Vectores de objectos

Material didáctico elaborado pelas diferentes equipas de
Introdução à Programação

Luís Caires (Responsável), Armanda Rodrigues, António Ravara, Carla Ferreira, Fernanda Barbosa, Fernando Birra, Jácome Cunha, João Araújo, Miguel Goulão, Miguel Pessoa Monteiro, e Sofia Cavaco.

Mestrado Integrado em Engenharia Informática FCT UNL

Agenda de Contactos

- **Objectivo**
  - Manipular uma agenda de contactos.
- **Descrição e Funcionalidades**
  - Cada contacto na agenda caracteriza-se por um nome, um telefone e um *e-mail*. Na agenda, o contacto é identificado pelo seu nome.
  - Deve ser sempre possível consultar e/ou alterar o telefone e o email de um dado contacto, indicando o nome do contacto.
  - Pode-se registar novos contactos, e remover contactos existentes.
  - É sempre possível listar a informação de todos os contactos.
- **Interacção com o utilizador**
  - A interface de utilização do programa é feita através de comandos (interpretador de comandos).

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.
> AC
> Joana Horas
> 91999999
> Joana@gmail.com
Contact added.
```

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.
> AC
> Joana Dias
> 99999999
> Joana@fct.unl.pt
Contact added.
```

Definição dos comandos

- Consultar telefone:
  - Consulta o telefone do contacto, se existir um contacto com o nome dado.

```
> GP
> Joana Dias
99999999
> GP
> Joana Meses
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 Meses
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 Meses
> 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 Meses
> JoanaEu@tu.ele.pt
Contact does not exist.
> GE
> Joana Dias
JoanaEu@tu.ele.pt
```

Definição dos comandos

- Listagem de contactos:
  - Lista os contactos.

```
> LC
Joana Dias;JoanaEu@tu.ele.pt;253253253
Joana Horas;Joana@gmail.com;91999999
```

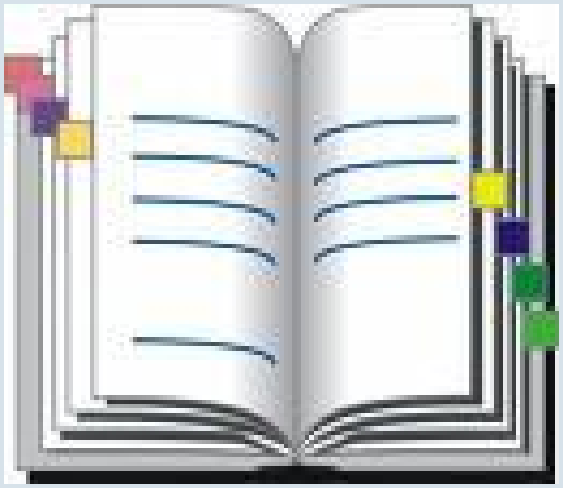
  - Lista os contactos – o que acontece se a lista estiver vazia?

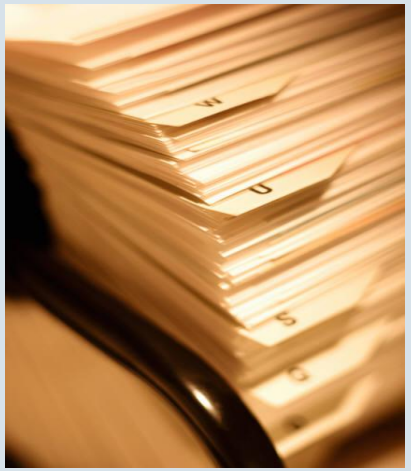
```
> LC
Contact book empty.
```

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.





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 int getPhone(...){...}
   private static String getEmail(...){...}
   private static void setPhone(...){...}
   private static void setEmail(...){...}
   private static void listAllContacts (...){...}
}
```

Classes do domínio

