encontramos aqui, pela primeira vez, o conceito de excepção. **Uma excepção representa uma situação fora do normal que é processada à parte, no ponto do programa mais conveniente para o programador.**

O mecanismo de tratamento de excepções será devidamente estudado em POO. Em IP referimos apenas o mínimo necessário para conseguir manipular ficheiros.

E se o ficheiro `sample.txt` não existir?
Ooops! Isto assim não funciona. Nem sequer conseguimos compilar. Obtemos a seguinte mensagem de erro:
Unhandled exception type `FileNotFoundException`
Isto acontece porque o construtor de `FileReader` pode lançar esta excepção. O nosso método tem de escolher entre duas opções: (1) tratar logo a excepção; ou (2) delegar o seu tratamento no método chamador (chama-se a isto **propagar a excepção**).

Programa principal - `Main`

```
public class Main {

  private static WeatherStation readFrom(Scanner in) {...}
  private static void writeToFile(WeatherStation ws) {...}

  /**
   * Programa principal
   */
  public static void main(String[] args) throws FileNotFoundException {
    Scanner in = new Scanner(new FileReader("sample.txt"));
    writeToFile(readFrom(in));
    in.close();
  }

}
```

A declaração `throws FileNotFoundException`faz o método `main` delegar a responsabilidade do tratamento da excepção no chamador. O chamador do método `main` é um método especial de biblioteca preparado para lidar com todas as excepções. Esse tratamento consiste em abortar a execução do programa e em escrever uma mensagem de erro com informação sobre a excepção.

Leitura de ficheiros de texto

```
/*
 * Cria uma estação meteorológica e lê uma sequência de temperaturas,
 * de um ficheiro chamado sample.txt armazenando-as na estação
 * meteorológica. Finalmente, devolve a estação meteorológica com as
 * temperaturas registadas.
 * @return a estação meteorológica, após lidas as temperaturas.
 *
 */
private static WeatherStation readFrom(Scanner in) {
  WeatherStation ws = new WeatherStation();
  while ( in.hasNextDouble() ) {
    double temp = in.nextDouble();
    in.nextLine();
    ws.sampleTemperature(temp);
  }
  return ws;
}
```

O nosso método `readFrom()` recebe um `Scanner` como parâmetro. Tanto funciona com um fluxo de dados vindo de um ficheiro de texto, como vindo do `System.in`.

Escrita de ficheiros de texto

```
/**
 * Escreve na consola algumas estatísticas sobre a estação meteorológica.
 * @param ws A estação meteorológica.
 * @throws FileNotFoundException
 */
private static void writeToFile(WeatherStation ws)  throws FileNotFoundException
{
  PrintWriter out = new PrintWriter("statistics.txt");
  if (ws.numberTemperatures()>0) {
    out.println("Estatisticas");
    out.println("Temperatura maxima: "  + ws.getMaximum());
    out.println("Temperatura minima: "  + ws.getMinimum());
    out.printf ("Media: %.2f\n", ws.getAverage());
  } else {
    out.println("Nao ha temperaturas registadas." );
  }
  out.close();
}
```

A declaração `throws FileNotFoundException` sinaliza que o método propaga a excepção `FileNotFoundException` porque o programador decidiu que este não era o melhor sítio para processar a excepção. O processamento é delegado no método chamador.

Escrita de ficheiros de texto

```
/**
 * Escreve na consola algumas estatísticas sobre a estação meteorológica.
 * @param ws A estação meteorológica.
 * @throws FileNotFoundException
 */
private static void writeToFile(WeatherStation ws)  throws FileNotFoundException
{
  PrintWriter out = new PrintWriter("statistics.txt");
  if (ws.numberTemperatures()>0) {
    out.println("Estatisticas");
    out.println("Temperatura maxima: "  + ws.getMaximum());
    out.println("Temperatura mínima: "  + ws.getMinimum());
    out.printf ("Media: %.2f\n", ws.getAverage());
  } else {
    out.println("Nao ha temperaturas registadas." );
  }
  out.close();
}
```

Uma vez que `writeToFile()` propaga a excepção `FileNotFoundException`, os métodos que o invocam têm de poder lidar com a excepção de algum modo.

