Capítulo 7

RTTI e Interfaces

No capítulo anterior, você viu que a operação de upcast é segura e fundamental para o uso de polimorfismo em orientação a objetos e que a palavra reservada *abstract* permite que se criem classes e métodos abstratos. Classes abstratas são classes que não podem ser instanciadas, isto é, que não podem dar origem a objetos. Os membros destas classes serão sempre instâncias das suas subclasses concretas. Métodos abstratos são declarados em uma classe sem que possuam implementação.

Neste capítulo, veremos como o mecanismo de RTTI ("RunTime Type Identification") permite a realização segura da operação de **downcasting** (inversa ao upcast). Este tipo de operação é útil em algumas aplicações e pode possibilitar a construção de código reusável. Veremos também o conceito de interfaces, usado em Java para permitir um tipo restrito de herança múltipla. Nas classes abstratas você fornece parte da interface sem se preocupar com a sua implementação, que será dada pelas classes herdeiras. A palavra reservada *interface* é usada em Java para produzirmos uma classe abstrata na qual todos os seus métodos são abstratos. Chamamos estas classes de puramente abstratas. Com interfaces um objeto de uma classe herdeira pode ser convertido através de upcast em várias classes bases distintas.

7.1 – RTTI

Inicialmente, ao estudarmos herança e polimorfismo, assumimos que apenas métodos estabelecidos na classe base devem ser sobrescritos na classe derivada. Tal procedimento reflete um uso puro da relação **é-um**, visto que a interface de uma classe estabelece o que ela é. A herança garante que qualquer classe derivada terá a mesma interface da classe base e nada mais. Objetos das classes derivadas podem ser substituídos perfeitamente pelos objetos da classe base e nunca se necessita de informação extra a respeito das subclasses quando você as está utilizando. Em outras palavras, tudo que se precisa fazer é um upcast a partir da classe derivada. A partir daí, você não precisa saber exatamente qual o tipo do objeto que está sendo manipulado. Basta deixar o polimorfismo lidar com os objetos das diferentes classes.

Contudo, logo se observa que a extensão das classes derivadas através da inclusão de novos métodos específicos é muito útil em situações onde que você deseja que os objetos da classe derivada sejam apenas parecidos com os da classe base.

Ao mesmo tempo que esta abordagem tem grande utilidade, ela apresenta uma dificuldade. Uma vez que é realizado um upcast, você não pode mais chamar os métodos específicos do objeto porque eles não fazem parte da interface da classe base. Para contornar esta dificuldade, é necessário redescobrir o tipo original do objeto de modo que você possa acessar seus métodos específicos. Isto pode ser feito através da operação de **downcast**. Tal operação permite que se mova para baixo na hierarquia de classes.

A operação de upcast é sempre segura porque a classe base não pode possuir uma interface maior do que a classe derivada. Porém, no caso da operação de downcast, é exatamente isto que ocorre. Como resultado, o sistema de tipos da linguagem pode ser violado. Para resolver este problema, deve haver um modo de garantir que a operação de downcast é correta, para que não se converta acidentalmente o objeto para um tipo errado e lhe seja enviada uma mensagem que ele não pode aceitar.

Em algumas linguagens (como C++) você deve realizar uma operação especial para conseguir um downcast seguro, mas em Java isso sempre ocorre. Assim, quando você realiza uma operação de cast em Java, esta operação é verificada em tempo de execução para confirmar se o tipo é apropriado. Se não é, será disparada uma exceção `ClassCastException`. Este ato de checagem dinâmica de tipos é conhecido como o mecanismo de RTTI. O exemplo seguinte demonstra como ele funciona:

```
// RTTI.java
// Downcasting & RTTI

class Util {
  public void f() {}
  public void g() {}
}

class MuitoUtil extends Util {
  public void f() {}
  public void g() {}
  public void u() {}
  public void v() {}
  public void w() {}
}

public class RTTI {
  public static void main(String[] args) {
    Util[] x = {
      new Util(),
      new MuitoUtil()
    };
    x[0].f();
    x[1].g();
    // Erro de compilacao: nao e' metodo de Util
    //! x[1].u();
    ((MuitoUtil)x[1]).u(); // Downcast/RTTI
    ((MuitoUtil)x[0]).u(); // Erro de execucao
  }
} ///:~
```

Exemplo 7. 1 - Downcast e RTTI

A classe `MuitoUtil` estende a interface de Util. Como ela herda de `Util`, objetos dessa classe podem ser convertidos via upcast para `Util`. Você pode ver isto ocorrendo na inicialização do vetor `x` em `main()`. Desde que os dois objetos do vetor são da classe `Util`, é possível enviar as mensagens `f()` e `g()` para ambos. Contudo, caso se tente enviar a mensagem `u()` (a qual existe apenas em `MuitoUtil`), ocorrerá um erro de compilação.

