

Interfaces e Classes Abstratas

Luiz Eduardo Virgilio da Silva
ICMC, USP

Parte do material foi obtido com os professores:
José Fernando Jr. (ICM/USP)



Sumário

- Conceito de interface
- Interfaces em Java
- Atualizações no Java 8
- Classes abstratas
- Classes abstratas vs interfaces

Interfaces

- Durante a criação de software, é comum que mais de um grupo de programadores trabalhe no mesmo projeto
- É fundamental estabelecer um “contrato” entre os grupos, de forma que os programas possam se comunicar
- Não importa como a implementação será feita
 - O importante é saber a definição do contrato
 - Garante que o software desenvolvido por um grupo se comunica com o outro através deste “contrato”
- Em POO, as *interfaces* fornece esse contrato

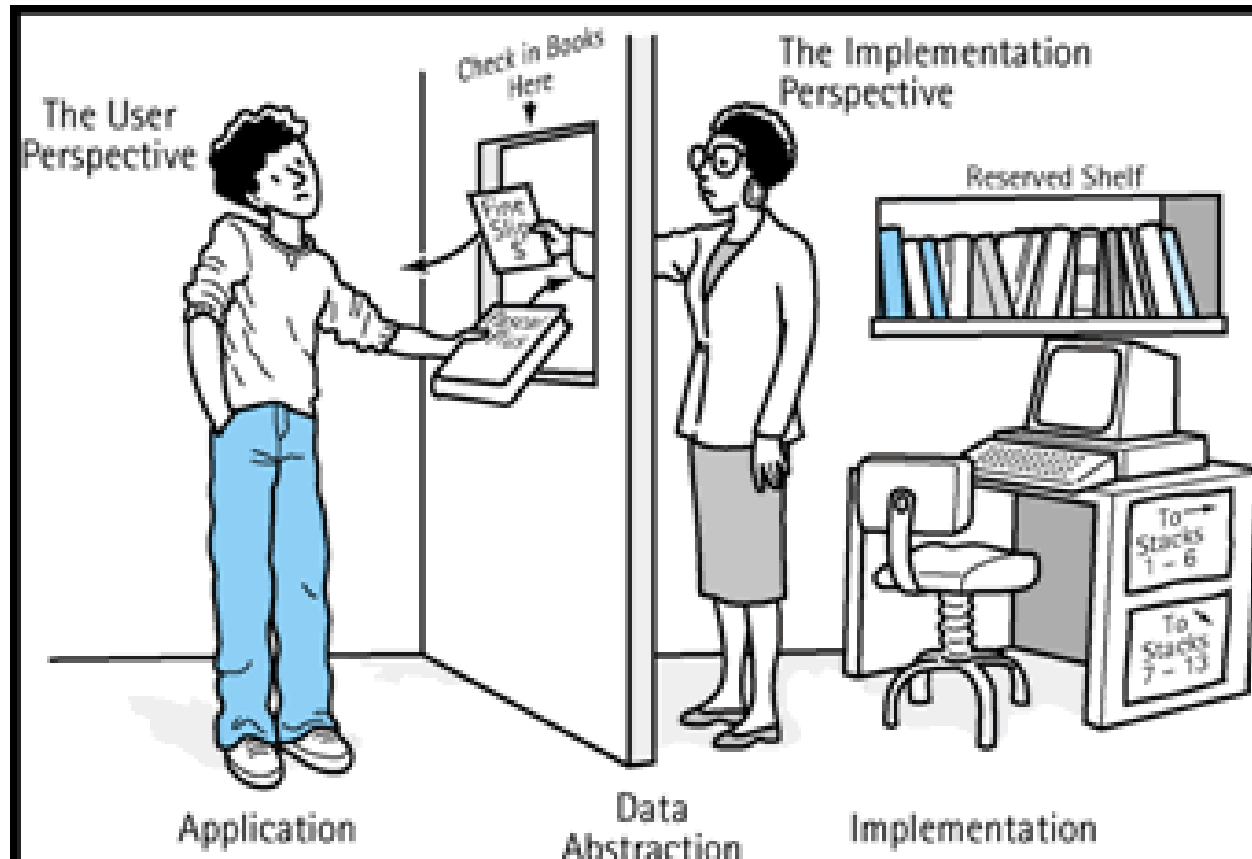
Interfaces

- Imagine que no futuro todos os carros fossem controlados por software (sem motorista)
- As indústrias de carro devem se reunir e definir um contrato, a partir do qual qualquer empresa de software pode criar um controlador de carro
 - Servirá para qualquer carro
 - Não depende dos detalhes de implementação
 - As funcionalidades dos carros podem ser aprimoradas, sem contudo ser preciso mudar os softwares controladores

