Interfaces e Classes Internas

SCC0604 - Programação Orientada a Objetos

Prof. Fernando V. Paulovich
http://www.icmc.usp.br/~paulovic
paulovic@icmc.usp.br

Instituto de Ciências Matemáticas e de Computação (ICMC)
Universidade de São Paulo (USP)

25 de julho de 2010

Sumário

Sumário

Introdução

- Aqui apresentaremos duas ferramentas que junto com a herança formam o conjunto necessário de técnicas para construir programas orientados a objetos

Introdução

- Aqui apresentaremos duas ferramentas que junto com a herança formam o conjunto necessário de técnicas para construir programas orientados a objetos
  - A primeira, chamada de **interface**, **dá suporte a existência de herança múltipla**

Introdução

- Aqui apresentaremos duas ferramentas que junto com a herança formam o conjunto necessário de técnicas para construir programas orientados a objetos
  - A primeira, chamada de **interface**, **dá suporte a existência de herança múltipla**
  - A segunda, denominada **classes internas**, possibilita que uma **classe seja definida dentro de outra**

Sumário

Interfaces: Uso de uma Superclasse Abstrata

- O mecanismo de **herança** e **classes abstratas** permite a **programação genérica**
- Programação genérica consiste em definir **algoritmos e procedimentos** que são aplicados a classes-base, mas cujo comportamento é **determinado dinamicamente** (em tempo de execução) dependendo do objeto (de uma classe derivada) que é usado
- A seguir um exemplo é apresentado

Exemplo: Programação Genérica

```
1  public abstract class Sortable {
2    /**
3    * Compara a outro objeto para determinar se é maior ou menor.
4    * @param b objeto para comparação.
5    * @return zero se this == b, positivo se this > b e negativo se this < b.
6    **/
7    public abstract int compareTo(Sortable b);
8  }
```

```
1  public class Util {
2    public static void sort(Sortable[] array) {
3      Sortable aux = null;
4
5      for(int i = 0; i < (array.length - 1); i++) {
6        for(int j = 0; j < ((array.length - 1) - i); j++) {
7          if(array[j].compareTo(array[j+1]) > 0) {
8            aux = array[j];
9            array[j] = array[j+1];
10           array[j+1] = aux;
11         }
12       }
13     }
14   }
15 }
```

Exemplo: Programação Genérica

```
1  public class Data extends Sortable {
2    ...
3    public int compareTo(Sortable b) {
4      int valor1 = 0, valor2 = 0;
5
6      valor1 = (this.ano-1)*365+(this.mes-1)*30+this.dia;
7
8      if(b instanceof Data) {
9        Data aux = (Data)b;
10       valor2 = (aux.ano-1)*365 + (aux.mes-1)*30+aux.dia;
11     }
12
13     return (valor1-valor2);
14   }
15   ...
16  }
```

Sumário

Uso de Interfaces

- Java não suporta **herança múltipla**, mas um efeito parecido pode ser alcançado usando-se interfaces

Uso de Interfaces

- Java não suporta **herança múltipla**, mas um efeito parecido pode ser alcançado usando-se interfaces
- Uma interface é um **contrato** de que alguma classe irá implementar certos métodos com certas características

Uso de Interfaces

- Java não suporta **herança múltipla**, mas um efeito parecido pode ser alcançado usando-se interfaces
- Uma interface é um **contrato** de que alguma classe irá implementar certos métodos com certas características
- Uma interface é declarada da mesma forma que uma classe, somente substituindo a palavra-chave **class** por **interface**

Uso de Interfaces

- A classe que implementa uma interface indica isso usando a palavra-chave **implements** (diferente de **extends**)

```
1  public interface Comparable {
2    /**
3    * Compara a outro objeto para determinar se é maior ou menor.
4    * @param b objeto para comparação.
5    * @return zero se this == b, positivo se this > b e negativo se this < b.
6    **/
7    public int compareTo(Object b);
8  }
```

```
1  public class Data implements Comparable {
2    public int compareTo(Object b) {
3      ...
4    }
5  }
```

Uso de Interfaces

```
1  public class Util {
2    public static void sort(Comparable[] array) {
3      Comparable aux = null;
4
5      for(int i = 0; i <= (array.length - 2); i++) {
6        for (int j = 0; j <= ((array.length - 2) - i); j++) {
7          if (array[j].compareTo(array[j+1]) > 0) {
8            aux = array[j];
9            array[j] = array[j+1];
10           array[j+1] = aux;
11           }
12         }
13       }
14     }
15 }
```

Propriedades das Interfaces

- Embora não seja possível criar objetos a partir de interfaces, pode-se declarar variáveis

```
Comparable data = new Data(1,1,2004);
```

