Agenda de Contactos

- Desenvolver em Java de um programa principal que permite ao utilizador manipular uma agenda de contactos.
  - A interface de utilização do programa é feita através de comandos (*interpretador* de comandos).
  - 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.

Interpretador de comandos

- O programa principal deve dar ao utilizador a possibilidade de execução de diversas operações numa agenda, o número de vezes que o utilizador pretender.
- Interface de utilização do programa: comandos

```
> AC (adiciona um contacto)
> RC (remove um contacto)
> GP (consulta o telefone de um contacto)
> GE (consulta o e-mail de um contacto)
> SP (actualiza o telefone de um dado contacto)
> SE (actualiza o e-mail de um dado contacto)
> LC (lista todos os contactos existentes na agenda)
> Q  (sair)
```

Definição dos comandos

- Adicionar novo contacto:
  - adiciona o novo contacto à agenda, se não existir um contacto com o mesmo nome.
    - `AC`
    - `Joana Dias`
    - `99999999`
    - `Joana@fct.unl.pt`
    - `Contact added.`
    - `AC`
    - `Joana Dias`
    - `99999999`
    - `Joana@fct.unl.pt`
    - `Contact already exists.`

Definição dos comandos

- Remover um contacto:
  - Remove o contacto da agenda, se existir um contacto com o nome dado.
    - `RC`
    - `Joana Dias`
    - `Contact removed.`
    - `RC`
    - `Joana Dias`
    - `Cannot remove contact.`

Definição dos comandos

- Consultar telefone:
  - Consulta o telefone do contacto, se existir um contacto com o nome dado.
    - `GP`
    - `Joana Dias`
    - `99999999`
    - `GP`
    - `Joana Anos`
    - `Contact does not exist.`

Definição dos comandos

- Consultar *e-mail*:
  - Consulta o *e-mail* do contacto, se existir um contacto com o nome dado.
    - `GE`
    - `Joana Dias`
    - `Joana@fct.unl.pt`
    - `GE`
    - `Joana Anos`
    - `Contact does not exist.`

Definição dos comandos

- Actualiza o telefone:
  - Actualiza o telefone do contacto, se existir um contacto com o nome dado.
    - `SP`
    - `Joana Dias`
    - `253253253`
    - `Contact updated.`
    - `SP`
    - `Joana Anos`
    - `253253253`
    - `Contact does not exist.`
    - `GP`
    - `Joana Dias`
    - `253253253`

Definição dos comandos

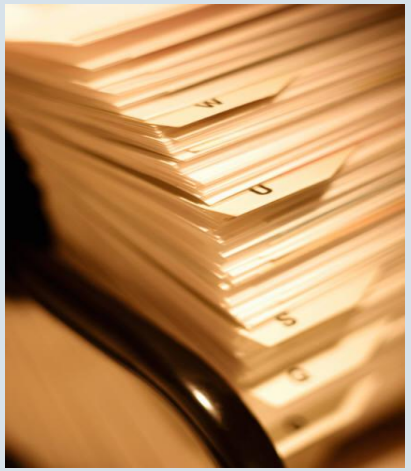
- Actualiza o *e-mail*:
  - Actualiza o *e-mail* do contacto, se existir um contacto com o nome dado.
    - `SE`
    - `Joana Dias`
    - `JoanaEu@tu.ele.pt`
    - `Contact updated.`
    - `SE`
    - `Joana Anos`
    - `JoanaEu@tu.ele.pt`
    - `Contact does not exist.`
    - `GE`
    - `Joana Dias`
    - `JoanaEu@tu.ele.pt`

Definição dos comandos

- Listagem de contactos:
  - Actualiza o *e-mail* do contacto, se existir um contacto com o nome dado.
    - `LC`
    - `Joana Dias;JoanaEu@tu.ele.pt;253253253`
    - `Joana Anos;Joana@tu.ele.pt;99999999`
    - `LC`
    - `Contact book empty.`

Agenda de Contactos

- Desenvolver em Java uma classe `ContactBook` cujos objectos são agendas de contactos.
  - Cada objecto desta classe contêm um conjunto de contactos. Logo é necessário também definir a classe `Contact` em que os objectos são contactos.



