Iteradores

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

Corpo Docente 2020/2021
António Ravara, Artur Miguel Dias, Bernardo Toninho,
Ema Vieira, Inês Fernandes, Margarida Mamede,
Miguel Monteiro, Rui Nóbrega

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)
```

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 (...){...}
  ...
}
```

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) {...}
}
```

Interface da classe Contact

- Operações reconhecidas (interface de `Contact`):

`String getName()`
Devolve o nome do contacto

`String getPhone()`
Devolve o telefone do contacto

`String getEmail()`
Devolve o e-mail do contacto

`void setPhone(String phone)`
Altera o telefone do contacto para `phone`
pre: `phone != null`

`void setEmail(String email)`
Altera o email do contacto para `email`
pre: `email != null`

`boolean equals(Contact otherContact)`
Retorna `true`, se o nome do contacto corrente é igual ao do contacto dado
pre: `otherContact != null`

Agenda de Contactos

- Operações reconhecidas (interface de `ContactBook`):

```
public boolean hasContact(String name)
```

Indica se existe na agenda um contacto com o nome dado

Pre: `name != null`

```
public int getNumberOfContacts()
```

Indica o número de contactos na agenda

```
public void addContact(String name, String phone, String email)
```

Adiciona um novo contacto na agenda

Pre: `!hasContact(name) && phone != null && email != null`

```
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 String 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, String 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)`

Programando a Agenda de Contactos

- Como visitar os vários elementos da colecção de contactos?
  - Dentro da classe `ContactBook` conseguimos ter acesso directo 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
    - 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 efectuar acessos directos ao vector

Programando a Agenda de Contactos

- Devemos recorrer a um **objecto iterador** para resolver este problema. O iterador oferece:
  - Uma forma consistente de percorrer os elementos de uma estrutura de dados
    - Neste momento estamos a trabalhar com vectores, mas há muitas outras estruturas de dados que podem ser percorridas de forma consistente, usando um iterador
      (vai conhecer várias nas disciplinas **POO e AED**).
  - Um iterador inclui um construtor e duas operações:
    - **Criação do iterador**, posicionado no início da colecção
    - Testar **se existem** mais elementos a visitar
    - **Visitar** (devolver) o próximo elemento da colecção