Propriedades das Interfaces

- Embora não seja possível criar objetos a partir de interfaces, pode-se declarar variáveis

```
Comparable data = new Data(1,1,2004);
```

- Hierarquias de interfaces também são possíveis de serem criadas

Propriedades das Interfaces

- Uma interface não pode conter atributos nem métodos estáticos, porém pode conter constantes

```
1 public interface Veiculo {
2   public static final int LIMITE_VEL = 110;
3   ....
4 }
```

Propriedades das Interfaces

- Uma classe pode implementar quantas interfaces for necessário

```
1 public class Data implements Cloneable, Comparable {
2 ...
3 }
```

A Interface **Cloneable**

- Ao fazer a cópia de uma variável, a original e a cópia são referências ao mesmo objeto.
- Uma alteração em qualquer uma das duas variáveis também afeta a outra

```
1 Data d1 = new Data(1,1,2004);
2 Data d2 = d1;
3 d2.setData(2,2,2005); //também altera d1
```

A Interface **Cloneable**

- Se você quiser que **d2** seja um novo objeto idêntico a **d1**, o método **clone()** deve ser empregado

```
1 Data d1 = new Data(1,1,2004);
2 Data d2 = (Data)d1.clone(); //precisa converter tipo
3 d2.setData(2,2,2005); //d1 não é alterado
```

A Interface **Cloneable**

- O método **clone()** é um método protegido da classe **Object**, o que significa que o mesmo não pode ser usado diretamente

A Interface **Cloneable**

- O método **clone()** é um método protegido da classe **Object**, o que significa que o mesmo não pode ser usado diretamente
- O método **clone()** da classe **Object** faz uma cópia bit-a-bit da classe

A Interface **Cloneable**

- Assim é de responsabilidade do projetista julgar se
  - O método **clone()** padrão é suficientemente bom

A Interface **Cloneable**

- Assim é de responsabilidade do projetista julgar se
  - O método **clone()** padrão é suficientemente bom
  - O método **clone()** padrão pode ser ajustado chamando-se **clone()** nos atributos que são objetos

A Interface **Cloneable**

- Assim é de responsabilidade do projetista julgar se
  - O método **clone()** padrão é suficientemente bom
  - O método **clone()** padrão pode ser ajustado chamando-se **clone()** nos atributos que são objetos
  - A situação não tem jeito e o **clone()** não deve ser usado

A Interface **Cloneable**

- A terceira opção é padrão, de forma que para escolher a primeira e a segunda, uma classe precisa

A Interface **Cloneable**

- A terceira opção é padrão, de forma que para escolher a primeira e a segunda, uma classe precisa
  - Implementar a interface **Cloneable**

A Interface **Cloneable**

- A terceira opção é padrão, de forma que para escolher a primeira e a segunda, uma classe precisa
  - Implementar a interface **Cloneable**
  - Redefinir o método **clone()** com o modificador de **acesso público**

A Interface **Cloneable**

- A terceira opção é padrão, de forma que para escolher a primeira e a segunda, uma classe precisa
  - Implementar a interface **Cloneable**
  - Redefinir o método **clone()** com o modificador de **acesso público**
- Se a classe que usar **clone()** não implementar a interface **Cloneable**, uma exceção será lançada (erro em tempo de execução)

A Interface **Cloneable**

```
1  public class Data implements Cloneable {
2    ...
3    @Override
4    public Object clone() {
5      try {
6        return super.clone();
7      } catch(CloneNotSupportedException e) {
8        return null;
9      }
10   }
11 }
```

A Interface Cloneable

```
1  public class Empregado implements Cloneable {
2    private Data dataContratacao;
3    ...
4    @Override
5    public Object clone() {
6      try {
7        Empregado e = (Empregado) super.clone();
8        e.dataContratacao = (Data)dataContratacao.clone();
9        return e;
10     } catch(CloneNotSupportedException e) {
11       return null;
12     }
13   }
14 }
```

Interfaces e *Callbacks*

- É possível fazer com que uma classe chamada se comunique com a classe chamadora. Isso normalmente é chamado de *callback*

Interfaces e *Callbacks*

- É possível fazer com que uma classe chamada se comunique com a classe chamadora. Isso normalmente é chamado de *callback*
- Para se criar um implementação genérica, um conjunto ("framework") de classes e interfaces que apresente mecanismos convenientes precisa ser provido

Interfaces e *Callbacks*

Exemplo

- Definir um "framework" que possibilite que uma classe (ouvinte ou *Listener*) seja chamada automaticamente em certos intervalos de tempo

```
1 public interface TimerListener {
2     public void timeElapsed(Timer t);
3 }
```

