Vectores de objectos

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

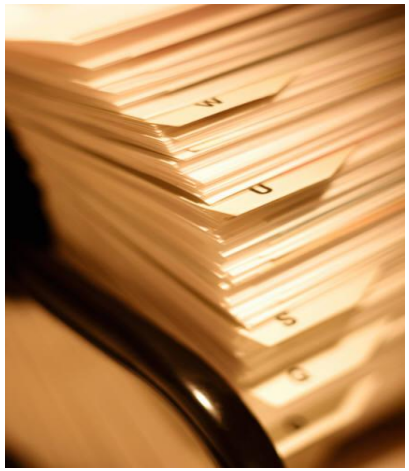
- O programa principal deve dar ao utilizador a possibilidade de execução de diversas operações numa agenda, no número que o utilizador pretender.
- Interface de utilização do programa: comandos

```
> 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)
```

Agenda de Contactos

- Desenvolver em Java uma classe `ContactBook` cujos objectos são agendas de contactos.
  - Cada objecto desta classe contém um conjunto de contactos.

    Logo é necessário definir a classe `Contact`, cujos 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 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) {...}
  public boolean equals(Contact other) {...}
}
```

Agenda de Contactos

- Uma agenda de contactos é um conjunto de vários contactos.
  - Cada contacto é identificado pelo seu nome.
  - Não podem existir contactos diferentes com o mesmo nome.

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(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)`

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 count;
    private Contact[] contacts;


    // Construtores
    public ContactBook() {
        count = 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();
    //...insercao de dados ...
    cb.addContact("João", 2538899, "j@gmail.pt");
    cb.addContact("Ana", 21248775, "a@gmail.pt");
    int res = cb.searchIndex("João Martins");
}
}
```

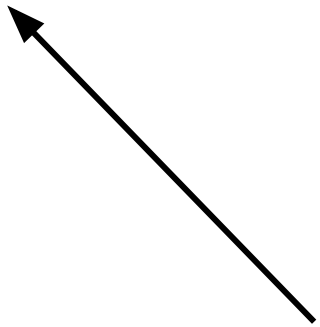
Chamada a método privado: erro

Programando a Agenda de Contactos

```
public class ContactBook {
    ...
    private Contact[] contacts; // vector de contactos
    private int count; // numero de contactos no vector
    ...
    private int searchIndex(String name){
        int i = 0; // percurso pelos elementos 0..count-1

        while(i < count && !contacts[i].getName().equals(name))
            i++;

        if(i == count)
            return -1;
        else
            return i;
    }
    ...
}
```



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 count; // numero de contactos no vector
    ...
    private int searchIndex(String name){
        int i = 0; // percurso pelos elementos 0..count-1

        while(i < count && !contacts[i].getName().equals(name))
            i++;

        if(i == count)
            return -1;
        else
            return i;
    }
    ...
}
```

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 count; // numero de contactos no vector
    ...
    private int searchIndex(String name){
        int i = 0; // percurso pelos elementos 0..count-1

        while(i < count && !contacts[i].getName().equals(name))
            i++;

        if(i == count)
            return -1;
        else
            return i;
    }
    ...
}
```

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 count; // numero de contactos no vector
    ...
    private int searchIndex(String name){
        int i = 0; // percurso pelos elementos 0..count-1
        int pos = -1; // por enquanto, ainda nao encontramos o elemento

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

Outra versão

Programando a Agenda de Contactos

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



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









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

Programando a Agenda de Contactos

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



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

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

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 count; // numero de contactos no vector
    ...

    /** Pre: !hasContact(name)&& phone != null && email != null */
    public void addContact(String name, String phone, String email){
        if (isFull()) resize();
        contacts[count++] = new Contact(name, phone, email);
    }
}
```

O contacto é adicionado no final do vector de contactos.

Programando a Agenda de Contactos

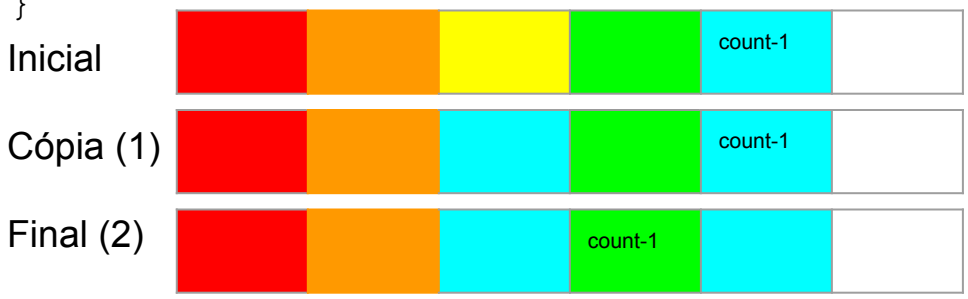
```
public class ContactBook {
    private static final int GROWTH_FACTOR = 2;
    ...
    private Contact[] contacts; // vector de contactos
    private int count; // numero de contactos no vector
    ...
    /** Pre: !hasContact(name)&& phone != null && email != null */
    public void addContact(String name, String phone, String email){ ... }

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

Métodos de apoio ao `addContact()`.

Programando a Agenda de Contactos

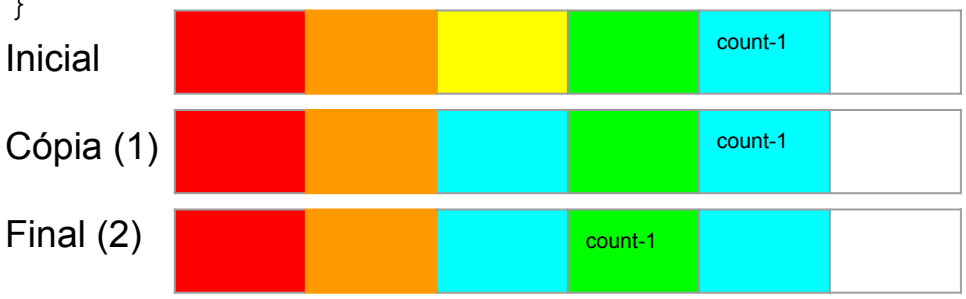
```
public class ContactBook {
    ...
    private Contact[] contacts; // vector de contactos
    private int count; // numero de contactos no vector
    ...
    /** Pre: hasContact(name)*/
    public void deleteContact(String name){
      contacts[searchIndex(name)] = contacts[count-1];
      count--;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



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 count; // numero de contactos no vector
    ...
    /** Pre: hasContact(name)*/
    public void deleteContact(String name){
      contacts[searchIndex(name)] = contacts[count-1];
      count--;
    }
    ...
}
```



Se `name` não identificar nenhum contacto na agenda (ver pré-condição), o método aborta.
A existência de `name` na agenda tem **MESMO** de ser verificada antes da chamada ao método, na classe que o chama.

Programando a Agenda de Contactos

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

    /** Pre: hasContact(name)&& email != null */
    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();
    }
}
```

Verificar **SEMPRE** pré-condições no exterior, antes da chamada dos métodos.