Contacto

- Cada contacto caracteriza-se por um nome, um telefone e um *e-mail*.
- Para criar objectos da classe `Contact` são necessários três argumentos: uma `String` que representa o nome do contacto; um `int` que representa o número de telefone; e uma `String` que representa o *e-mail*.

Contacto

- Operações (interface de Contact):

```
public String getName()
```

Devolve o nome do contacto

```
public int getPhone()
```

Devolve o telefone do contacto

```
public String getEmail()
```

Devolve o e-mail do contacto

```
public void setPhone(int phone)
```

Altera o telefone do contacto para `phone`

```
public void setEmail(String email)
```

Altera o telefone do contacto para `e-mail`

```
public String toString()
```

Devolve a descrição do contacto (nome, telefone e e-mail)

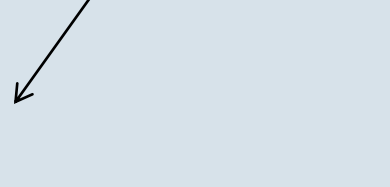
Programando a classe `Contact`

```
public class Contact {
   private String name;
   private int phone;
   private String email;

   public Contact(String name, int phone, String email) {
      this.name = name;
      this.phone= phone;
      this.email= email;
    }
    public String getName() { … }
    public int getPhone() { … }
    public String getEmail() { … }
    public void setPhone(int phone) { … }
    public void setEmail(String email) { … }
     …
}
```

Programando a classe `Contact`

```
public class Contact{
   private String name;
   private int phone;
   private String email;

   public Contact(String name, int phone, String email) {
      this.name = name;
      this.phone= phone;
      this.email= email;
    }
    public String getName() { … }
    public int getPhone() { … }
    public String getEmail() { … }
    public void setPhone(int phone) { … }
    public void setEmail(String email) { … }
     …
}
```

Nota: o conjunto de operações suportadas não permite alterar o nome de um contacto



Agenda de Contactos

- Uma agenda de contactos é um conjunto de vários contactos.
  - Cada contacto é identificado pelo seu nome. Assume-se que não existem contactos com nomes iguais.

Agenda de Contactos

- Associado a cada agenda existe um conjunto de contactos.
- **Operações** que são realizadas na agenda:
  - Consulta do telefone de um contacto, dado o nome;
  - Consulta do *e-mail* de um contacto, dado o nome;
  - Inserção de um novo contacto, dado o nome, número de telefone e *e-mail*;
  - Remoção de um contacto, dado o nome;
  - Alteração do *e-mail* de um contacto, dado o nome;
  - Alteração do telefone de um contacto, dado o nome;

Agenda de Contactos

- Operações reconhecidas (interface de `ContactBook`):

`public int getPhone(String name)`
Método que devolve o telefone de um contacto associado a um dado nome

`public String getEmail(String name)`
Método que devolve o e-mail de um contacto associado a um dado nome

`public void addContact(String name, int phone, String email)`
Método que insere um novo contacto

`public void deleteContact(String name)`
Método que remove um contacto

`public void setPhone(String name, int phone)`
Método que altera o telefone de um contacto

`public void setEmail(String name, String email)`
Método que altera o e-mail de um contacto

`public boolean hasContact(String name)`
Método que indica se um dado contacto existe

`public int numberOfContacts()`
Método que indica o número de contactos na agenda

Agenda de Contactos

- Operações reconhecidas (interface de `ContactBook`):
  - Para percorrer a agenda vamos ter várias operações para iterar sobre todos os contactos. Assumimos que durante o percurso pela agenda não são feitas inserções nem remoções de contactos.

`public void` `initializeIterator`**`()`**

Método que inicia a iteração sobre os contactos da agenda

`public` `boolean hasNext`**`()`**

Método que indica se existe um contacto seguinte.