Para acessar a interface estendida de um objeto da classe `MuitoUtil`, você deve tentar realizar downcast. Se o tipo convertido é correto, a operação será bem sucedida. Caso contrário, será sinalizada a exceção `ClassCastException`. Não há necessidade de se

escrever código para tratar esta exceção porque ela é especial, indicando um erro de programação que pode ocorrer em qualquer parte do programa.

Através do mecanismo de RTTI também é possível saber qual o tipo que você está lidando antes de tentar fazer o downcast. Para tanto, você pode usar o operador *instanceof* de Java para verificar se um objeto é uma instância de um tipo particular. O resultado deste operador é de valor booleano. Assim, você pode usá-lo em expressões como a seguinte:

```
if(x instanceof Cachorro)
  ((Cachorro)x).late();
```

O `if` acima verifica se o objeto `x` pertence a classe `Cachorro` antes de fazer o downcast de `x` para `Cachorro`. Desta maneira, pode-se evitar a tentativa de conversão de um objeto de outra classe em um `Cachorro`, o que provocaria uma exceção. O exemplo seguinte ilustra o uso de *instanceof*:

```
// GatosECaes.java
// Exemplo Simples de RTTI

class Gato {
  private int numeroGato;
  Gato(int i) {
    numeroGato = i;
  }
  void miar() {
    System.out.println("Gato #" +
            numeroGato + " miou!");
  }
  void brincar() {
    System.out.println ("Novelo");
  }
}

class Cao {
  private int numeroCao;
  Cao(int i) {
    numeroCao = i;
  }
  void latir() {
    System.out.println("Cao  #" +
            numeroCao + " latiu!");
  }
  void brincar() {
    System.out.println ("Osso");
  }
}

public class GatosECaes {
  public static void main(String[] args) {
    Object [] Domesticos = new Object [10];
    for(int i = 0; i < 10; i = i + 2)
      Domesticos[i] = new Gato(i);
    for(int i = 1; i < 10; i = i + 2)
```

```
      Domesticos[i] = new Cao(i);
    for(int i = 0; i < Domesticos.length; i++) {
      if (Domesticos[i] instanceof Gato)
        ((Gato)Domesticos[i]).miar();
      if (Domesticos[i] instanceof Cao)
        ((Cao)Domesticos[i]).latir();
    }
    //! ((Cao) Domesticos[0]).brincar();
  }
}
```

Exemplo 7. 2 - O operador *instanceof*

No exemplo acima, você pode observar o uso do operador *instanceof* para identificar se o objeto em `Domesticos[i]` é um `Gato` ou um `Cao` antes de fazer o downcast.

Você pode estar se perguntando quando deverá usar a operação de downcast. Um dos usos desta operação ocorre quando é necessário criar estruturas de dados totalmente genéricas. Neste caso, os elementos da estrutura serão da classe Object. Ao retirarmos um elemento desta estrutura será necessário fazer downcast para a classe original do elemento de modo a poder utilizar as suas operações específicas. Em Java, este tipo de downcast é amplamente utilizado nas classes que oferecem coleções e iteradores (elas são estruturas totalmente genéricas). Outro tipo de uso ocorre quando se deseja diferenciar dinamicamente o comportamento de objetos de uma certa categoria. Por exemplo, num programa editor de formas geométricas, pode-se desejar realizar uma operação de "highlight" nos objetos de uma forma geométrica (círculo, triângulo ou quadrado). Neste caso, é necessário identificar dinamicamente os objetos do tipo desejado para realizar esta operação.

7.2 – Interfaces

A palavra reservada *interface* estende ainda mais o conceito de classe abstrata. Você poderá pensar um uma interface como uma classe abstrata pura. O uso de interfaces lhe permitirá estabelecer o formato de uma classe: nome dos métodos, lista de argumentos e tipos de retorno, mas sem que estes métodos tenham sua implementação definida. Uma interface também pode conter atributos, mas estes são implicitamente *static* e *final*.