Interfaces

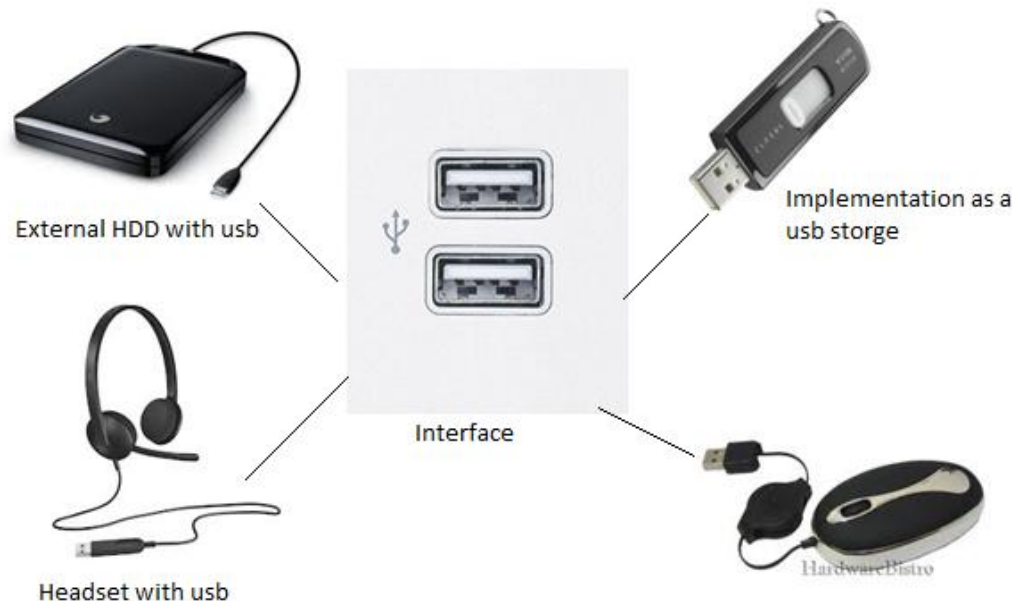


Interfaces



Interfaces

- O padrão USB é um exemplo
- Está presente em diferentes dispositivos
- As empresas que criam dispositivos que se conectam através de USB, só precisam conhecer o protocolo (mensagens trocadas) de uma conexão USB



Interfaces

- Em Java, *interfaces* são um tipo especial de referência, parecido com classes
 - NÃO podem ser instanciadas
 - Assim como classes, interfaces podem ser **public** ou *package-private*
 - Só podem conter campos constantes
 - Implicitamente são **public**, **static** e **final**
 - Os métodos são definidos apenas pela sua assinatura
 - Implicitamente são **public** e **abstract**
- Só podem ser **implementadas** por classes
- Podem ser **herdadas** por outra interface

Interfaces

- Definida em conjunto pelas montadoras de carro

```
public interface OperateCar {  
    // constant declarations, if any  
    // method signatures  
    // Direction is an enum with values LEFT and RIGHT  
  
    int turn(Direction direction, double radius,  
             double startSpeed, double endSpeed);  
    int changeLanes(Direction direction, double startSpeed,  
                   double endSpeed);  
    int signalTurn(Direction direction, boolean signalOn);  
    int getRadarFront(double distanceToCar, double speedOfCar);  
    int getRadarRear(double distanceToCar, double speedOfCar);  
  
    // more method signatures  
}
```

Interfaces

- Para usar uma interface, ela deve ser implementada
 - Todos os métodos precisam ser implementados na classe (neste caso, pela montadora BMW)

```
public class OperateBMW760i implements OperateCar {  
    // the OperateCar method signatures, with implementation –  
    // for example:  
  
    int signalTurn(Direction direction, boolean signalOn) {  
        // code to turn BMW's LEFT turn indicator lights on  
        // code to turn BMW's LEFT turn indicator lights off  
        // code to turn BMW's RIGHT turn indicator lights on  
        // code to turn BMW's RIGHT turn indicator lights off  
    }  
  
    // other members, as needed – for example, helper classes  
    // not visible to clients of the interface  
}
```

Interfaces

- No exemplo anterior, cada carro deverá implementar a interface **OperateCar**
- Chevrolet, Toyota, BMW, etc. implementarão ao seu modo esses métodos, de acordo com o carro
- Porém, a interface **OperateCar** estabelece o contrato entre as empresas de software e a montadora de carros
 - Controladores sabem quais métodos podem utilizar (interface) para controlar um carro