Quando se chega ao fim da travessia, devolve false, caso contrário devolve true

`public` `Contact next`**`()`**

Método que devolve a informação referente ao contacto corrente

Agenda de Contactos

- Uma agenda de contactos é um conjunto de vários contactos…
- Necessitamos poder guardar `N` contactos, por isso vamos precisar de:
  - *Vector* de contactos com uma dada *capacidade máxima*;
  - Número que indica *quantos* contactos existem no vector;
  - *Índice* do contacto corrente quando estamos a iterar *todos* os contactos da agenda.

Programando a Agenda de Contactos

```
public class ContactBook{
   public static final int MAX_CONTACTS = 50;

   private int counter;
   private Contact[] contacts;
   private int currentContact;

   // Construtores
   public ContactBook() {
     counter = 0;
     contacts= new Contact[MAX_CONTACTS];
     int currentContact = -1;
   }
   ...
}
```

Número de contactos existentes no vector

Vector de objectos `Contact`

Posição do contacto corrente

Inicialização do vector `contacts`, com a dimensão `MAX_CONTACTS`.

Agenda de Contactos

- Todas as operações da agenda (excepto as referentes à iteração dos contactos da agenda e ao número de contactos) necessitam de procurar um contacto dado o seu nome.
  - Adicionar um contacto: necessita saber se já existe um contacto contacto com esse nome;
  - Remover um dado contacto, indicando o nome: necessita procurar o contacto pelo nome;
  - Consultar ou alterar o *e-mail* de um dado contacto, indicando o nome: necessita procurar o contacto pelo nome;
  - Consultar ou alterar o telefone de um dado contacto, indicando o nome: necessita procurar o contacto pelo nome;
  - Consultar se um dado contacto existe, indicando o nome: necessita procurar o contacto pelo nome.
- Logo deve existir uma operação *auxiliar* acessível a apenas os métodos da classe. Logo, deve ter visibilidade *privada*:

```
private int searchIndex(String name)
```

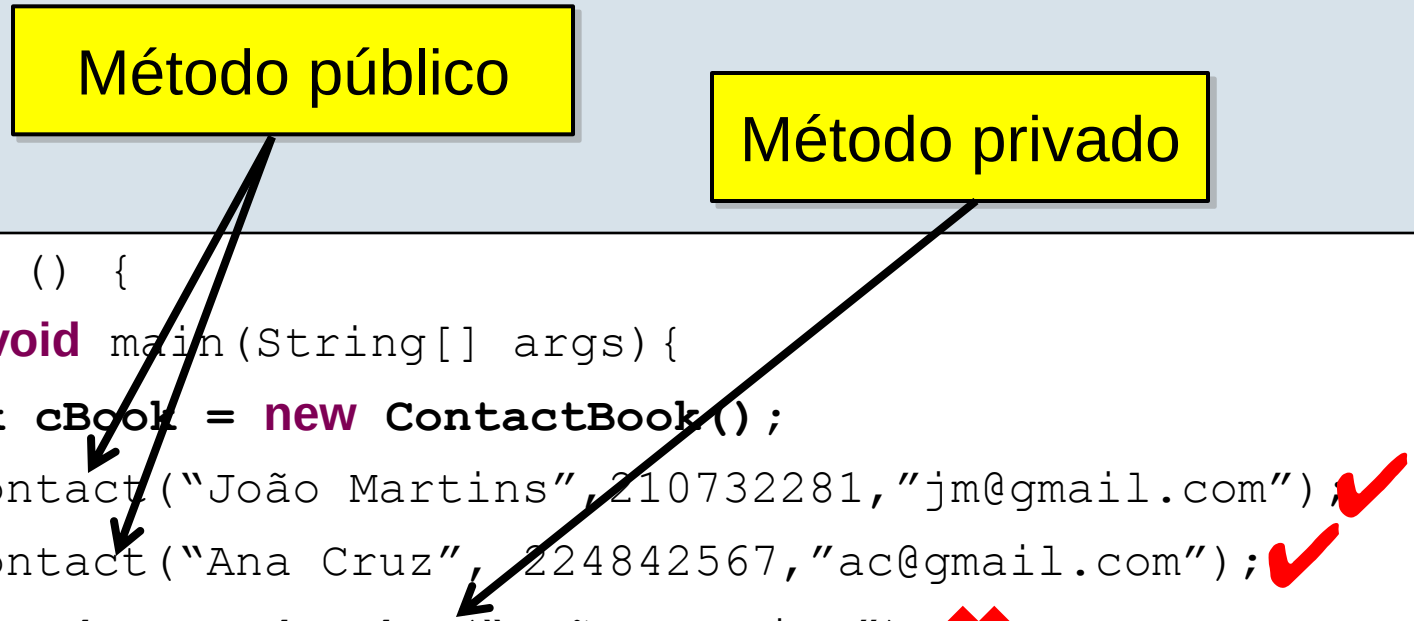
Método que devolve a posição no vector `contacts` do contacto associado a `name` (devolve -1 caso o nome não exista na lista de contactos)