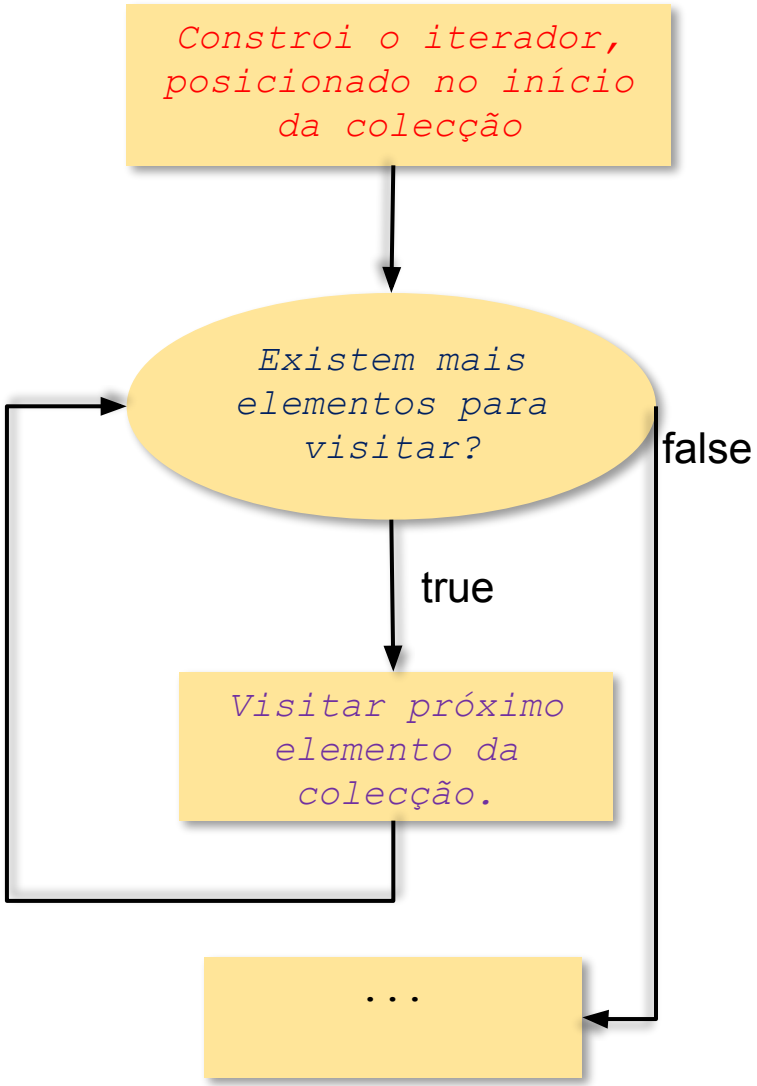
Programando a Agenda de Contactos

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

    public ContactIterator iterator() {
        return new ContactIterator(contacts, counter);
    }
}
```

A operação `iterator()` é a maneira habitual das colecções darem aos clientes a possibilidade de iterar sobre os seus elementos, sem que os clientes criem dependências dos detalhes de implementação da colecção.
Ao ser criado, o objecto iterador recebe `contacts` e o `counter` pois precisa deles para manter, no seu estado interno, o ponto em que se encontra a travessia dos elementos.
Para que a ocultação `contacts` e `counter` não seja violada, o iterador deve ser criado dentro da colecção - daí a operação `iterator()`.

Programando a Agenda de Contactos



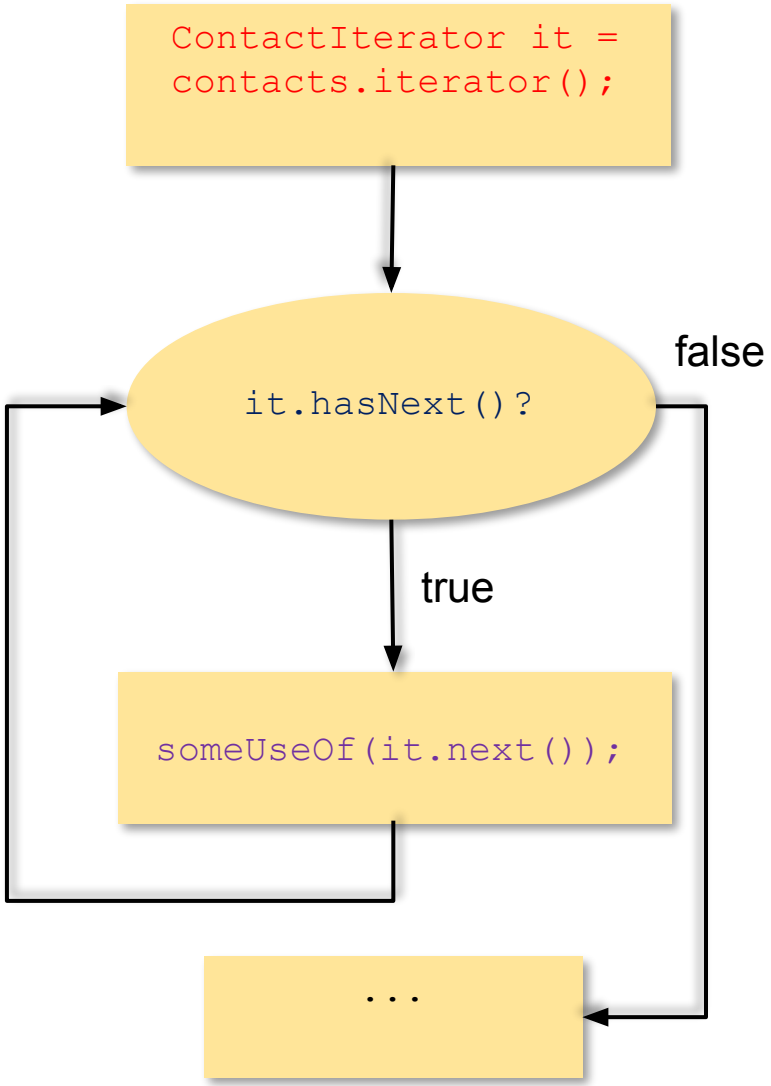
Três operações necessárias:

new *Construtor*(...)
Cria o iterador, posicionado no início.

boolean hasNext()
Testa se existem mais elementos para visitar.

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

Utilizar o iterador



Exemplo

```
//criar agenda
ContactBook contacts = new ContactBook();

// ...carregar dados na agenda...

