www.sc.senai.br | 0800 48 1212

Interfaces / Polimorfismo

Aula 11 – 23/05

Interfaces

O que é uma interface no SO Windows?

A interface de um SO representa o que pode ser visto e a forma como podemos interagir com o SO, por meio de botões, menus, ícones outros.

Interfaces

Para que serve a interface USB?

Por meio de um padrão estabelecido diversos dispositivos podem ser conectados via USB como uma forma de conectar e expandir a capacidade de um computador. De um lado, temos a indústria que definiu o padrão e o publicou; de outro temos os fabricantes que estudaram os padrões e o desenvolveram dispositivos que pode se comunicar via USB..

Interfaces em Java

A modelagem com interfaces provê uma grande flexibilidade para os sistemas porque por meio do seu uso podemos separar totalmente a especificação da implementações. Com isso podemos ter soluções que podem facilmente trabalhar com implementações diferentes da mesma interface.

Interfaces em Java

A interface é uma forma de especificação de comportamento de classes, onde definimos todos os métodos que devem ser implementados pela classes, garantindo que as classes que implementem a interface terão obrigatoriamente todos os métodos na interface.

Interfaces em Java

Interfaces define a forma como iremos interagir com as classes que a implementam.

Podemos falar que são protótipos de classes.

Podemos falar que são 100% abstratas.

Definindo uma Interface

Métodos:

Uma interface pode conter apenas definições de métodos (métodos abstratos), não podem conter nenhum implementado. No entanto, não é necessário utilizar o modificador **abstract**, visto que, todos os métodos são abstratos por padrão.

Todos os métodos de uma interface são públicos e diferentemente das classes, quando não utilizamos nenhum modificador de acesso, são automaticamente definidos com públicos.

Definindo uma Interface

atributos:

Uma interface pode conter apenas atributos públicos e explicitamente inicializados, não havendo nenhum restrição para a utilização do modificador **static** ou **final**.

Todos os atributos de uma interface são **final** (constantes) e **static** por padrão, mesmo quando não explicitamente declarados.

Declarando uma Interface

Uma interface deve ser declarada utilizando a palavra reservada **interface**, ao invés de **class**.

Exemplo:

```
public interface Trasportavel {
   String UNIDADE = “Kg”;
   public double getVolume();
}
```

Declarando uma Interface

Interfaces devem ser definidas em arquivos com o mesmo nome da interface declarada com a extensão .java.

Serão compiladas para geração de arquivo .class identicamente “as classes convencionais.

Obs. assim como não podemos instanciar uma classe abstract não podemos instanciar interfaces.

Abstração

A abstração (análise focada) também é importante ao projetar uma interface, Uma prática é utilizar, quando possível, um adjetivo como nome de uma interface.

Exemplos: Transportavel,Tributavel, Rastreavel, Perecivel.

Implementado uma Interface

Para implementarmos uma interface em uma classe, utilizamos a instrução **implements** e por isso precisamos obrigatoriamente implementar todos os métodos definidos pela interface, respeitando a assinatura dos métodos.

Exemplo a Classe Mobilia implementa a interface Transportavel.

Implementado uma Interface

Vamos parar e pensar, muitas outras classe podem implementar está interface, pois existem muitas coisas transportáveis que podemos modelar, por exemplo equipamentos eletrônicos pessoas, roupas, etc.

Importante: Diferente da herança que podemos **herdar** de **uma** classe só podemos **implementar inúmeras interfaces** em uma classe.

Implementado uma Interface

Vamos imaginar uma classe AlimentoBase está classe deverá ser adaptável as interfaces Transportavel e Perecivel. A subclasse Alimento será empregada com um adaptador, para assumir as características das interfaces.

Ver os exemplos de AlimentoBase, Alimento, Transportavel e Perecivel.

Implementado uma Interface

Uma classe abstrata pode implementar uma interface sem implementar todos os seus métodos, pois como vimos anteriormente uma classe abstrata pode conter métodos abstratos. Os métodos não implementados serão considerados como métodos abstratos. Qualquer classe concreta (não abstrata) que estenda a classe abstrata terá que implementar todos os métodos pendentes.

Estendendo uma Interface

Uma interface pode ser estendida por outras, fazendo com que a interface filha herde todos os métodos definidos pela super interface, ou seja a classe que implementar a interface filha terão que implementar todos os métodos definidos nas duas interfaces.