Agenda de Contactos

- Operação privada:

```
private int searchIndex(String name)
```

Método que devolve a posição no vector `contacts` do contacto associado a `name` (devolve -1 caso o nome não exista na lista de contactos)



Programando a Agenda de Contactos

```
public class ContactBook {
    ...
    private Contact[] contacts; // vector de contactos
    private int counter; // número de contactos no vector
    private int currentContact; // posição do contacto corrente num varrimento
    ...
    private int searchIndex(String name){
      int i = 0; // percurso pelos elementos 0..counter
      int result = -1;
      boolean find = false; // indicador de existência
      while((i < counter) && (!find))
          if( contacts[i].getName().equals(name) == 0)
                find = true;
          else i++;
      if (find) result = i;
      return result;
    } ...
```

Objecto `Contact` guardado na posição i do vector

Programando a Agenda de Contactos

```
public class ContactBook {
    ...
    private Contact[] contacts; // vector de contactos
    private int counter; // número de contactos no vector
    private int currentContact; // posição do contacto corrente num varrimento
    ...
    private int searchIndex(String name){
      int i = 0; // percurso pelos elementos 0..counter
      int result = -1;
      boolean find = false; // indicador de existência
      while((i < counter) && (!find))
          if( contacts[i].getName().equals(name) )
                find = true;
          else i++;
      if (find) result = i;
      return result;
    } ...
```

Consultar a classe `String` para obter informação sobre o método `equals`

Agenda de Contactos

Igualdade de objectos:

- Dado uma classe `Type`, a operação em `Type` para verificar a igualdade de 2 objectos `Type` será a seguinte:

  `public boolean` `equals(`*`Type`* `anotherObject)`

  Devolve true se o objecto corrente (i.e., `this`) é "igual" ao objecto do mesmo tipo passado como argumento e, false caso contrário.

- Ao programarmos `equals`, decidimos qual é o critério de comparação que pretendemos.

Programando a Agenda de Contactos

```
public class ContactBook {
    ...
    private Contact[] contacts; // vector de contactos
    private int counter; // número de contactos no vector
    private int currentContact; // posição do contacto corrente num varrimento
    ...
    public void addContact(String name, int phone, String email){
      if (searchIndex(name) == -1) { // novo contacto
          if (counter == contacts.length) { // vector cheio
                Contact tmp[] = new Contact[2*contacts.length];
                int i=0;
                while (i < counter) { tmp[i] = contacts[i]; i++; }
                contacts = tmp;
          }
          contacts[counter] = new Contact(name,phone,email);
          counter++;
      }
}
```

Programando a Agenda de Contactos

```
public class ContactBook {
    ...
    private Contact[] contacts; // vector de contactos
    private int counter; // número de contactos no vector
    private int currentContact; // posição do contacto corrente
    ...
    public void addContact(String name, int phone, String email){
      if (searchIndex(name) == -1) { // novo contacto
          if (counter == contacts.length) { // vector cheio
                Contact tmp[] = new Contact[2*contacts.length];
                int i=0;
                while (i < counter) { tmp[i] = contacts[i]; i++; }
                contacts = tmp;
          }
          contacts[counter] = new Contact(name,phone,email);
          counter++;
    }
```

Atenção: cada elemento de um vector de objectos tem de ser criado explicitamente.