```
public class ContactBook {
   public boolean hasContact(...){...}
   public int getNumberOfContacts(){...}
   public void addContact(...){...}
   public void importContacts(...){...}
   public int getPhone(...){...}
   public String getEmail(...){...}
   public void deleteContact(...){...}
   public void setPhone(...){...}
   public void setEmail(...){...}
   public void initializeIterator(...){...}
   public boolean hasNext(){...}
   public Contact next(){...}
}
```

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

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 endereço de ***e-mail***.
- Deve existir uma funcionalidade que indica se dois contactos são iguais, isto é se têm o mesmo nome.

Contacto

- Operações (interface da classe `Contact`):

  `String getName()`

  Devolve o nome do contacto

  `int getPhone()`

  Devolve o telefone do contacto

  `String getEmail()`

  Devolve o e-mail do contacto

  `void setPhone(int phone)`

  Altera o telefone do contacto para `phone`

  `void setEmail(String email)`

  Altera o email do contacto para `e-mail`

  `boolean equals(Contact otherContact)`

  Retorna `true`, caso o nome do contacto corrente seja igual ao nome do contacto dado

  pre: `otherContact != null`

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) { … }
    public boolean equals(Contact otherContact) { … }
}
```

Programando a classe `Contact`

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

    public Contact(String name, int phone, String email) { … }

    public String getName() { … }
    public int getPhone() { … }
    public String getEmail() { … }
    public void setPhone(int phone) { … }
    public void setEmail(String email) { … }
    public boolean equals(Contact otherContact) { … }
}
```

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

Programando a classe `Contact`

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

    public Contact(String name, int phone, String email) { … }

    public String getName() { … }
    public int getPhone() { … }
    public String getEmail() { … }
    public void setPhone(int phone) { … }
    public void setEmail(String email) { … }
    public boolean equals(Contact otherContact) { … }
}
```

Discutiremos o equals mais tarde...

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;
  - 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 boolean hasContact(String name)
```

Indica se existe na agenda um contacto com o nome dado

```
public int getNumberOfContacts()
```

Indica o número de contactos na agenda

```
public void addContact(Contact contact)
```

Adiciona um novo contacto na agenda

Pre: `!hasContact(contact.getName())`

```
public void deleteContact(String name)
```

Remove o contacto da agenda, cujo nome é o dado

Pre: `hasContact(name)`

Agenda de Contactos

- Operações reconhecidas (interface de `ContactBook`):

```
...
public int getPhone(String name)
```

Consulta o telefone do contacto associado a um dado nome

Pre: `hasContact(name)`

```
public String getEmail(String name)
```

Consulta o e-mail do contacto associado a um dado nome

Pre: `hasContact(name)`

```
public void setPhone(String name, int phone)
```

Altera o telefone do contacto com o nome dado

Pre: `hasContact(name)`

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

Altera o e-mail do contacto com o nome dado

Pre: `hasContact(name)`

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;

Programando a Agenda de Contactos

```
public class ContactBook{
    private static final int MAX_CONTACTS = 50;
    private static final int GROWTH_FACTOR = 2;
    private int counter;
    private Contact[] contacts;


    // Construtores
    public ContactBook() {
        counter = 0;
        contacts = new Contact[MAX_CONTACTS];

    }
    ...
}
```

Número de contactos existentes no vector

Vector de objectos `Contact`

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

Agenda de Contactos

- Até agora, todas as operações da agenda necessitam de procurar um contacto dado o seu nome.
  - Adicionar um contacto: necessita saber **se já existe um contacto com esse nome**;
  - Remover um dado contacto, indicando o nome: necessita e **procurar o contacto pelo nome**;
  - Consultar ou alterar o *e-mail* de um dado contacto, indicando o nome: necessita de **procurar o contacto pelo nome**;
  - Consultar ou alterar o telefone de um dado contacto, indicando o nome: necessita de **procurar o contacto pelo nome**;
  - Consultar se um dado contacto existe, indicando o nome: necessita de **procurar o contacto pelo nome**.

- 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)

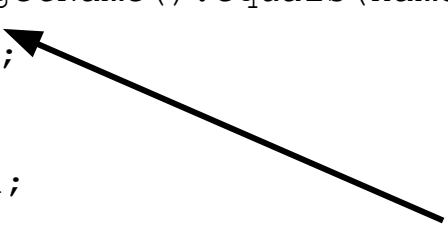
Chamadas a método público: ok

```
public class Main () {
public static void main(String[] args){
    ContactBook cb = new ContactBook();
    cb.addContact(new Contact("João", 2538899, "j@gmail.pt"));
    cb.addContact(new Contact("Ana", 21248775, "a@gmail.pt"));
    int res = cb.searchIndex("João Martins");
  }
}
```

Chamada a método privado: erro

Programando a Agenda de Contactos