//criar Iterador (método iterator usa
//internamente o construtor do iterador
ContactIterator it = contacts.iterator();

//usar Iterador
while (it.hasNext())

    someUseOf(it.next());

//terminou iteração
```

Iterador da Agenda de Contactos

`public ContactIterator(...)`

Inicializa o iterador para o primeiro contacto da agenda

- Operações reconhecidas (interface de `ContactIterator`):
  - Para percorrer a agenda vamos usar uma sequência das três operações para iterar sobre todos os contactos.
  - Assumimos que durante o percurso pela agenda não são feitas inserções nem remoções de contactos.

`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()*

Iterador da Agenda de Contactos

- **Variáveis de instância**: além do vector de contactos e do número que indica quantos contactos existem no vector), precisamos de guardar:

  - *Índice* do próximo contacto a devolver pelo iterador (quando estamos a iterar *todos* os contactos da agenda).

Classe ContactIterator

```
public class ContactIterator {
   private Contact[] contacts; // vector de contactos
   private int counter; // número de contactos no vector
   private int nextContact; // posição do próximo contacto
   ...
}
```

Classe ContactIterator

```
public class ContactIterator {
    private Contact[] contacts;
    private int counter;
    private int nextContact;

    // Construtores
    public ContactIterator(                    ...                    ) {
        ...
        ...
        nextContact = 0;
    }
    ...
}
```

Posição do próximo contacto

(usado apenas para iterar os contactos)

Classe ContactIterator

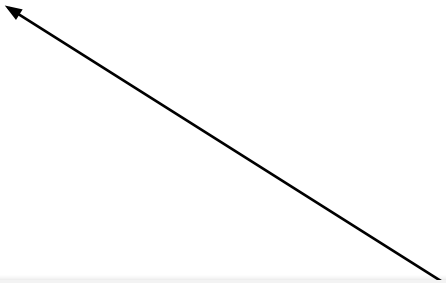
```
public class ContactIterator {
    private Contact[] contacts; // vector de contactos
    private int counter; // número de contactos no vector
    private int nextContact; // posição do próximo contacto

    // Construtores
    public ContactIterator(Contact[] contacts, int counter) {
        this.contacts = contacts;
        this.counter = counter;
        nextContact = 0;
    }
    ...
}
```

O iterador recebe `contacts` e o `counter` no seu construtor. Isto dá a impressão de que o objecto colecção está a expor todo o seu estado interno. Porém, na prática não está, pois a criação do próprio objecto iterador é feita numa operação da colecção - o método `iterator()`.

Classe ContactIterator

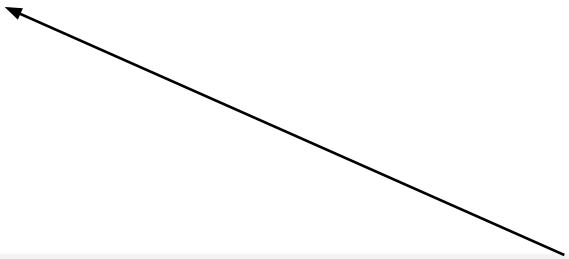
```
public class ContactIterator {
   private Contact[] contacts; // vector de contactos
   private int counter; // número de contactos no vector
   private int nextContact; // posição do próximo contacto
   ...
   public boolean hasNext(){
       return nextContact < counter;
   }
   ...
}
```



Só existem mais contactos para percorrer enquanto a posição do próximo contacto for menor que o número de elementos

Classe ContactIterator