Definindo Interfaces

- Assim como classes, as interfaces podem ser definidas como `public` ou `package-private` (ausente)
- Uma interfaces pode estender (herdar) várias interfaces

```
public interface Interface0 extends Interface1, Interface2, Interface3
```

- Uma classe pode implementar várias interfaces
 - Supre a falta de herança múltipla

```
public class MyClass implements Interface0, Interface3
```

Usando Interfaces

- Suponha que queiramos comparar objetos
 - Verificar se um objeto é maior que outro
- Como saber, de forma genérica, que um objeto é maior que outro?
 - Depende do tipo do objeto
 - Também depende de qual atributo queremos comparar
- Qual a melhor forma de fazer isso?
 - Deixar que cada objeto implemente a maneira como deve ser comparado com outro
 - Definir uma interface que estabelece um contrato entre os objetos e quem precisa fazer a comparação

Usando Interfaces

- Objetos que desejamos comparar devem implementar a interface Relatable
 - Força a implementação dos métodos da interface (neste caso apenas um)
 - Compara o objeto atual (this) com o passado por parâmetro (other)

```
public interface Relatable {  
    // returns 1 if this is greater than other  
    // returns 0 if this is equal to other  
    // return -1 if this is less than other  
    public int isLargerThan(Relatable other);  
}
```

Usando Interfaces

```
public class Rectangle implements Relatable {  
    public int width = 0;  
    public int height = 0;  
    public Point origin;  
  
    // constructors  
  
    // methods  
    public int getArea() {  
        return width * height;  
    }  
  
    public int isLargerThan(Relatable other) {  
        Rectangle otherRect = (Rectangle) other;  
        if (this.getArea() < otherRect.getArea())  
            return -1;  
        else if (this.getArea() > otherRect.getArea())  
            return 1;  
        else  
            return 0; }  
}
```

Usando Interfaces

- No exemplo anterior, o método **isLargerThan** possui um parâmetro do tipo **Relatable**
- Ou seja, interfaces podem ser usadas como qualquer outro tipo em Java
- O tipo da interface referencia qualquer objeto que implementa aquela interface
 - Serão visíveis apenas os métodos da interface

Novos Métodos de Interfaces

- A partir do Java 8, além dos métodos abstratos (sem corpo) também é possível definir dois outros tipos de métodos em uma interface
 - default
 - static
- Esses métodos devem conter uma implementação (corpo) dentro da própria interface
- Se nenhum modificador **default** ou **static** for especificado, o método é reconhecido como **abstract** (só assinatura)

Novos Métodos de Interfaces

- Métodos default

```
public interface Relatable {  
    // returns 1 if this is greater than other  
    // returns 0 if this is equal to other  
    // return -1 if this is less than other  
    public default int isLargerThan(Relatable other) {  
        ...  
    }  
}
```

Novos Métodos de Interfaces

- A vantagem de métodos **default** surge quando uma interface precisa ser atualizada
- Considere que definimos a interface abaixo e que ela é utilizada em vários programas

```
public interface DoIt {  
    void doSomething(int i, double x);  
    int doSomethingElse(String s);  
}
```

Novos Métodos de Interfaces

- Se quisermos adicionar um novo método, temos que redefinir a interface

```
public interface DoIt {  
    void doSomething(int i, double x);  
    int doSomethingElse(String s);  
    boolean didItWork(int i, double x, String s);  
}
```

- O problema é que todos os programas que usam essa interface vão falhar
 - Eles não implementam corretamente a nova versão da interface (falta o último método)

Novos Métodos de Interfaces

- Uma possível solução é criar outra interface que estende a antiga
 - Dessa forma, os programas que usam a interface antiga não vão falhar
 - Programadores podem optar por migrar ou não para a nova interface

```
public interface DoItPlus extends DoIt {  
    boolean didItWork(int i, double x, String s);  
}
```

Novos Métodos de Interfaces

- Uma outra alternativa é criar um método **default** na interface
 - Como ele já oferece a implementação, não causará falhas nos programas
 - Classe que implementa a interface não tem a obrigação de definir o comportamento do método **default**
 - Diferentemente dos métodos abstract

```
public interface DoIt {  
    void doSomething(int i, double x);  
    int doSomethingElse(String s);  
    default boolean didItWork(int i, double x, String s) {  
        // method body goes here  
    }  
}
```

Novos Métodos de Interfaces

- Quando uma interface que contém métodos **default** é estendida, podemos
 - Manter a implementação da interface estendida
 - Basta não mencionar o método na nova interface
 - Redefinir o método **default**, tornando-o abstract
 - Redefinir o método **default**, sobrescrevendo-o