Note que uma interface especifica apenas o formato de uma classe, e não provê nenhuma implementação. É como se ela estabelecesse um protocolo (algumas linguagens orientadas a objeto possuem como palavra reservada a palavra *protocol* no lugar de *interface*) entre as classes que a implementam. Portanto, qualquer código que faz uso de uma interface em particular saberá quais métodos poderão ser chamados para aquela interface.

Para criar uma interface, usa-se a palavra chave *interface* no lugar de *class*. Como na definição de uma classe, você poderá especificar uma interface como pública através da palavra reservada *public* (desde que esta interface seja definida em um arquivo de mesmo nome) ou simplesmente não especificar o acesso e torná-la amiga ("friendly"), podendo então ser usada apenas pelas classes do mesmo pacote em que ela estiver definida.

Para criar uma classe que implementa uma interface em particular (ou um grupo de interfaces) é usada a palavra reservada *implements*. Uma vez implementada, uma interface torna-se uma classe como as outras, que pode inclusive ser estendida O diagrama de classes seguinte, elaborado para o exemplo dos instrumentos musicais, mostra esta situação:

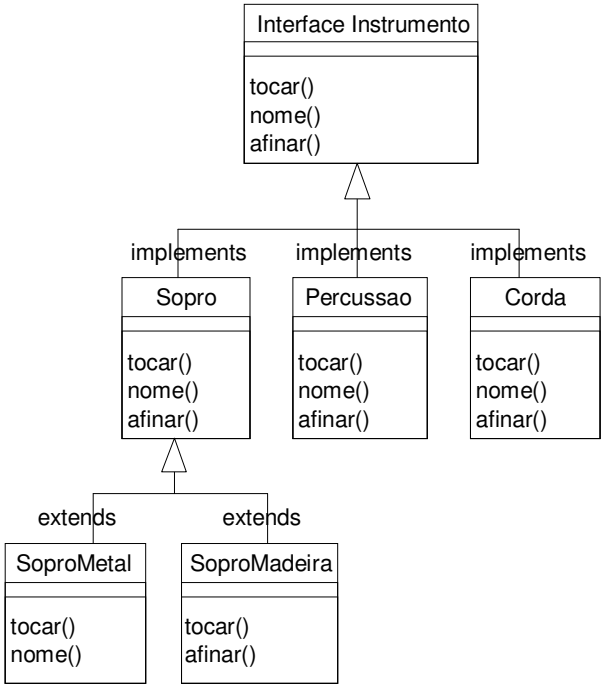

Figura 7. 1 - Diagrama de Classes e Interfaces

Os métodos de uma interface são sempre públicos. Ao se definir uma interface, você pode até explicitar este fato precedendo os métodos com a palavra *public*, porém estes métodos serão públicos mesmo se o especificador de acesso for omitido. Além disso, se usarmos outro especificador de acesso, o compilador emitirá uma mensagem de erro.

Portanto, ao definirmos uma classe que implementa uma interface, devemos especificar **explicitamente** o acesso aos métodos da interface como públicos, caso contrário estes serão dados como amigos e você estará reduzindo a acessibilidade de um método durante a herança (o que não é permitido pela linguagem), provocando um erro de compilação.

Esta situação é mostrada no exemplo seguinte, que trás a implementação de uma versão modificada do exemplo dos instrumentos musicais, referente ao diagrama anterior. Note que cada método da interface é automaticamente público, mesmo com a omissão do especificador de acesso *public*.

```
//MusicaInterface.java

interface Instrumento {
 int i = 5; // static & final
 // nao pode haver implementacao dos metodos:
 void tocar();  // automaticamente publicos
 String nome();
 void afinar();
```

```
}

class Sopro implements Instrumento {
 public void tocar() {
  System.out.println("Sopro.tocar()");
 }
 public String nome() { return "Sopro"; }
 public void afinar() {}
}

class Percussao implements Instrumento {
 public void tocar() {
  System.out.println("Percussao.tocar()");
 }
 public String nome() { return "Percussao"; }
 public void afinar() {}
}

class Corda implements Instrumento {
 public void tocar() {
  System.out.println("Corda.tocar()");
 }
 public String nome() { return "Corda"; }
 public void afinar() {}
}

class SoproMetal extends Sopro {
 public void tocar() {
  System.out.println("SoproMetal.tocar()");
 }
 public void afinar() {
  System.out.println("SoproMetal.Afinando()");
 }
}

class SoproMadeira extends Sopro {
 public void tocar() {
  System.out.println("Madeira.tocar()");
 }
 public String nome() { return "SoproMadeira"; }
}

public class MusicaInterface {
 static void melodia(Instrumento i) {
  // ...
  i.tocar();
 }
 static void sinfonia(Instrumento[] e) {
  for(int i = 0; i < e.length; i++)
  melodia(e[i]);
 }
```

```
 public static void main(String[] args) {
  Instrumento[] orquestra = new Instrumento[5];
  int i = 0;
  // Upcasting:
  orchestra[i++] = new Sopro();
  orchestra[i++] = new Percussao();
  orchestra[i++] = new Corda();
  orchestra[i++] = new SoproMetal();
  orchestra[i++] = new SoproMadeira();
  sinfonia(orquestra);
 }
}
```