Programa principal - Main

```
public class Main {
  /**
   * @throws FileNotFoundException
   */
  public static void main(String[] args) throws FileNotFoundException {
    testFiles();
  }
  /**
   * Método auxiliar de teste da WeatherStation com ficheiros
   * Cria uma estação, recebe dados, gera relatório.
   * @throws FileNotFoundException
   */
  private static void testFiles() throws FileNotFoundException {
    Scanner in = new Scanner(new FileReader("sample.txt"));
    WeatherStation ws = readFrom(in);
    writeToFile(ws);
  }
  private static void writeToFile(WeatherStation ws)
                                  throws FileNotFoundException {...}
}
```

O que fazer com a excepção?

```
public class Main {
   /**
    * @throws FileNotFoundException
    */
   public static void main(String[] args) throws FileNotFoundException {
      testFiles();
   }
   …
}
```

O ideal seria o método testFiles() conseguir abrir e processar o ficheiro. Se conseguir, excelente.
Caso contrário, a excepção gerada por testFiles() é recebida pelo método main e propagada para o chamador que, neste caso, aborta a execução e escreve uma mensagem de erro.

Devemos “apanhar a excepção”

```
public class Main {

   ...
   public static void main(String[] args) {
      try {
          testFiles();
      } catch (FileNotFoundException e) {
        System.out.println("Problemas nos ficheiros");
    }
   }
}
```

Se o acesso ao ficheiro falhar,o código executado Será este

Podemos *proteger* o nosso código com a instrução `try...catch`!
Nesse caso, o método `main` decidiu tratar a excepção em vez de a propagar.
A clausula `throws` desaparece do `main`, mas mantém-se nos métodos privados de leitura e escrita!

A construção `try... catch`

- A construção `try... catch` permite apanhar as excepções. A sua sintaxe (simplificada) é:

```
try {
   //bloco de instruções que pode lançar a excepção
} catch (ExceptionType name) {
   //instruções a executar caso ocorra a excepção
}
```

- Por exemplo:

```
try {
   //bloco de instruções que pode lançar a excepção
} catch (FileNotFoundException e) {
   System.out.println("Ficheiro nao acessivel");
}
```

Leitura e escrita de ficheiros

- Em resumo:
  - A leitura e escrita em ficheiros de texto é feita de modo muito semelhante ao que já se fazia na consola
    - Na leitura, inicializamos o `Scanner` com um objecto do tipo `FileReader`, em vez de `System.in`
    - Na escrita, usamos um objecto do tipo `PrintWriter`, em vez de `System.out`
    - Como o ficheiro de leitura, ou de escrita, podem não estar acessíveis, temos de prever essa possibilidade com a excepção `FileNotFoundException`
  - Além da utilização de ficheiros, vimos também uma primeira abordagem ao mecanismo de excepções do Java
    - Por agora, o tratamento dado a excepções é delegar (através da declaração `throws`) a sua resolução na operação que chama a operação onde a excepção ocorre e propagar essa delegação até ao programa principal
    - Mais à frente estudará detalhadamente o mecanismo de tratamento de excepções e formas mais sofisticadas de lidar com elas.

Agenda de Contactos

Agenda de contactos

- Objectivo
  - Manipular uma agenda de contactos.
- Interacção com o utilizador

  ...
  - A informação dos contactos existentes no início do programa encontra-se num ficheiro de texto.
  - No final da execução do programa deverá ser escrito em ficheiro de texto toda a informação referentes aos contactos, para posterior uso.

Escrita da agenda em ficheiro

- O que é necessário definir na classe Main?
- Constante com o nome do ficheiro que vai guardar os contactos

```
private static final String FILENAME = "contacts.txt";
```

- Método de escrita do ficheiro, que vai chamar os métodos de iteração do `ContactBook`

```
private static void writeContactBook (ContactBook
cb, String filename) throws FileNotFoundException
```

- Chamada ao método de escrita do ficheiro, antes do final do método `main`
- O método main necessita de referência a `throws FileNotFoundException` na assinatura

Adicionar à classe `Main`

```
private static void writeContactBook(ContactBook cBook,
        String filename) throws FileNotFoundException {
    PrintWriter pw = new PrintWriter(filename);
    pw.println(cBook.getNumberOfContacts());
    ContactIterator it = cBook.iterator();
    while (it.hasNext()){
        Contact c = it.next();
        pw.print(c.getPhone()+" ");
        pw.print(c.getEmail()+" ");
        pw.println(c.getName());

    }
    pw.close();
}
```

É o objeto na variável `pw` que nos permite escrever no ficheiro de dados. Temos de ter o cuidado de **abrir o ficheiro**, **escrever nele**, e **no final fechar o ficheiro**.