Novos Métodos de Interfaces

- Colisão de interfaces
 - Se uma classe implementa duas interfaces que possuem métodos **default** com a mesma assinatura, o método deve ser sobrescrito
- Precedência de superclasses
 - Se uma superclasse provê uma implementação de um método que tem a mesma assinatura de um método **default** de uma interface, a implementação da superclasse tem precedência

Novos Métodos de Interfaces

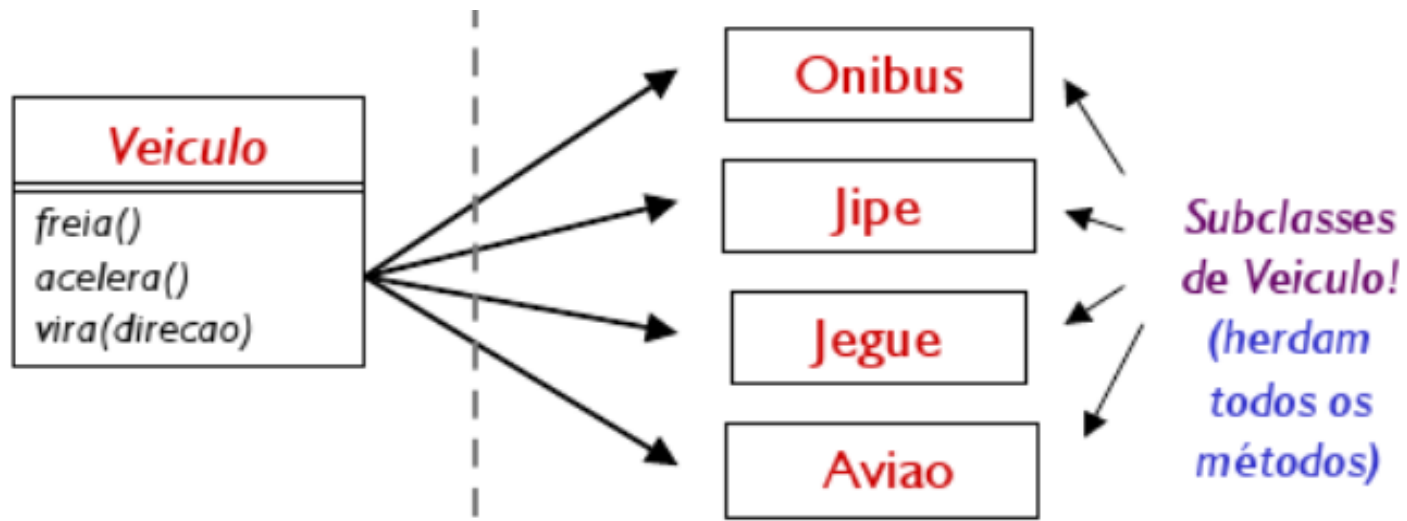
- Métodos **static**
 - Similar aos métodos **default**, devem conter uma implementação
 - Foram permitidos para organizar melhor alguns métodos que estão estritamente relacionados a uma interfaces e não a uma classe

Classes Abstratas

- Em muitos casos, desejamos definir uma classe geral, que representa objetos de maneira genérica, mas que não faz sentido possuir uma instância
- **Exemplo:** Classes Animal, Vaca, Gato, Ovelha
 - No mundo real, todo animal é de algum subtipo
 - Não faz sentido que exista um objeto Animal
 - Porém, a definição da classe Animal é vantajosa, pois permite compartilhar as características comuns de todos os animais
- Neste caso, faz mais sentido que a classe Animal seja **abstrata**
 - As outras classes são ditas **concretas**

Classes Abstratas

- Todo veículo será sempre de um dos subtipos
- Definimos, neste problema, que não faz sentido existir instâncias da classe Veiculo



Classes Abstratas

- Vimos que os métodos de uma interface são implicitamente abstratos
 - Não possuem corpo (implementação)
 - Estabelecem o contrato mas não o comportamento
 - Obrigam as classes a implementarem
- Métodos abstratos também podem ser definidos em uma classe, **desde que a classe seja abstrata**