Programando a Agenda de Contactos

```
public class ContactBook {
    ...
    private Contact[] contacts; // vector de contactos
    private int counter; // número de contactos no vector
    private int currentContact; // posição do contacto corrente
    ...
    public void deleteContact(String name){
     int index = searchIndex(name);
     if (index != -1) { // contacto existente
          int i=index;
          while (i < counter-1) {
               contacts[i] = contacts[i+1];
               i++;
          }
          counter--;
    }
...
```

Programando a Agenda de Contactos

```
public class ContactBook {
    ...
    private Contact[] contacts; // vector de contactos
    private int counter; // número de contactos no vector
    private int currentContact; // posição do contacto corrente
    ...
    public void setEmail(String name, String email){
      int pos = searchIndex(name);
      if (pos != -1)  // contacto existe
          contacts[pos].setEmail(email);
    }
    /* Pre: o contacto existe */
    public String getEmail(String name){
      return contacts[searchIndex(name)].getEmail();
    }
```

Agenda de Contactos

- Operações reconhecidas (interface de ContactBook):
  - Para percorrer a informação referente a todos os contactos vamos ter várias operações para iterar entre todos os contactos. Assumimos que durante o percurso pela agenda não são feitas inserções nem remoções de contactos.

```
public void initializeIterator()
```
Método que inicia a iteração sobre os contactos da agenda

```
public boolean hasNext()
```
Método que indica se existe um contacto seguinte.

Quando se chega ao fim da travessia, devolve false, caso contrário devolve true

```
public Contact next()
```
Método que devolve a informação referente ao contacto corrente

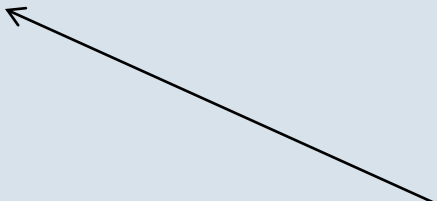
Classe ContactBook

```
public class ContactBook{
    ...
    private Contact[] contacts; // vector de contactos
    private int counter; // número de contactos no vector
    private int currentContact; // posição do contacto corrente
    ...
    public void initializeIterator(){
        if (counter > 0)
            currentContact = 0;
        else currentContact = -1
    }
    ...
}
```

Coloca o contacto corrente da iteração como sendo o contacto que está na posição 0

Classe ContactBook

```
public class ContactBook{
    ...
    private Contact[] contacts; // vector de contactos
    private int counter; // número de contactos no vector
    private int currentContact; // posição do contacto corrente
    ...
    public boolean hasNext(){
        return ((currentContact >= 0 ) &&
                (currentContact < counter));
    }
    ...
}
```



Só existem mais contactos para percorrer se a posição do contacto corrente for menor que o número de elementos e maior ou igual a 0

Classe ContactBook

```
public class ContactBook{
    ...
    private Contact[] contacts; // vector de contactos
    private int counter; // número de contactos no vector
    private int currentContact; // posição do contacto corrente
    ...
    public Contact next(){
        Contact contact = null;
        if (this.hasNext())
            contact = contacts[currentContact++];
        return contact;
    }
}
```

Devolve o contacto e prepara o contacto corrente para a próxima iteração (next)

Classe ContactBook

```
public class ContactBook{
    ...
    private Contact[] contacts; // vector de contactos
    private int counter; // número de contactos no vector
    private int currentContact; // posição do contacto corrente
    ...
    public Contact next(){
        Contact contact = null;
        if (this.hasNext())
            contact = contacts[currentContact++];
        return contact;
    }
}
```

null é a referência nula, o “zero” das referências a objectos.

Classe Main

- Exemplo de uso do iterador de ContactBook:

```
public class Main {
  public static void main(String[] args) {
        ContactBook cBook = new ContactBook();

        //…

        //Listar todos os contactos na consola
        cBook.initializeIterator();
        while( cBook.hasNext() ) {
            System.out.println(cBook.next().toString());

}
```

Iteradores

- Relação entre o controlo dum ciclo **for** e os iteradores:

```
   for(int i = 0; i < counter; i++ ) {
        System.out.println(contacts[i].toString());
   }

public void initializeIterator(){
   if (counter != 0) currentContact = 0;
   else currentContact = -1
}
public boolean hasNext(){
   return ((currentContact>=0)&&(currentContact < counter));
}
public Contact next(){
   Contact contact= null;
   if ( hasNext() ) contact= accounts[currentContact++];
   return contact;
}
```