Utilizamos herança quando queremos especifica uma nova categoria de classes, as quais são um subconjunto das classe definidas pela super-interface.

Estendendo uma Interface

No nosso exemplo, focado em um sistema de logística criaremos a interface Inflamavel, a partir de Transportavel. Desta forma garantimos que todas as classes que implementarem a interface Inflamavel obrigatoriamente irão implementar Transportavel definida como super-interface.

Estendendo uma Interface

A classe galão de combustível irá implementar a interface Inflamavel.

Ver exemplo:

Interface x abstract

É uma boa prática de modelagem de classe trabalharmos sempre no sentido mais abstrato para mais concreto, do genérico para o especifico e assim por diante. Para melhor organizarmos isso, existem os seguintes recursos:

- ✓ Interface tudo é abstrato, separação total de definição e implementação.
- ✓ Classe abstract parte concreta para abstrata.
- ✓ Classes concreta tudo concreto.
- ✓ Classes final definição completa de um objeto.

Interface erros comuns

- ✓ métodos não implementados.
- ✓ modificador com menor acessibilidade.
- ✓ instanciação de uma interface.
- ✓ Terminar a definição do método com {} ao invés de “;”.

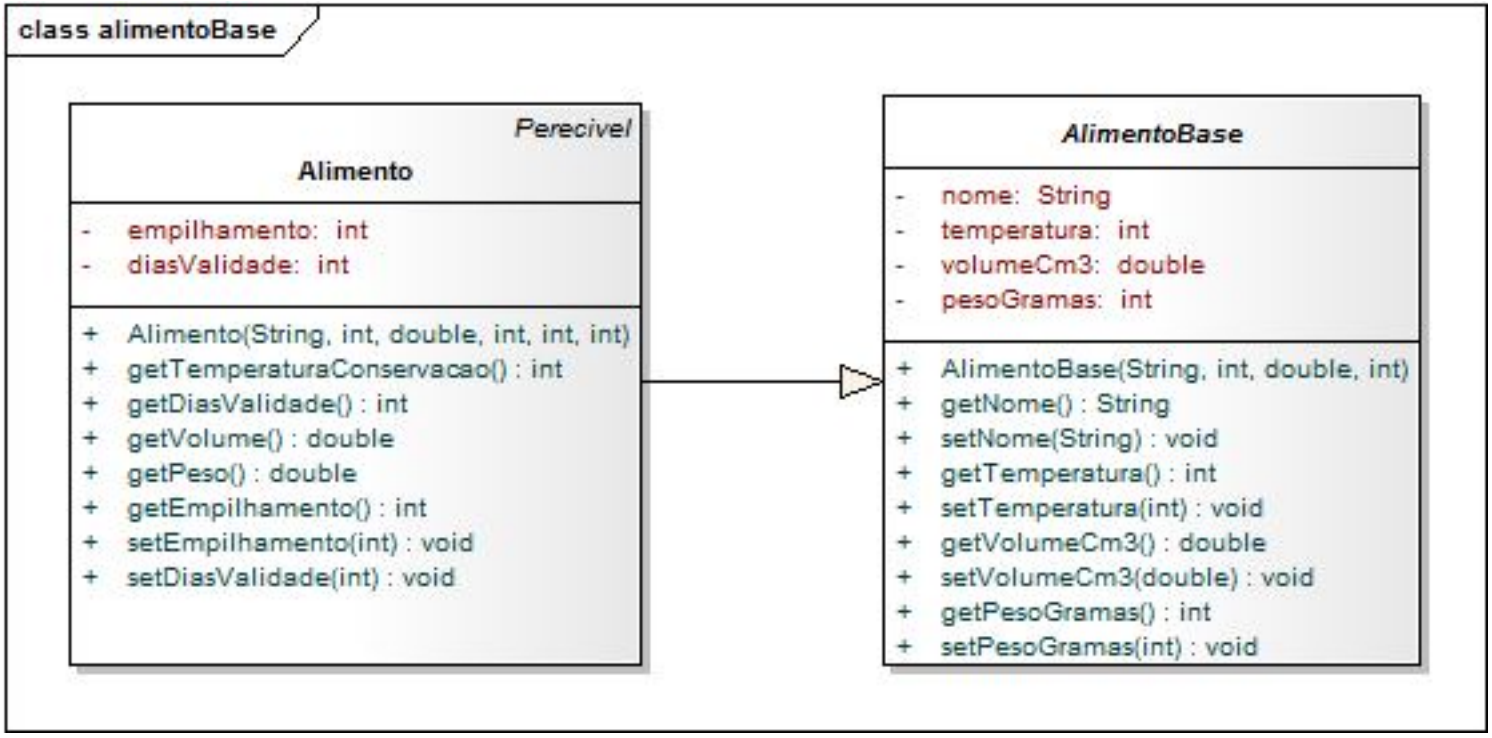
Interface x abstract

É uma boa prática de modelagem de classe trabalharmos sempre no sentido mais abstrato para mais concreto, do genérico para o especifico e assim por diante. Para melhor organizarmos isso, existem os seguintes recursos:

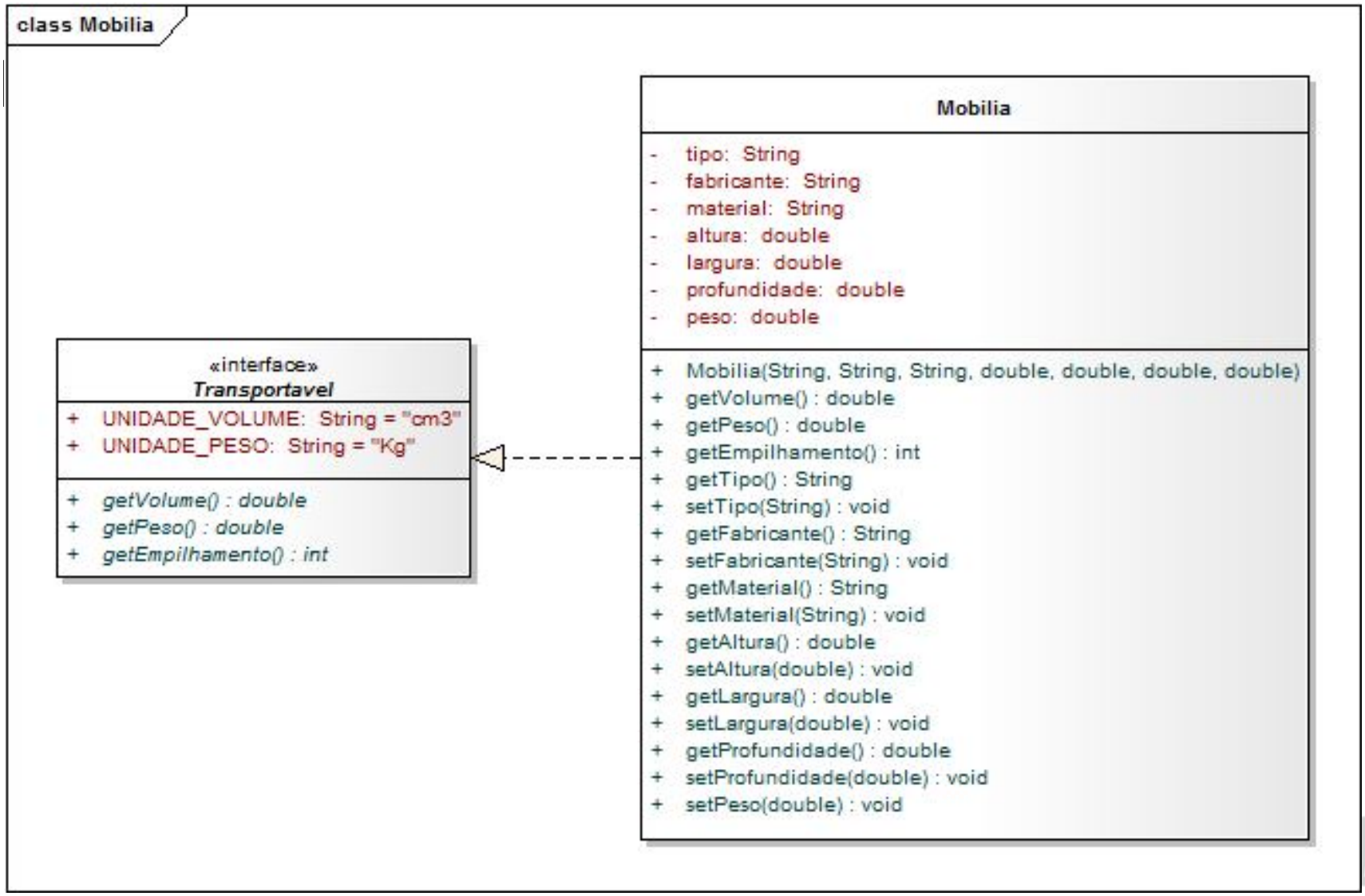
- ✓ Interface tudo é abstrato, separação total de definição e implementação.
- ✓ Classe abstract parte concreta para abstrata.
- ✓ Classes concreta tudo concreto.
- ✓ Classes final definição completa de um objeto.