Adicionar à classe `Main`

```
private static void writeContactBook(ContactBook cBook,
        String filename) throws FileNotFoundException {
    PrintWriter pw = new PrintWriter(filename);
    pw.println(cBook.getNumberOfContacts());
    ContactIterator it = cBook.iterator();
    while (it.hasNext()){
        Contact c = it.next();
        pw.print(c.getPhone()+" ");
        pw.print(c.getEmail()+" ");
        pw.println(c.getName());
    }
    pw.close();
}
```

Atenção:
Em cada iteração do ciclo apenas invocamos o `next()` uma vez, caso contrário avançaríamos mais que um contacto.

Adicionar à classe `Main`

```
public class Main {
  public static final String FILENAME = "contacts.txt";
  ...
  public static void main(String[] args) throws FileNotFoundException {
        Scanner in = new Scanner(System.in);
    ContactBook cBook = new ContactBook();
    String comm = getCommand(in);
    while (!comm.equals(QUIT)){
            if (comm.equals(ADD_CONTACT))
                addContact(in,cBook);
            else
                System.out.println(WRONG_COMM);
            comm = getCommand(in);
    }
    writeContactBook(cBook,FILENAME);
    System.out.println(BYE);
        in.close();
  }
  ...
}
```

Ler a agenda de ficheiro

- Em `ContactBook:`
  - Não é preciso desenvolver mais nada.
- Na Classe `main:`
  - Vamos usar a constante `FILENAME`;
  - Vamos usar método estático `addContact();`
    - Que por sua vez usa o `addContact()` do `ContactBook`;
  - Vamos desenvolver o método
    - `private static void readContactBook( ContactBook cBook, String filename) throws FileNotFoundException`
    - Este método vai chamar o método `addContact()`

Ler a agenda de ficheiro

```
private static void readContactBook(ContactBook cBook,
    String filename) throws FileNotFoundException {

        System.out.println("Reading Contacts File...");
        FileReader fich = new FileReader(filename);
        Scanner fin = new Scanner(fich);
        int cont = fin.nextInt();
        fin.nextLine();
        for(int i=0; i<cont; i++){
            addContact(fin,cBook);
        }
     fin.close();
}
```

O `Scanner` usa um `FileReader`

Leitura e escrita da agenda em ficheiro

```
public class Main {
  private static final String FILENAME = ”contacts.txt";
  ...
  public static void main(String[] args) throws FileNotFoundException{
    Scanner in = new Scanner(System.in);
    ContactBook cBook = new ContactBook();
    readContactBook(cBook, FILENAME);
    String comm = getCommand(in);
    while (!comm.equals(QUIT)){
        if (comm.equals(ADD_CONTACT))
            addContact(in,cBook);
        else
            System.out.println(WRONG_COMM);
        comm = getCommand(in);
    }
    writeContactBook(cBook,FILENAME);
    System.out.println(BYE);
    in.close();
  }
}
```

Leitura e escrita da agenda em ficheiro

```
public class Main {
  private static final String FILENAME = "contacts.txt";
  ...
  public static void main(String[] args) throws FileNotFoundException{
    Scanner in = new Scanner(System.in);
    ContactBook cBook = new ContactBook();
    readContactBook(cBook, FILENAME);
    String comm = getCommand(in);
    while (!comm.equals(QUIT)){
        if (comm.equals(ADD_CONTACT))
            addContact(in,cBook);
        else
            System.out.println(WRONG_COMM);
        comm = getCommand(in);
    }
    writeContactBook(cBook,FILENAME);
    System.out.println(BYE);
    in.close();
  }
}
```

Ao correr, o ideal é tentar usar o ficheiro. Se conseguir, excelente. Caso contrário, falhar com o erro do sistema de execução.