Interfaces e *Callbacks*

```
1  public class Timer extends Thread {
2    private TimerListener listener;
3
4    public Timer(TimerListener listener) {
5      this.listener = listener;
6    }
7
8    public void run() {
9      while(true) {
10       try {
11         this.sleep(3000);
12       } catch(InterruptedException e) {
13         e.printStackTrace();
14       }
15
16       listener.timeElapsed(this);
17     }
18   }
19 }
```

Interfaces e *Callbacks*

```
1  public class AlarmClock implements TimerListener {
2
3    public AlarmClock() {
4      Timer t = new Timer(this);
5      t.start();
6    }
7
8    public void timeElapsed(Timer t) {
9       System.out.println("Acordar!!!!");
10   }
11 }
```

Interfaces e *Callbacks*

```
1  public class Main {
2    public static void main(String[] args) {
3      AlarmClock a = new AlarmClock();
4    }
5  }
```

Sumário

Classes Internas

- Em Java é possível definir classes dentro de classe, as chamadas **classes internas**

Classes Internas

- Em Java é possível definir classes dentro de classe, as chamadas **classes internas**
- Um classe interna é interessante pois

Classes Internas

- Em Java é possível definir classes dentro de classe, as chamadas **classes internas**
- Um classe interna é interessante pois
  - Um objeto de uma classe interna **pode acessar os membros privados** da classe mais externa

Classes Internas

- Em Java é possível definir classes dentro de classe, as chamadas **classes internas**
- Um classe interna é interessante pois
  - Um objeto de uma classe interna **pode acessar os membros privados** da classe mais externa
  - Classes internas são **invisíveis para outras classes** do mesmo pacote

Classes Internas

- Em Java é possível definir classes dentro de classe, as chamadas **classes internas**
- Um classe interna é interessante pois
  - Um objeto de uma classe interna **pode acessar os membros privados** da classe mais externa
  - Classes internas são **invisíveis para outras classes** do mesmo pacote
  - Classes internas anônimas são práticas quando se quer definir **callbacks** em tempo de execução

Classes Internas

- Em Java é possível definir classes dentro de classe, as chamadas **classes internas**
- Um classe interna é interessante pois
  - Um objeto de uma classe interna **pode acessar os membros privados** da classe mais externa
  - Classes internas são **invisíveis para outras classes** do mesmo pacote
  - Classes internas anônimas são práticas quando se quer definir **callbacks** em tempo de execução
  - Classes internas são muito convenientes quando se quer escrever **programas dirigidos por eventos**

Classes Internas

```
1  public class Externa {
2
3    public Externa() {
4      ...
5    }
6
7    public class Interna {
8      ...
9    }
10 }
```

Classes Internas

```
1  public class Externa {
2    private int atributo = 0;
3
4    public Externa() {
5      ...
6    }
7
8    public class Interna {
9      public Interna() {
10       atributo = 10;
11     }
12   }
13 }
```

Classes Internas

```
1  public class Externa {
2    private int atributo = 0;
3
4    public Externa() {
5      ...
6    }
7
8    public class Interna {
9      private int atributo=0;
10
11     public Interna() {
12       this.atributo = 10;
13       Externa.this.atributo = 10;
14     }
15   }
16 }
```

Classes Internas Anônimas

- É possível criar um objeto estendendo uma classe ou implementando uma interface sem ser necessário dar um nome para a nova classe criada

Classes Internas Anônimas

- É possível criar um objeto estendendo uma classe ou implementando uma interface sem ser necessário dar um nome para a nova classe criada
- Para isso existe as **Classes Anônimas**

Classes Internas Anônimas

```
1  public class ThreadFactory {
2
3    public Thread getThread() {
4      Thread t = new Thread() {
5
6        @Override
7        public void run() {
8          ...
9        }
10
11     };
12
13     return t;
14   }
15
16 }
```

Classes Internas

- As classe internas serão mais aprofundadas na aula quando interfaces gráficas forem apresentadas

Classes Internas

- As classe internas serão mais aprofundadas na aula quando interfaces gráficas forem apresentadas
- Mais informações sobre classes internas, consulte Core Java 2 Volume I : Fundamentos, pág. 213-230

Resumo

	Objetos	**Herança**	**Métodos**	**Atributos**
Interface	Não pode ter instâncias	Pode ser implementada (**implements**)	Somente assinatura dos métodos	Somente constantes
Classe Abstrata	Não pode ter instâncias	Pode ser estendida (**extends**)	Métodos concretos e abstratos	Constantes e atributos
Classe Final	Pode ter instâncias	Não pode ser estendida	Somente métodos concretos	Constantes e atributos