Iteradores

- Relação entre o controlo dum ciclo **for** e os iteradores:

Inicialização do i corresponde à inicialização do iterador

```
for(int i = 0; i < counter; i++ ) {
      System.out.println(contacts[i].toString());
}

public void initializeIterator(){
   if (counter != 0) currentContact = 0;
   else currentContact = -1
}
public boolean hasNext(){
   return ((currentContact>=0)&&(currentContact < counter));
}
public Contact next(){
   Contact contact= null;
   if ( hasNext() ) contact= accounts[currentContact++];
   return contact;
}
```

Iteradores

- Relação entre o controlo dum ciclo **for** e os iteradores:

Condição de controlo do ciclo corresponde ao método hasNext()

```
for(int i = 0; i < counter; i++ ) {
      System.out.println(contacts[i].toString());
}

public void initializeIterator(){
   if (counter != 0) currentContact = 0;
   else currentContact = -1
}
public boolean hasNext(){
   return ((currentContact>=0)&&(currentContact < counter));
}
public Contact next(){
   Contact contact= null;
   if ( hasNext() ) contact= accounts[currentContact++];
   return contact;
}
```

Iteradores

- Relação entre o controlo dum ciclo **for** e os iteradores:

Incremento do índice i corresponde ao método hasNext()

```
for(int i = 0; i < counter; i++ ) {
      System.out.println(contacts[i].toString());
}


public void initializeIterator(){
   if (counter != 0) currentContact = 0;
   else currentContact = -1
}
public boolean hasNext(){
   return ((currentContact>=0)&&(currentContact < counter));
}
public Contact next(){
   Contact contact= null;
   if ( hasNext() ) contact= accounts currentContact++ ;
   return contact;
}
```

Interpretador de comandos classe Main

- O interpretador faz parte da interacção com o utilizador. Logo, deve constar da classe Main. Para cada comando deve existir um método *estático* que lê e/ou apresenta resultados:

```
public class Main {
        public static void main(String[] args) { ... }

        private static void addContact( ... ) { ... }
        private static void deleteContact( ... ) { ... }
        private static void getPhone( ... ) { ... }
        private static void getEmail( ... ) { ... }
        private static void setPhone( ... ) { ... }
        private static void setEmail( ... ) { ... }
        private static void listAllContacts( ... ) { ... }
}
```

Classe Main: método addContact()

```
>AC
Joana Dias
213334455
joana@ggg.tt.pt
Contact Added.
```

```
private static void addContact(Scanner in, ContactBook cBook) {
  String name = "";
  String email = "";
  int phone=0;
  name = in.nextLine();
  phone = in.nextInt();
  in.nextLine();
  email = in.nextLine();
  if (!cBook.hasContact(name)){
        cBook.addContact(name, phone, email);
        System.out.println(CONTACTADDED);
     }
  else System.out.println(CONTACTEXIST);
}
```

Classe Main: constantes

```
public class Main {
...
   //Constantes que definem os comandos
   public static final String ADD_CONTACT    = "AC";
   public static final String REMOVE_CONTACT = "RC";
   public static final String GET_CONTACT    = "GC";
   public static final String GET_PHONE      = "GP";
   public static final String GET_EMAIL      = "GE";
   public static final String SET_PHONE      = "SP";
   public static final String SET_EMAIL      = "SE";
   public static final String LIST_CONTACTS  = "LC";
   public static final String QUIT           = "Q";

   //Constantes que definem as mensagens de erro
   public static final String WRONG_COMM = "Invalid Command.";
   public static final String CONTACT_EXISTS =
     "Contact already exists.”;
   public static final String CANNOT_REMOVE = "Cannot remove contact.";
   public static final String NAME_NOT_EXIST =
     "Contact does not exist.";
   public static final String CONTACT_ADDED = "Contact added.";

}
```