```
public class ContactBook {
    ...
    private Contact[] contacts; // vector de contactos
    private int counter; // numero de contactos no vector
    ...
    private int searchIndex(String name){
        int i = 0; // percurso pelos elementos 0..counter-1
        int result = -1; // por enquanto, ainda nao encontramos o elemento
        boolean found = false; // indicador de existencia
        while((i < counter) && (!found))
            if( contacts[i].getName().equals(name))
                found = true;
            else i++;
        if (found) result = i;
        return result;
    }
    ...
}
```



Instância da classe `Contact` guardada na posição `i` do vector

Programando a Agenda de Contactos

```
public class ContactBook {
    ...
    private Contact[] contacts; // vector de contactos
    private int counter; // numero de contactos no vector
    ...
    private int searchIndex(String name){
        int i = 0; // percurso pelos elementos 0..counter-1
        int result = -1; // por enquanto, ainda nao encontramos o elemento
        boolean found = false; // indicador de existencia
        while((i < counter) && (!found))
            if( contacts[i].getName().equals(name))
                found = true;
            else i++;
        if (found) result = i;
        return result;
    }
    ...
}
```

Invocamos o método `getName()` do objecto de tipo `Contact` guardado na posição `i` do vector.
O resultado da avaliação de **`contacts[i].getName()`** é, portanto, uma `String` com o nome do contacto guardado na posição `i` do vector `contacts`.

Programando a Agenda de Contactos

```
public class ContactBook {
    ...
    private Contact[] contacts; // vector de contactos
    private int counter; // numero de contactos no vector
    ...
    private int searchIndex(String name){
        int i = 0; // percurso pelos elementos 0..counter-1
        int result = -1; // por enquanto, ainda nao encontramos o elemento
        boolean found = false; // indicador de existencia
        while((i < counter) && (!found))
            if( contacts[i].getName().equals(name) )
                found = true;
            else i++;
        if (found) result = i;
        return result;
    }
    ...
}
```

Finalmente, comparamos o nome do contacto guardado na posição `i` do vector `contacts` com name. Para comparar Strings, usamos o método `equals`, que está definido na classe `String`. Consulte a documentação da classe `String` para obter informação sobre o método `equals`.

Programando a Agenda de Contactos

```
public class ContactBook {
    ...
    private Contact[] contacts; // vector de contactos
    private int counter; // numero de contactos no vector
    ...
    private int searchIndex(String name){
        int i = 0; // percurso pelos elementos 0..counter-1
        int result = -1; // por enquanto, ainda nao encontramos o elemento

        while((i < counter) && (result == -1)) {
            if( contacts[i].getName().equals(name) )
                result = i;
            i++;
        }
        return result;
    }
    ...
}
```

Outra versão

Programando a Agenda de Contactos

```
public class ContactBook {
    ...
    private Contact[] contacts; // vector de contactos
    private int counter; // numero de contactos no vector
    ...



    public boolean hasContact(String name) {
        return ( searchIndex(name) >= 0 );
    }









    public int getNumberOfContacts() {
        return counter;
    }
    ...
}
```

Programando a Agenda de Contactos

```
public class ContactBook {
    ...
    private Contact[] contacts; // vector de contactos
    private int counter; // numero de contactos no vector
    ...



    public boolean hasContact(String name) {
        return ( searchIndex(name) >= 0 );
    }

    public int getNumberOfContacts() {
        return counter;
    }
    ...
}
```

Para determinar se um elemento pertence ou não à colecção, basta procurar o seu índice. Repare que, se o elemento não existir, `searchIndex(name)` devolve `-1`. A operação `hasContact` esconde da classe cliente a forma como os contactos são, de facto, guardados.

Programando a Agenda de Contactos

```
public class ContactBook {
    ...
    private Contact[] contacts; // vector de contactos
    private int counter; // numero de contactos no vector
    ...