```
public class ContactIterator {
   private Contact[] contacts; // vector de contactos
   private int counter; // número de contactos no vector
   private int nextContact; // posição do próximo contacto
   ...
   /** Pre: hasNext() */
   public Contact next(){
      return contacts[ nextContact++ ];
   }
}
```



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

Classe ContactIterator

```
public class ContactIterator {
   private Contact[] contacts; // vector de contactos
   private int counter; // número de contactos no vector
   private int nextContact; // posição do próximo contacto

   public ContactIterator(Contact[] contacts, int counter) {
       this.contacts = contacts;
       this.counter = counter;
       nextContact = 0;
   }
   public boolean hasNext(){
       return nextContact < counter;
   }
   /** Pre: hasNext() */
   public Contact next(){
       return contacts[nextContact++];
   }
}
```

Classe Main

Exemplo de uso do iterador `ContactIterator` de `ContactBook`:

```
public class Main {

    /** Listar todos os contactos na consola */
    private static void listAllContacts(ContactBook cBook){
        ContactIterator it = cBook.iterator();
        while( it.hasNext() ) {
            Contact c = it.next();
            System.out.println( c.getName() + ";" +
                c.getEmail()+ ";" + c.getPhone() );
        }
    }
    public static void main(String[] args) {
        ContactBook cBook = new ContactBook();
        //...
    }
}
```

Iteradores

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

O ciclo for, imprimindo no ecrã (localizado na main) teria de ter acesso ao vector de contactos. Não seria possível pois todas as várias de instância da agenda são privadas.

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

```
public ContactIterator(Contact[] contacts, int counter) {
    //...
    nextContact = 0;
}
public boolean hasNext(){
    return nextContact < counter;
}
public Contact next(){
    return accounts[ nextContact++ ];
}
```

Iteradores

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

Inicialização do i corresponde à criação do iterador

```
for( int i = 0; i < counter; i++ ) {
    System.out.println(contacts[i].getName() + ";" +
    contacts[i].getEmail()+ ";" + contacts[i].getPhone());
}
public ContactIterator(Contact[] contacts, int counter) {
    //...
    nextContact = 0;
}
public boolean hasNext(){
    return nextContact < counter;
}
public Contact next(){
    return accounts[ nextContact++ ];
}
```

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].getName() + ";" +
    contacts[i].getEmail()+ ";" + contacts[i].getPhone());
}
public ContactIterator(Contact[] contacts, int counter) {
    //...
    nextContact = 0;
}
public boolean hasNext(){
    return nextContact < counter;
}
public Contact next(){
    return accounts[ nextContact++ ];
}
```

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].getName() + ";" +
    contacts[i].getEmail()+ ";" + contacts[i].getPhone());
}
public ContactIterator(Contact[] contacts, int counter) {
    //...
    nextContact = 0;
}
public boolean hasNext(){
    return nextContact < counter;
}
public Contact next(){
    return contacts[ nextContact++ ];
}
```

E se quisermos importar todos os contactos de outra agenda?

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

```
/** Pre: other != null */
public void importContacts(ContactBook other) {
   ContactIterator it = other.iterator();
   while (it.hasNext()) {
       Contact contact = it.next();
       if (!hasContact(contact.getName()))
          addContact(contact.getName(),
                     contact.getPhone(),contact.getEmail() );
   }
}
```

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 **`static`** 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( ... ) { ... }
    private static void listAllContactsSorted( ... ) { ... }
}
```

Classe Main: método `addContact()`

```
>AC 213334455 joana@ggg.tt.ptJoana Dias
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 phone=in.next();
    String email = in.next();
    String name = in.nextLine().trim();
    if (cBook.hasContact(name))
        System.out.println(CONTACT_EXISTS);
    else  {
        cBook.addContact(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 LIST_ORDERED     = "LO";
    public static final String QUIT             = "Q";

    public static final String PROMPT             = "> ";

    //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) {
        System.out.print(PROMPT);
        String input = in.next().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)) {
            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();
}
```

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 (...){...}
  ...
}
```

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) {...}
}
```