Exemplo 7. 3 - Uso de Interfaces

Observe que não importa se o upcasting está sendo feito para uma classe regular chamada instrumento, uma classe abstrata chamada instrumento ou para uma interface chamada instrumento. O comportamento é idêntico.

7.2.1 – Herança Múltipla em Java

As interfaces, assim como as classes e métodos abstratos, fornecem modelos de comportamento esperados na implementação de outras classes. No entanto, as interfaces fornecem muito mais funcionalidades para o projeto orientado a objetos e, consequentemente, para a linguagem Java do que as classes e métodos abstratos.

Após uma reflexão mais profunda ou alguma experiência em um projeto orientado a objetos mais complexo, você pode descobrir que a simplicidade de uma hierarquia de classes pode ser bastante restritiva, particularmente quando se tem algum comportamento que precisa ser utilizado por classes em diferentes ramos da mesma árvore.

Algumas linguagens de programação, como C++, incluem o conceito de herança múltipla, aonde uma classe pode herdar de mais de uma superclasse e obter comportamento (métodos) e atributos de todas as suas superclasses simultaneamente. Um problema da herança múltipla é que ela eleva o grau de complexidade de uma linguagem. Questões como a chamada de métodos e a organização da hierarquia tornam-se bem mais complicadas com a herança múltipla e mais predispostas a ambigüidades e confusões. E como os projetistas da linguagem Java pretendiam tornar a linguagem simples, rejeitaram a herança múltipla em favor da herança simples, bem menos complexa. Na herança simples, uma classe somente pode herdar diretamente de uma única classe.

Então como resolver a necessidade do comportamento comum que não se encaixa na hierarquia de classes? A linguagem Java possui uma outra hierarquia completamente separada da hierarquia de classes, que possibilita esse comportamento misto. Assim, ao criarmos uma classe, esta classe possui apenas uma superclasse principal, mas pode selecionar e escolher diferentes comportamentos comuns da outra hierarquia. Essa outra hierarquia é a hierarquia de interfaces. Como foi dito anteriormente, uma interface em Java contém apenas definições abstratas e constantes de métodos e atributos.

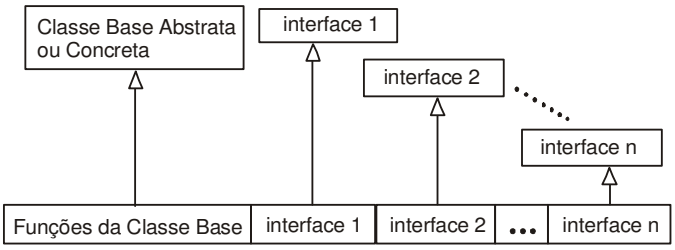

Figura 7. 2 - Herança Múltipla em Java

O exemplo seguinte mostra uma classe concreta combinada com diversas interfaces para produzir uma nova classe:

```
// Aventura.java

interface PoderLutar {
 void lutar();
}
interface PoderNadar {
 void nadar();
}
interface PoderVoar {
 void voar();
}
class PersonagemDeAcao {
 public void lutar() {}
}
class Heroi extends PersonagemDeAcao
 implements PoderLutar, PoderNadar, PoderVoar {
 public void nadar() {}
 public void voar() {}
}
public class Aventura {
 static void t(PoderLutar x) { x.lutar(); }
 static void u(PoderNadar x) { x.nadar(); }
 static void v(PoderVoar x) { x.voar(); }
 static void w(PersonagemDeAcao x) { x.lutar(); }

 public static void main(String[] args) {
  Heroi h = new Heroi();
  t(h); // eh tratado como PodeLutar
  u(h); // eh tratado como PodeNadar
  v(h); // eh tratado como PodeVoar
  w(h); // eh tratado como PersonagemDeAcao
 }
}
```

Exemplo 7. 4 - Herança Múltipla

Como você pode notar, a classe `Herói` combina a classe concreta `PersonagemDeAcao` com as interfaces `PoderLutar`, `PoderNadar`, `PoderVoar`. Note também que a assinatura do método `lutar()` é a mesma na interface `PoderLutar` e na classe `PersonagemDeAcao`. Apesar de o método `lutar()` não ser definido explicitamente pela classe `Heroi`, o mesmo foi definido na classe `PersonagemDeAcao` (da qual `Heroi` é subclasse).