    // Pre: !hasContact(other.getName())
    public void addContact(Contact contact) {
        if (counter == contacts.length)
            resize();
        contacts[counter++] = other;
    }
}
```

Programando a Agenda de Contactos

```
public class ContactBook {
    ...
    private Contact[] contacts; // vector de contactos
    private int counter; // numero de contactos no vector
    ...
    private boolean isFull() {
        return counter == contacts.length;
    }
    // Pre: !hasContact(other.getName())
    public void addContact(Contact contact){
        if (isFull())
            resize();
        contacts[counter++] = other;
    }
}
```

Podemos melhorar o código usando um método auxiliar `isFull()` para determinar se o vector está, ou não, cheio.

Programando a Agenda de Contactos

```
public class ContactBook {
    ...
    private Contact[] contacts; // vector de contactos
    private int counter; // numero de contactos no vector
    ...

    // Pre: !hasContact(other.getName())
    public void addContact(Contact contact){
        if (isFull())
            resize();
        contacts[counter++] = contact;
    }
}
```

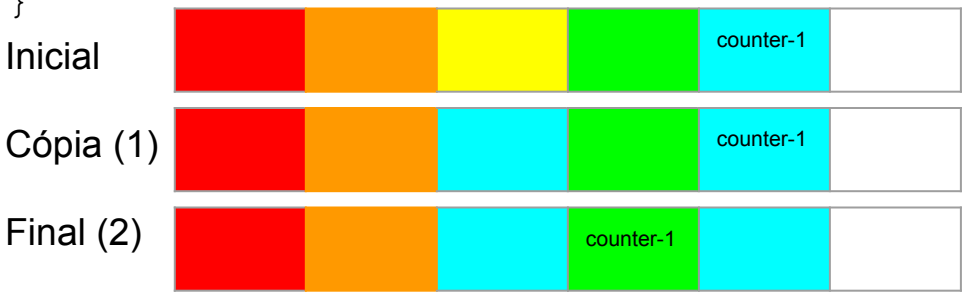
Atenção: cada elemento de um vector de objectos tem de ser criado explicitamente na classe onde este método for invocado.

Programando a Agenda de Contactos

```
public class ContactBook {
    private static final int GROWTH_FACTOR = 2;
    ...
    private Contact[] contacts; // vector de contactos
    private int counter; // numero de contactos no vector
    ...
    // Pre: !hasContact(name)
    public void addContact(Contact contact){ ... }

    private void resize() {
        Contact[] tmp = new Contact[GROWTH_FACTOR * contacts.length];
        for (int i=0; i < counter; i++)
            tmp[i] = contacts[i];
        contacts = tmp;
    }
}
```

Programando a Agenda de Contactos

```
public class ContactBook {
    ...
    private Contact[] contacts; // vector de contactos
    private int counter; // numero de contactos no vector
    ...
    // Pre: hasContact(name)
    public void deleteContact(String name){
      contacts[searchIndex(name)] = contacts[counter-1];
      counter--;
    }
    ...
}
```



Repare que, para apagar o contacto amarelo, o que fazemos é (1) encontrar a posição onde queremos apagar e copiar para lá o último elemento da colecção. Finalmente, (2) decrementamos o contador, para não ficarmos com o mesmo elemento repetido. Note que **esta solução não preserva a ordem de inserção na colecção.**

Programando a Agenda de Contactos

```
public class ContactBook {
    ...
    private Contact[] contacts; // vector de contactos
    private int counter; // numero de contactos no vector
    ...

    // Pre: hasContact(name)
    public void setEmail(String name, String email){
        contacts[searchIndex(name)].setEmail(email);
    }

