Iteradores Ordenados

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) {...}
}
```

E se quisermos apresentar a agenda ordenada?

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

```
public void sort()
```

- Podemos usar este método para ordenar a agenda por ordem alfabética (do nome dos contactos) crescente.

E se quisermos a agenda ordenada?

Na classe **`Contact`**, necessitamos de ter algum método para comparar dois contactos - neste caso usando a ordem alfabética dos nomes.

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

Na classe **`ContactBook`**, implementamos o algoritmo *Bubble Sort* (veja a definição detalhada deste algoritmo nos diapositivos de sistematização das operações sobre vectores).

```
public void sort() {
    for (int i = 1; i < count; i++)
        for (int j = count - 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: criar novo objecto agenda e importar todos os contactos da agenda actual
2. Ordenar os contactos da nova agenda, usando um método `sort`
3. Usar o método `listAllContacts` para imprimir a informação de todos os contactos na nova agenda ordenada.

```
private static void listAllContactsSorted(ContactBook cbook) {
   ContactBook cbook2 = new ContactBook();
   cbook2.importContacts(cbook);
   cbook2.sort();
   listAllContacts(cbook2);
}
```

Outra maneira de implementar a listagem ordenada

- A maneira de realizar a listagem ordenada mostrada nos diapositivos anteriores funciona, mas...

- Requer *várias* travessias pelo vector de contactos aquando da construção do iterador ordenado
  - Uma primeira travessia para importar os contactos para o segundo objecto agenda
  - Uma segunda, para ordenar os contactos (na prática, tem um custo computacional ainda superior)
  - Uma terceira, para efectuar a própria listagem

- Embora a abordagem atrás mostrada seja aceitável, podemos pensar numa alternativa, que continua a cumprir o requisito de não alterar a ordem inicial dos contactos.

Outra maneira de implementar a listagem ordenada

- Uma boa alternativa, passa por basear a construção do objecto iterador na *inserção ordenada* dos contactos.
  - Deixa de ser necessário usar um segundo objecto agenda de contactos
  - Deixa de ser necessário usar uma operação de ordenação
- Para a inserção ordenada - numa nova classe, variante de `ContactIterator`: `ContactIteratorOrd` - podemos aproveitar o algoritmo da operação `insertAt()` apresentada numa aula anterior
  - Essa operação recebe a posição (índice) - `pos` - em que se pretende inserir o novo elemento
  - Temos pois de criar uma operação (`insertSort()`) que identifique essa posição e que chame `insertAt()`, partindo do princípio que os elementos já existentes se encontram ordenados

Outra maneira de implementar a listagem ordenada: `ContactIteratorOrd`

As partes públicas da variante do iterador a seguir apresentada - iterador ordenado: `ContactIteratorOrd` - são idênticas às de `ContactIterator`. Também as suas variáveis de instância.

```
public class ContactIteratorOrd {
    private Contact[] contacts;
    private int count;
    private int nextContact;

    public ContactIteratorOrd(Contact[] toIterate, int count) { ... }

    ...

    public boolean hasNext() {
        return nextContact < contacts.length;
    }

    /** Pre: hasNext() */
    public Contact next() {
        return contacts[ nextContact++ ];
    }
}
```

Outra maneira de implementar a listagem ordenada: `ContactIteratorOrd`

Na classe `ContactIteratorOrd`, a inicialização do objecto iterador (no construtor) é mais complexa:

- A classe continua a ter campos `contacts`, `count` e `nextContact`
- O construtor continua a receber o vector e `count`
  - Porém, o que faz com eles tem algumas diferenças
- O construtor cria o seu próprio objecto vector `contacts`, com dimensão à medida de `count`
- O construtor percorre todos os elementos do vector recebido (até `count` *exclusivé*), inserindo ordenadamente cada um no novo vector
- Incializa o seu `nextContact` a zero, tal como em `ContactIterator`.

```
public ContactIteratorOrd(Contact[] toIterate, int count) {
    this.contacts = new Contact[count];
    this.count = 0;
    for (int i = 0; i < count; i++)
        this.insertSort(toIterate[i]);
    nextContact = 0;
}
```

`InsertSort()` para contactos

```
private void insertSort(Contact contact) {
    int pos=-1;
    int i=0;
    if (isFull()) resize();
     while (i<count && pos==-1)
        if (contacts[i].greaterThan(contact))
           pos=i;
        else i++;
    if (pos == -1)
      pos = count;
    insertAt(contact,pos);
}

private void insertAt(Contact contact, int pos) {
    openSpace(pos); //abre caminho para o novo elemento
    contacts[pos] = contact;
    count++;
}
```