Interface e UML

Representação UML:

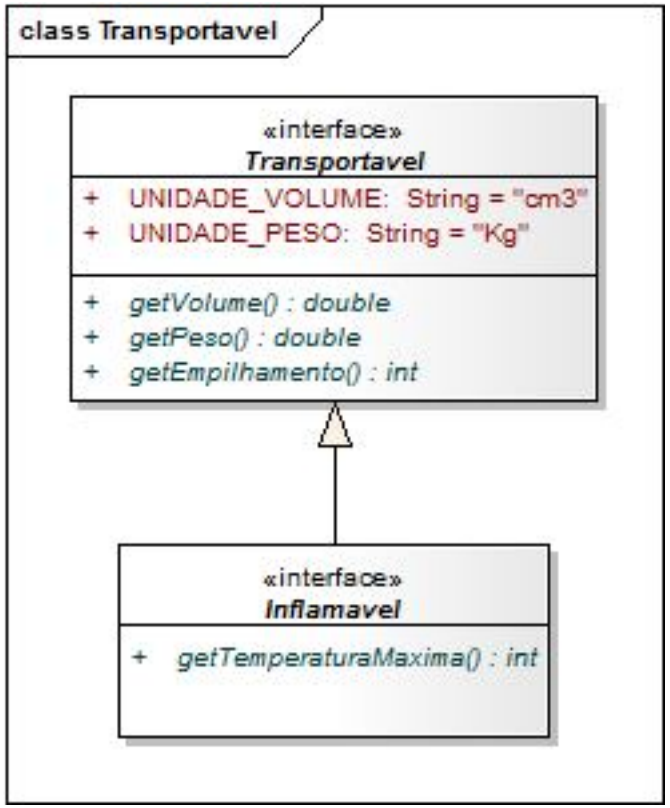


Interface e UML



Interface e UML

Representação UML:



Laboratório

Praticar a utilização de interface.

Cast de objetos e polimorfismo

Em algumas condições é necessário mudar a forma de operar e visualizar um objeto. Nestas situações empregamos as operações de cast para trabalhar com o objeto utilizando parte ou todos seus métodos e atributos.

Cast de tipos primitivos

Relembrando cast com tipos primitivos:

```
short s = 4096;
```

//cast up, copiamos os 16 bits da variavel para os 32 bits de um float.

```
float f = s;
```

int i = 123; //32 bits

//cast down copiamos os 8 bits menos significativos de i para os 8 bits de um byte.