    // Pre: hasContact(name)
    public String getEmail(String name){
        return contacts[searchIndex(name)].getEmail();
    }
}
```

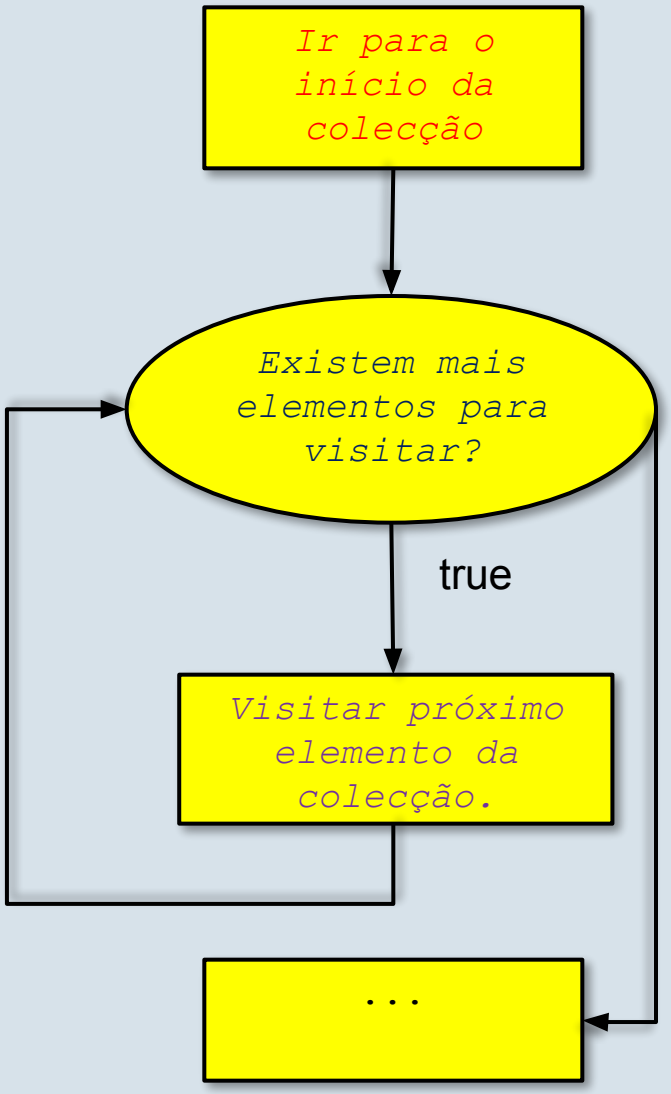
Programando a Agenda de Contactos

- Como visitar os vários elementos da colecção de contactos?
  - Dentro da classe `ContactBook` conseguimos aceder directamente à estrutura de dados (neste caso, um vector) em que os contactos estão guardados
  - Fora da classe **não queremos** que a estrutura de dados usada seja acessível, ou sequer visível
    - Se um dia a quisermos trocar por outra mais eficiente, ou mais versátil, as classes clientes da nossa classe não devem sofrer qualquer alteração
    - Além disso, se pudermos tornar a visita aos vários elementos (também conhecida como *iteração sobre vários elementos*) mais legível e reutilizável, devemos recorrer a uma solução padrão, em vez de aceder directamente ao vector

Programando a Agenda de Contactos

- Devemos recorrer a um **iterador** para resolver este problema. O iterador oferece:
  - Uma forma consistente de iterar sobre estruturas de dados
    - Neste momento estamos a trabalhar com vectores, mas há muitas outras estruturas de dados que podem ser iteradas (vai conhecer bastantes nas cadeiras de POO e AED) de forma consistente, usando um iterador.
  - Três operações:
    - Ir para o início da colecção a visitar
    - Testar se existem mais elementos a visitar
    - Visitar o próximo elemento da colecção

Programando a Agenda de Contactos



Três operações necessárias:

`void initializeIterator()`
Ir para o início da colecção.

`boolean hasNext()`
Testar se existem mais elementos para visitar.

`type` `next()`
Visitar o próximo elemento.
`type` deve ser substituído pelo tipo dos elementos da colecção (neste caso, `Contact`).

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()`

Inicia a iteração sobre os contactos da agenda

`public boolean hasNext()`

Indica se existe um contacto seguinte.

Quando se chega ao fim da travessia, devolve **`false`**

caso contrário devolve **`true`**

`public Contact next()`

Devolve o contacto corrente

`Pre:` `hasNext()`

Agenda de Contactos

- Variáveis de instância – além do vector acompanhado (vector de contactos e número que indica quantos contactos existem no vector), precisamos de guardar:
  - *Índice* do contacto corrente quando estamos a iterar *todos* os contactos da agenda.

Classe ContactBook

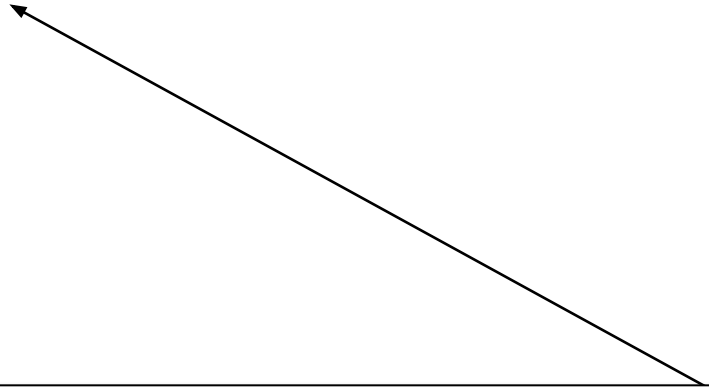
```
public class ContactBook{
    private static final int MAX_CONTACTS = 50;