Na classe `Aventura`, podemos ver quatro métodos que recebem como argumentos, elementos das várias interfaces e de uma classe concreta. Quando um objeto `Heroi` é criado, ele pode ser passado para todos estes métodos, o que significa que é realizado um upcast para cada uma dessas interfaces ou classes.

Como dito anteriormente, interfaces podem ser consideradas classes puramente abstratas. Isto nos leva a seguinte pergunta: quando usaremos interfaces e quando usaremos classes abstratas? Bem, lembre-se que uma interface lhe oferece os benefícios de uma classe abstrata e os benefícios de uma interface. Portanto, se for possível criar sua classe base sem nenhuma definição de métodos ou atributos é indicado o uso de interface no lugar de uma classe abstrata. De fato, se você precisar de uma classe base, sua primeira escolha deve ser sempre por usar uma interface, e caso você precise definir métodos ou atributos, então a escolha seguinte deve ser por uma classe abstrata e, por fim, se necessário, uma classe concreta.

7.2.2 – Combinação de Interfaces e a Colisão de Nomes

Você poderá encontrar uma pequena armadilha ao implementar interfaces múltiplas. No exemplo acima, a interface `PoderLutar` e e a classe `PersonagemDeAcao` possuem um método idêntico: `void voar()`. Isto não chega a ser um problema pois o método é idêntico em ambas as classes. Mas, e se não fosse? Observe o exemplo abaixo:

```
// ColisaoInterface.java
interface I1 { void f(); }
interface I2 { int f(int i); }
interface I3 { int f(); }
class C { public int f() { return 1; } }
class C2 implements I1, I2 {
  public void f() {}
  public int f(int i) { return 10; } // sobrecarga
}
class C3 extends C implements I2 {
  public int f(int i) { return 100; } // sobrecarga
}
class C4 extends C implements I3 {
  public int f() { return 1000; } // sobrescrita
}
// Problemas:
//! class C5 extends C implements I1 {}
//! interface I4 extends I1, I3 {}
```

Exemplo 7. 5 - Colisão de Nomes

Esta dificuldade ocorre porque a combinação de herança e implementação de interface produz funções sobrecarregadas que se diferem apenas pelo tipo de retorno, o que não é permitido em Java. Caso as duas últimas linhas do código do exemplo anterior não estivessem comentadas, qualquer tentativa de compilação produziria a seguinte mensagem de erro:

```
ColisaoInterface.java:23: f() in C cannot
implement f() in I1; attempting to use
incompatible return type
found : int
required: void
ColisaoInterface.java:24: interfaces I3 and I1 are
incompatible; both define f(), but with different
return type
```

O uso de um mesmo nome para métodos em diferentes interfaces que serão combinadas futuramente geralmente causa confusão na legibilidade do código. Portanto, sempre que for possível, tente evitar isto.

7.2.3 – Estendendo uma Interface com Herança

Você pode facilmente adicionar novas declarações de métodos a uma interface usando herança. Usando herança podemos ainda combinar diversas interfaces em uma nova interface. Em ambos os casos você produzirá uma nova interface, como pode ser visto no exemplo abaixo:

```
// ShowDeHorror.java
interface Monstro {
 void ameaçar();
}

interface MonstroPerigoso extends Monstro {
 void destruir();
}

interface Letal {
 void matar();
}

class DragonZilla implements MonstroPerigoso {
 public void ameaçar() {}
 public void destruir() {}
}

interface Vampiro
 extends MonstroPerigoso, Letal {
 void beberSangue();
}
```

```
class ShowDeHorror {
 static void u(Monstro b) { b.ameaçar(); }
 static void v(MonstroPerigoso d) {
  d.ameaçar();
  d.destruir();
 }

 public static void main(String[] args) {
  DragonZilla if2 = new DragonZilla();
  u(if2);
  v(if2);
 }
}
```

Exemplo 7. 6 - Estendendo uma Interface com Herança

`MonstroPerigoso` é uma extensão simples de `Monstro` que fornece uma nova interface, que é implementada em `DragonZilla`. A sintaxe usada em `Vampiro` só é válida ao se aplicar herança à interfaces. Normalmente você só pode usar a palavra *extends* com uma única classe. Mas como uma interface pode ser formada pela combinação de várias interfaces, *extends* pode referenciar múltiplas interfaces na construção de uma nova interface. Como você pôde ver, neste caso, os nomes destas interfaces são separados por vírgula.