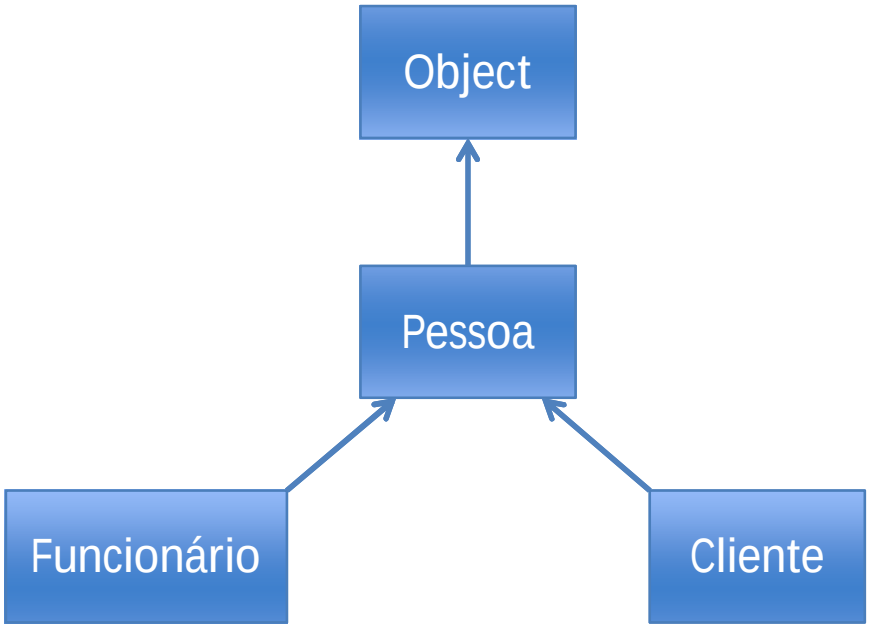
```
byte b = (byte) i;
```

Cast de objetos e polimorfismo

A operação de cast entre objetos é semelhante à operação de cast com primitivos, mas com uma diferença profunda: os objetos por trás das variáveis não são copiados ou truncados. No máximo as características e funcionalidades estarão apenas ocultas, podendo ser retribuídas posteriormente.

Cast de objetos e polimorfismo

Vamos adotar a seguinte estrutura para podermos explicar isso:



Cast de objetos e polimorfismo

É possível fazer operação de cast entre objetos desde que estejam em uma mesma hierarquia; não podemos fazer cast entre classes irmãs, tal como Funcionario e Cliente.

Cast up (Widening)

Com base na hierarquia, em havendo uma variável declarada e instanciada, conclui-se que:

Cliente é uma **Pessoa**, e toda **Pessoa** é um **Object**.

Ver exemplo de TesteCastUp.java

Cast up (Widening)

Ao visualizarmos um **Cliente** como uma **Pessoa** perdemos a capacidade de manipular os métodos getCpf e setCpf, mas podemos ainda trabalhar com os gets e sets para os atributos nome e rg. Apesar da variável de manipulação do tipo **Pessoa** o objeto continua sendo um **Cliente**, que não perdeu seu cpf. Ao visualizarmos uma **Pessoa** como um **Object** seremos capazes de manipular apenas os membros (atributos e métodos) definidos na classe **Object**, como toString().

Cast up (Narrowing)

A operação de cast down (narrowing) é a oposta à operação cast up (widening), isto é ao invés de generalizarmos um objeto vamos especializa-lo.

A generalização é uma operação mais previsível do que a especialização, porque à analise e hierarquia de classes permite saber se a operação é ou não possível, na especialização, ao contrário, a operação irá depender da forma como o objeto foi criado.

Cast up (Narrowing)

Vejamos um exemplo, se o objeto é criado e declarado como **Cliente**, e sobre um cast up para **Pessoa** devendo-se a forma como foi criado um **Cliente**. No entanto, se o objeto é criado e declarado com **Pessoa**, não é possível fazer o cast down para transforma-lo em **Cliente**.

Obs: todo o Cliente é uma Pessoa, mas nem toda a Pessoa é um cliente, assim a operação de cast down deverá ser feita sempre de forma explicita.

Cast up (Narrowing)

Ver os exemplo:

TesteCastDown.java

Operador instanceof

Para evitar o problema **ClassCastException** no exemplo anterior podemos utilizar antes da operação de cast down o operador **instanceof**, para verificar se o objeto referenciado por uma variável é compatível com uma determinada classe ou interface.

TesteCastDown.java

Polimorfismo

Polimorfismo é a palavra de origem grega que significa muitas formas. É um poderoso recurso da orientação a objetos que é utilizado das seguintes formas:

- ✓ Definimos um tipo base (classe ou interface) e criamos classes derivadas, por herança ou por implementação de interface e assim temos várias formas para um tipo base;
- ✓ Utilizamos uma declaração de variável de um tipo-base para manipular (via cast up) um objeto de qualquer uma de seus tipos derivado.

Polimorfismo

Onde uma superclasse é esperada podemos utilizar uma instância de uma subclasse. Onde uma interface é esperada podemos utilizar uma instancia de uma classe implementadora.

Benefícios: Existem duas formas de se beneficiar do polimorfismo na POO.