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

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

Posição do contacto corrente

(usado quando se itera os contactos)

Classe ContactBook

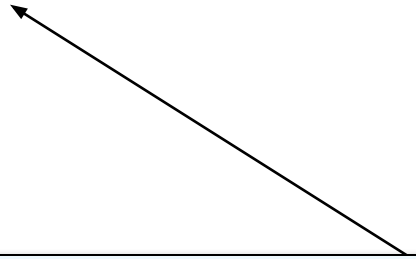
```
public class ContactBook{
    ...
    private Contact[] contacts; // vector de contactos
    private int counter; // numero de contactos no vector
    private int currentContact; // posicao do contacto corrente
    ...
    public void initializeIterator(){
        currentContact = 0;
    }
    ...
}
```



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

Classe ContactBook

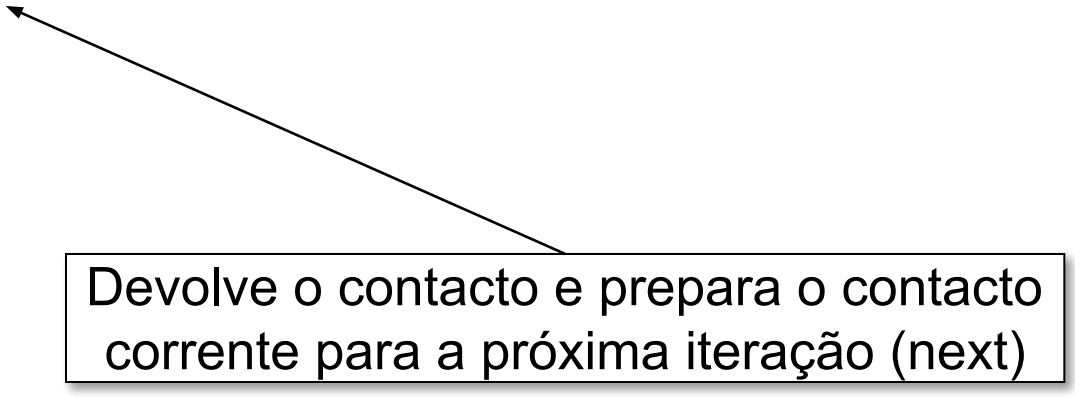
```
public class ContactBook{
    ...
    private Contact[] contacts; // vector de contactos
    private int counter; // numero de contactos no vector
    private int currentContact; // posicao do contacto corrente
    ...
    public boolean hasNext(){
        return ((currentContact >= 0 ) &&
                (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
    ...
    //Pre: hasNext()
    public Contact next(){
        return contacts[currentContact++];
    }
}
```



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 */
    private static void listAllContacts(ContactBook cBook){
        cBook.initializeIterator();
        while( cBook.hasNext() ) {
            Contact c = cBook.next();
            System.out.println(  c.getName() + ";"
                               + c.getEmail()+ ";"
                               + c.getPhone() );
        }
    }
}
```

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(){
    currentContact = 0;
}
public boolean hasNext(){
    return ( (currentContact >= 0) && (currentContact < counter) );
}
public Contact next(){
    return accounts[currentContact++];
}
```

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(){
    currentContact = 0;
}
public boolean hasNext(){
    return ( (currentContact >= 0) && (currentContact < counter) );
}
public Contact next(){
    return accounts[currentContact++];
}
```

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(){
    currentContact = 0;
}
public boolean hasNext(){
    return ( (currentContact >= 0) && (currentContact < counter) );
}
public Contact next(){
    return accounts[currentContact++];
}
```