7.2.4 – Inicializando os Atributos de Uma Interface

Os atributos definidos em uma interface são automaticamente **static** e **final**. Os mesmos não podem deixar de ser inicializados na declaração, mas podem ser iniciados com expressões não constantes. Por exemplo:

```
// ValoresAleatorios.java
import java.util.*;
public interface ValoresAleatorios {
 int rint = (int)(Math.random() * 10);
 long rlong = (long)(Math.random() * 10);
 float rfloat = (float)(Math.random() * 10);
 double rdouble = Math.random() * 10;
}
```

Exemplo 7. 7 - Iniciando Atributos de Interfaces

Uma vez que estes atributo são *static*, eles são inicializados na primeira vez em que a classe é carregada, o que acontece quando um destes atributos é acessado pela primeira vez.

7.2.5 – A Interface Comparable

A linguagem Java possibilita a comparação de objetos de duas maneiras. A primeira é através do método `equals()`, presente na classe `Object`, e que pode ser redefinido em qualquer classe (lembre-se que em Java, todas as classes são herdeiras da classe `Object`). Note que o método `equals()` retorna apenas um valor booleano informando se o objeto passado como parâmetro é igual ao objeto corrente, e não trás nenhuma informação comparativa entre os dois objetos envolvidos. A segunda maneira, é através do

método `compareTo()`, que pode ser incorporado às classes pela implementação da interface `java.lang.Comparable`. Este método recebe um outro objeto como argumento e retorna um valor negativo caso o objeto passado como argumento seja menor que o objeto corrente, zero caso estes objetos tenham o mesmo valor e um número positivo caso o argumento seja maior que o objeto corrente.

O exemplo abaixo mostra uma classe que implementa a interface `Comparable` e ilustra esta comparação através do método `Arrays.sort()`, presente na biblioteca padrão da linguagem Java:

```
// TipoComp.java
// Implementa a Interface Comparable
import java.util.*;
public class TipoComp implements Comparable {
  int i;
  int j;

  public TipoComp(int n1, int n2) {
    i = n1;
    j = n2;
  }
  // sobrecarga do método toString:
  public String toString() {
    return "[i = " + i + ", j = " + j + "]";
  }
  // implementacao do metodo compareTo
  public int compareTo(Object rv) {
    int rvi = ((TipoComp)rv).i;
    return (i < rvi ? -1 : (i == rvi ? 0 : 1));
  }
  public static void main(String[] args) {
    TipoComp[] a = new TipoComp[6];
    // preenche o vetor com objetos TipoComp:
    a[0] = new TipoComp(5,1);
    a[1] = new TipoComp(3,2);
    a[2] = new TipoComp(9,4);
    a[3] = new TipoComp(7,1);
    a[4] = new TipoComp(11,5);
    a[5] = new TipoComp(8,3);
    Arrays.sort(a);
    for (int n=0; n<=5; n++){
      // imprime o vetor depois de ordenado
      System.out.println("a["+n+"]:"+a[n]+" ");
    }
  }
}
```

Exemplo 7. 8 - A Interface Comparable

Ao definir uma a função de comparação, você se responsabiliza pelo do critério de comparação que será empregado quando os objetos de sua classe forem comparados. No exemplo acima, apenas o valor do atributo i é levado em conta na comparação dos objetos e o valor de j é ignorado.

7.3 – Exercícios

1. Continue o primeiro exercício do capítulo 6 (problema do Exército), percorrendo agora o vetor já preenchido de membros do Exército e contando o número de militares da categoria Major. Considere que você não tem como saber de antemão como o vetor se encontra preenchido.
2. Continue o terceiro exercício do capítulo 6 (problema dos móveis), agora incluindo um novo tipo de móvel (Sofá-Cama) no vetor de móveis. Mova, agora, todos os sofás para um vetor de sofás e todas as camas para um vetor de camas. Lembre-se que os sofá-cama deverão participar dos dois vetores (sofás e camas).