Classe Main: método getCommand()

```
public class Main {
...
    //Método para recolher comando do Scanner
   private static String getCommand(Scanner in) {
      String input = "";

      System.out.print("> ");
      input = in.nextLine().toUpperCase();
      return input;

    }

...

}
```

Método main: Interpretador de comandos

Interpretador com apenas os comandos "AC" e "Q"

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

  Scanner in = new Scanner(System.in);
  ContactBook cBook = new ContactBook();
  String comm = getCommand();

  while (!comm.equals(QUIT)){
       if (comm.equals(ADD_CONTACT))
             addContact(in,cBook);
       else System.out.println(WRONG_COMM);
       comm = getCommand();
  }
  System.out.println("Goodbye!");
}
```

Escrita da agenda em ficheiro

- O que é necessário definir na classe Main?
- Constante com o nome do ficheiro que vai guardar os contactos
  - `public static final String FILE = "contacts.txt";`
- Método de escrita do ficheiro, que vai chamar os métodos de iteração do ContactBook
  - `public static void writeContactBook(ContactBook cb, String file)throws Exception`
- Chamada ao método de escrita do ficheiro, antes do final do método main
- O método main necessita de referência a `"throws Exception"` na assinatura

Adicionar à classe Main

```
public static void writeContactBook(ContactBook cBook,
          String file) throws Exception {
     PrintWriter pw = new PrintWriter(file);
     Contact c = null;
     pw.println(cBook.getNumberOfCOntacts());
     cBook.initIteration();
     while (cBook.hasNextContact()){
          c = cBook.nextContact();
          pw.println(c.getName());
          pw.println(c.getPhone());
          pw.println(c.getEmail());
     }
     pw.close();
   }
```

Adicionar à classe Main

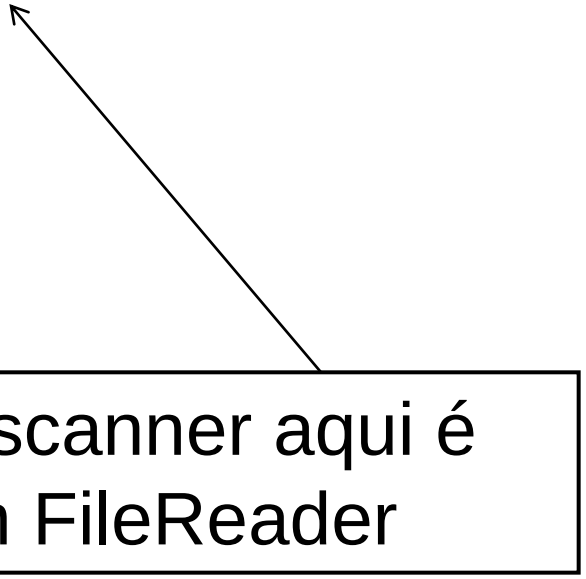
```
public Main {
  public static final String FILE = ”contacts.txt";
  ...
  public static void main(String[] args) throws Exception{
        Scanner in = new Scanner(System.in);
        ContactBook cBook = new ContactBook();
        String comm = getCommand();

        while (!comm.equals(QUIT)){
              if (comm.equals(ADD_CONTACT))
                    addContact(in,cBook);
              else System.out.println(WRONG_COMM);
              comm = getCommand();
        }
        writeContactBook(cBook,FILE);
        System.out.println("Goodbye!");
  }
  ...
}
```

Ler a agenda de ficheiro

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

Ler a agenda de ficheiro

```
public static void readContactBook(ContactBook cBook,
                    String file) throws Exception {

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



Ler a agenda de ficheiro

```
public Main {
  public static final String FILE = ”contacts.txt";
  ...
  public static void main(String[] args) throws Exception{
       Scanner in = new Scanner(System.in);
       ContactBook cBook = new ContactBook();
       readContactBook(cBook,FILE);
       String comm = getCommand();

       while (!comm.equals(QUIT)){
              if (comm.equals(ADD_CONTACT))
                     addContact(in,cBook);
              else System.out.println(WRONG_COMM);
              comm = getCommand();
       }
       writeContactBook(cBook,FILE);
       System.out.println("Goodbye!");
  }
  ...
}
```