Iteradores

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

Incremento do índice `i` corresponde ao método `next()`

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

public void initializeIterator(){
    currentContact = 0;
}
public boolean hasNext(){
    return ( (currentContact >= 0) && (currentContact < counter) );
}
public Contact next(){
    return accounts[currentContact++];
}
```

E se quisermos importar todos os contactos de outra agenda?

- Necessitamos de, um a um, copiar os não repetidos, da outra agenda para a nossa - usar de novo um iterador!
- Acrescenta-se à classe `ContactBook` o método `importContacts()`

```
public void importContacts(ContactBook other) {
        other.initializeIterator();
        while (other.hasNext()) {
            Contact contact = other.next();
            if (!foundContact(contact))addContact(contact);
        }
}
private boolean foundContact(Contact other) {
    boolean found = false;
    for (int i=0; i < counter && !found; i++)
        if (contacts[i].equals(other)) found = true;
    return found;
}
```

Comparar objectos: o método `equals`

Equivalência de objectos:

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

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

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

- Em Java, os operadores `==` e `!=` comparam **referências** de objectos.
  - Ou seja, `a == b` testa se `a` e `b` são, de facto, **duas referências** para **o mesmo objecto em memória** , ou seja, **compara a identidade de objectos**
  - Em contrapartida `a != b` testa se `a` e `b` são duas referências para objectos distintos, ou seja, com diferentes identidades, mesmo que sejam equivalentes
  - Exemplo:

```
Contact a = new Contact("Joao Dias",219999,"jd@fct.unl.pt");
Contact b = new Contact("Joao Dias",210000,"jd@fct.unl.pt");
boolean x = (a==b); // false, a e b têm identidades diferentes!
boolean y = a.equals(b); // true, so estamos a comparar os nomes
```

Método `equals`

- Todas as classes têm, por omissão, um método `equals` já pré-definido
  - Frequentemente, as classes de bibliotecas do Java têm um `equals` que compara os valores dos objectos
    - Na classe `String`, se a sequência de caracteres do parâmetro for igual à do objecto **`this`**, `equals` devolve **`true`**. Caso contrário, devolve **`false`**.
  - As classes que definimos, por omissão, têm um `equals` equivalente ao ==.
  - Normalmente, essa definição não serve os nossos propósitos no domínio da aplicação, portanto, temos de o redefinir, especificando para tal o nosso próprio `equals`
  - Ao programar o método `equals`, define-se qual é o critério de comparação pretendido:
    - A comparação envolve normalmente todas variáveis de instância (mas podemos usar apenas algumas, se a especificação do problema o permitir – exemplo: podemos usar apenas o nome para comparar contactos, se assumirmos que o nome é um identificador único de contacto e que, portanto, dois contactos com nomes iguais se consideram iguais).

Programando a classe `Contact`

```
public class Contact{
    private String name;
    private int phone;
    private String email;
    // pre: name != null && email!= null
    public Contact(String name, int phone, String email){…}
    public String getName() { … }
    public int getPhone() { … }
    public String getEmail() { … }
    public void setPhone(int phone) { … }
    // pre: email!= null
    public void setEmail(String email) { … }
    // pre: otherContact != null
    public boolean equals(Contact otherContact) {
        return name.equals(otherContact.getName());
    }
}
```

Método `equals` da classe String, que verifica se duas sequências de caracteres são iguais.

E se quisermos apresentar a agenda ordenada?

- Necessitamos de acrescentar um método que ordena uma agenda
- Vamos usar o algoritmo bubblesort, para implementar a ordenação do vector de contactos

```
public void bubbleSort(Contact[] contacts)
```

- Podemos usar este método para ordenar a agenda por ordem alfabetica.

E se quisermos a agenda ordenada?

Na classe `Contact`, necessitamos de ter um método para comparar dois contactos, usando a ordem alfabética dos nomes.

```
public boolean greaterThan(Contact other) {
    return this.getName().compareTo(other.getName()) > 0;
}
```

Na classe `ContactBook`, implementamos o algoritmo BubbleSort (veja a definição detalhada deste algoritmo nos slides com operações sobre vectores).