- ✓ Parâmetros e retornos polimórficos;
- ✓ Coleções heterogêneas;

Parâmetros polimórficos

Considerando a hierarquia de classe: Pessoa, Cliente e Funcionario; onde uma Pessoa é esperada podemos utilizar um cliente ou um Funcionario.

Ver exemplo:

RelatorioPessoas.java

TesteParametrosPolimorficos.java

Parâmetros polimórficos

O método **imprime(Pessoa p)** da classe **RelatorioPessoas** recebe um parâmetros do tipo **Pessoa**, que poderá ser um objeto de qualquer classe de **Pessoa**. (**Cliente, Funcionario**). Por isso esse array é uma coleção heterogênea.

Parâmetros polimórficos

O método **imprime(Pessoa p)** da classe **RelatorioPessoas** recebe um parâmetros do tipo **Pessoa**, que poderá ser um objeto de qualquer classe de **Pessoa**. (**Cliente, Funcionario**). Por isso esse array é uma coleção heterogênea.

Laboratório

Praticar a utilização de Polimorfismo.

Método equals() e hashCode()

equals()

O que aconteceu quando comparamos dois objetos da mesma classe utilizando o comparador == ?

Inicialmente, acreditamos que os atributos que compõem cada um dos objetos seriam comparados, retornando **true** se todos forem idênticos, mas não é o que a JVM faz, ela verifica se os **endereço de memoria são os mesmos**.

Método equals()

Para testar a equivalência de dois objetos levando em conta os valores de seus atributos (seus conteúdos) devemos utilizar o método **equals**. A classe **Object**, define o método **equals** e podemos sobrescrevê-lo para o critério de equivalência entre os objetos da nossas classes.

Método equals()

Ao sobrescrever o método **equals** temos que levar em consideração as seguintes relações:

- ✓ **reflexão**: x.equals(x) deve ser true para qualquer x diferente de null;
- ✓ **simetria**: para x e y diferente de null, se x.equals(y) é true. y.equals(x) também deve ser true;

Método equals()

- ✓ **transitividade**: para x,y,z diferentes de null, se x.equals(y) é true, e y.equals(z) é true então x.equals(z) também deve ser true.
- ✓ **consistência**: para x e y diferentes de null, multiplas chamadas de x.equals(y) devem sempre retornar o mesmo valor;
- ✓ Para x diferente de null x.equals(null) deve sempre retornar false;

Método equals()

- ✓ **transitividade**: para x,y,z diferentes de null, se x.equals(y) é true, e y.equals(z) é true então x.equals(z) também deve ser true.
- ✓ **consistência**: para x e y diferentes de null, multiplas chamadas de x.equals(y) devem sempre retornar o mesmo valor;
- ✓ Para x diferente de null x.equals(null) deve sempre retornar false;

Método hashCode()

O método **hashCode** é importante e deve ser sobrescrito para trabalharmos com coleções Java de alto desempenho do tipo Hashtable. Este método deve retornar um número inteiro (hash) calculando à partir dos atributos considerados para o método equals.

Método hashCode()

O que acontece quando o hashCode não é implementado?

Neste casso herdamos o método na clase Object, que não é capaz de calcular o código hash como uma função de atributos considerados no método equals. O problema surge ao armazenar objetos em coleções de alto desempenho do Java, operações de localização de objetos podem não funcionar, pois dependem do código hash.

Método hashCode()

As implementações de **equals** e **hashCode** devem ser coerentes, ou seja, sempre que dois objetos forem considrados iguais pelo método **equals**, devem possuir o mesmo **hashCode**. Embora não sejam obrigatório é recomendado implementar os métodos **equals** e **hashCode** de forma que objetos diferentes tenham códigos **hash** diferentes.

Método hashCode()

E o que acontece quando o hashCode não é coerente com equals?

Também vamos sofrer efeitos negativos ao armazenar objetos em coleções de alto desempenho do Java. Dois problemas podem acontecer neste caso.

- ✓ Se não considerarmos todos atributos envolvidos no métodos **equals**, vamos degenerar o desempenho das operações de localização.
- ✓ Se considerarmos tributos que não foram envolvidos no método **equals**, as operações de localização podem não funcionar.

Método hashCode()

A implementação de hashCode de uma classe pode ser baseada na combinação:

- ✓ de valores dos atributos primitivos
- ✓ de valores de **hashCode** de seus atributos reference.