Devemos “apanhar a excepção”

```
public class Main {
  public static final String FILENAME = ”contacts.txt";
  ...
  public static void main(String[] args) {
    try {
        Scanner in = new Scanner(System.in);
        ContactBook cBook = new ContactBook();
        readContactBook(cBook,FILENAME);
        String comm = getCommand(in);
        while (!comm.equals(QUIT)){
            if (comm.equals(ADD_CONTACT))
                addContact(in,cBook);
            else System.out.println(WRONG_COMM);
            comm = getCommand(in);
        }
        writeContactBook(cBook,FILENAME);
        System.out.println(BYE);
        in.close();
    } catch (FileNotFoundException e) { ... }
  }
}
```

Podemos proteger o nosso código com a instrução `try...catch`! Nesse caso, o método `main` já não tem de propagar a excepção. A clausula `throws` desaparece do `main`, mas mantém-se em `readContactBook` e `writeContactBook`!

Se o acesso ao ficheiro falhar,o código executado Será este

Podemos apanhar as excepções de modo mais preciso

```
public class Main {
  public static final String FILENAME = "contacts.txt";
  ...
  public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        ContactBook cBook = new ContactBook();
        try {
              readContactBook(cBook,FILENAME);
        } catch (FileNotFoundException e) { ... }
        String comm = getCommand(in);

        while (!comm.equals(QUIT)){
           if (comm.equals(ADD_CONTACT))
             addContact(in,cBook);
           else System.out.println(WRONG_COMM);
           comm = getCommand(in);
        }
        try {
          writeContactBook(cBook,FILENAME);
        } catch (FileNotFoundException e) { ... }
        System.out.println(BYE);
        in.close();
  }
}
```

Podemos proteger o nosso código com a instrução `try...catch`! Nesta segunda implementação, estamos apenas a proteger as chamadas potencialmente problemáticas. Assim, se o `readContactBook` levantar uma excepção, o programa não carrega contactos, mas continua a executar. Nesse caso, o método `main` já não tem de propagar a excepção. A clausula `throws` desaparece do `main`, mas mantém-se em `readContactBook` e `writeContactBook`!

Estrutura da aplicação

Interface com o utilizador

```
public class Main { ...
  public static void main(...) {
    ContactBook cBook;
    ...
  }
  private static void addContact(...){...}
  private static void importContacts(...){...}
  private static void deleteContact(...){...}
  private static String getPhone(...){...}
  private static String getEmail(...){...}
  private static void setPhone(...){...}
  private static void setEmail(...){...}
  private static void listAllContacts (...){...}
  private static void readContactBook(...)
                throws FileNotFoundException{...}
  private static void writeContactBook(...)
                throws FileNotFoundException{...}
  ...
}
```

Classes do domínio

```
public class ContactBook {
  public boolean hasContact(...){...}
  public int getNumberOfContacts(){...}
  public void addContact(...){...}
  public void importContacts(...){...}
  public String getPhone(...){...}
  public String getEmail(...){...}
  public void deleteContact(...){...}
  public void setPhone(...){...}
  public void setEmail(...){...}
  public ContactIterator iterator() {...}
}
```

```
public class ContactIterator {
  public boolean hasNext(){...}
  public Contact next(){...}
}
```

```
public class Contact {
  public Contact(...){...}
  public String getName() {...}
  public String getPhone() {...}
  public String getEmail() {...}
  public void setPhone(String phone) {...}
  public void setEmail(String email) {...}
}
```

Recapitulando, a construção `try... catch`

- A construção `try... catch` permite apanhar as excepções. A sua sintaxe (simplificada) é:

```
try {
   //bloco de instruções que pode lançar a excepção
} catch (ExceptionType name) {
   //instruções a executar caso ocorra a excepção
}
```

- Por exemplo:

```
try {
   //bloco de instruções que pode lançar a excepção
} catch (FileNotFoundException e) {
   System.out.println("Ficheiro não acessível");
}
```

Por agora, é suficiente...

- A forma de tratar excepções descrita aqui é suficiente no contexto de IP
- Em Programação Orientada pelos Objectos, a segunda cadeira de programação do MIEI, estudará em detalhe o mecanismo de excepções do Java