```
public void bubbleSort(Contact[] contacts) {
    for (int i = 1; i < counter - 1; i++)
        for (int j = counter - 1; j >= i; j--)
            if (contacts[j-1].greaterThan(contacts[j])) {
                Contact temp = contacts[j - 1];
                contacts[j-1] = contacts[j];
                contacts[j] = temp;
            }
}
```

Comando para listagem ordenada

Considere-se um comando extra para listar a informação dos contactos da agenda por ordem alfabética de nome.

Coloca-se na classe Main o seguinte método auxiliar que:

1. faz uma cópia da agenda
   `new ContactBook().importContacts(cbook)`
2. Ordena-a, usando um outro método bubbleSort, agora na classe Main, que:
   a. Cópia o vector de contactos da agenda, com o iterador, e
   b. ordena-o, chamando então o bubbleSort da classe `ContactBook`
3. imprime a infomação de todos os contactos na agenda, com o método `listAllContacts`.

```
Private static void listAllContactsSorted(ContactBook cbook){
  listAllContacts(bubbleSort(new ContactBook().importContacts(cbook)));
}
```

Iterador para listagem ordenada

Outra alternativa é implementar um iterador ordenado.

Considere-se que a classe `ContactBook` tem agora também as variáveis de instância `currentOrd` e `iteraOrd`, e os métodos `hasNextOrd` e `nextOrd` para lidar com um iterador ordenado.

A inicialização do iterador ordenado implica obter uma cópia ordenada do vector de contactos (para depois os imprimir por ordem alfabética).

```
public void initializeOrdIterator() {
    //copia de vector
    iteraOrd = new Contact[counter];
    for(int i = 0; i < counter; i++)
        iteraOrd[i] = contacts[i];
    // ordenar contactos por nome
    bubbleSort(iteraOrd);
    currentOrd = 0;
}
```

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.
```

Repare que estamos a passar como argumentos duas referências para os objectos, `in` e `cBook`. Por estarmos a passar referências para objectos, o que acontecer a estes objectos dentro do método será visível fora do método. Por outras palavras, no final deste método, o `Scanner in` terá lido mais algum input e o `ContactBook cBook` poderá ter novos contactos!

```
private static void addContact(Scanner in, ContactBook cBook) {
    String name = in.nextLine();
    int phone=in.nextInt();
    in.nextLine();
    String email = in.nextLine();
    if (cBook.hasContact(name))
        System.out.println(CONTACT_EXISTS);
    else  {
        cBook.addContact(new Contact(name, phone, email));
        System.out.println(CONTACT_ADDED);
    }
}
```

Classe Main: constantes

```
public class Main {
    ...
    //Constantes que definem os comandos
    public static final String ADD_CONTACT      = "AC";
    public static final String IMPORT_CONTACTS  = "IC";
    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
    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.";
    public static final String BYE = "Goodbye.";

    ...
}
```

Classe `Main`: método `getCommand()`

```
public class Main {
    ...
    /** Metodo 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

```
public static void main(String[] args) {
    Scanner in = new Scanner(System.in);
    ContactBook cBook = new ContactBook();
    String comm = "";
    do {
        comm = getCommand(in);
        if (!comm.equals(QUIT)) {
            if (comm.equals(ADD_CONTACT))
                addContact(in,cBook);
            else if …
                …
            else
                System.out.println(WRONG_COMM);
        }
  } while (!comm.equals(QUIT));
  System.out.println(BYE);
  in.close();
}
```

Método `main`: Interpretador de comandos

```
public static void main(String[] args) {
    Scanner in = new Scanner(System.in);
    ContactBook cBook = new ContactBook();
    String comm = "";
    do {
        comm = getCommand(in);
        if (!comm.equals(QUIT)) {
            switch (comm) {
            case ADD_CONTACT:
                addContact(in,cBook); break;
                …
            default:
                System.out.println(WRONG_COMM);
            }
        }
    } while (!comm.equals(QUIT));
  }
  System.out.println(BYE);
  in.close();
}
```

Também pode ser implementado com um switch

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 int getPhone(...){...}
   private static String getEmail(...){...}
   private static void setPhone(...){...}
   private static void setEmail(...){...}
   private static void listAllContacts (...){...}
}
```

Classes do domínio

```
public class ContactBook {
   public boolean hasContact(...){...}
   public int getNumberOfContacts(){...}
   public void addContact(...){...}
   public void importContacts(...){...}
   public int getPhone(...){...}
   public String getEmail(...){...}
   public void deleteContact(...){...}
   public void setPhone(...){...}
   public void setEmail(...){...}
   public void initializeIterator(...){...}
   public boolean hasNext(){...}
   public Contact next(){...}
}
```

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