Classes Abstratas

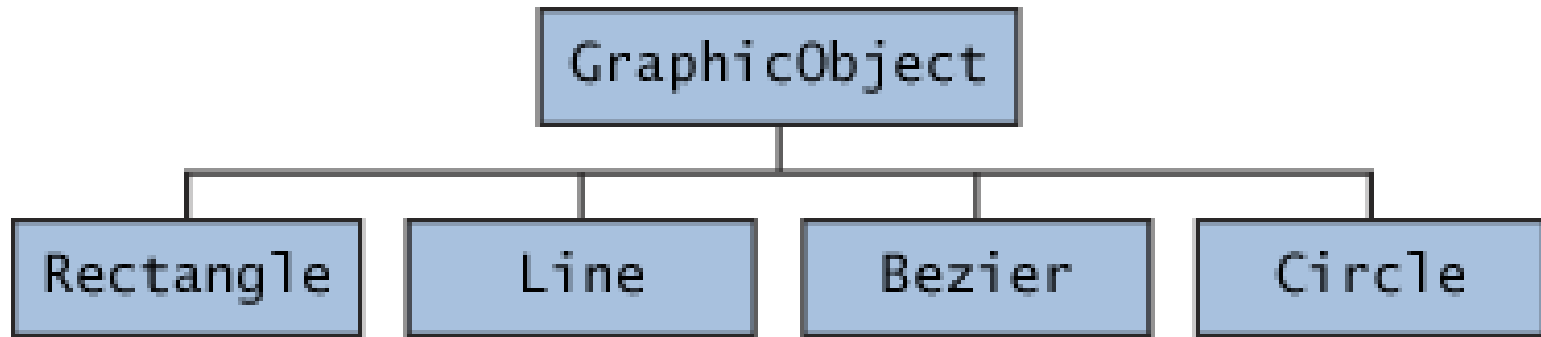
- Uma classe abstrata
 - Pode conter métodos abstratos e não abstratos
 - Pode conter campos como qualquer outra classe
 - Não pode ser instanciada (**new**)
 - Pode ser herdada
- Quando classes abstratas são herdadas, métodos abstratos devem ser implementados
 - Ou a nova classe deve ser declarada abstrata

```
public abstract class GraphicObject {  
    // declare fields  
    // declare non abstract methods  
  
    abstract void draw();  
}
```

Classes Abstratas

- **Exemplo.** Imagine uma aplicação para desenhar diferentes formas geométricas
 - Linhas, Círculos, Curvas Bezier e Retângulos
- Todas essas formas possuem estados e comportamentos
 - **Estados:** posição, orientação, cor da linha, cor de fundo
 - **Comportamentos:** moveTo, rotate, resize, draw
- Alguns atributos e comportamentos são iguais para todos os métodos
 - **Ex:** position, cor de fundo e moveTo

Classes Abstratas



- Esta é uma situação perfeita para uma superclasse abstrata
- Nela, os membros definem os estados e comportamentos compartilhados por todos os subtipos

Classes Abstratas

```
abstract class GraphicObject {  
    int x, y; // position  
    ...  
  
    void moveTo(int newX, int newY) {  
        x = newX;  
        y = newY;  
    }  
  
    abstract void draw();  
    abstract void resize();  
}
```


Classes Abstratas

```
class Circle extends GraphicObject {  
    void draw() {  
        ...  
    }  
    void resize() {  
        ...  
    }  
}
```

```
class Rectangle extends GraphicObject {  
    void draw() {  
        ...  
    }  
    void resize() {  
        ...  
    }  
}
```

Classe Abstrata Implementa Interface

- Uma classe que implementa uma interface deve, necessariamente, implementar todos os métodos abstratos
- Se a classe que implementa a interface for abstrata, essa exigência desaparece
 - Alguns métodos podem ser implementados e outros não
- Os métodos que ainda não foram definidos na classe abstrata deve ser definido na subclasse desta classe abstrata

Classe Abstrata Implementa Interface

```
abstract class AbstractClass implements Interface1 {  
    // implements all but one method of Interface1  
}
```

```
class ConcreteClass extends AbstractClass {  
    // implements the remaining method in Interface1  
}
```

Classes Abstratas vs Interfaces

- Semelhanças

- Ambas não podem ser instanciadas
- Ambas podem conter métodos com ou sem implementação
 - Antes do Java 8, interfaces não podiam conter métodos com implementação (**default** ou **static**)

- Diferenças

- Classes abstratas podem conter campos que não são public static final (constantes)
- Métodos concretos (não abstratos) em classes abstratas podem ter definido seu modificador de acesso
 - Em interfaces, qualquer método é sempre público

Classes Abstratas vs Interfaces

- Qual utilizar?
 - Depende da aplicação
- Em geral, interfaces são utilizadas por classes que não tem relação entre si
 - Serializable, Clonable, Comparable
 - Não existe uma relação forte (herança) entre as classes
- Se há a necessidade de oferecer atributos, interfaces não serão úteis
 - Com herança, os atributos serão herdados
 - Naturalmente existe uma dependência maior entre as classes

Classes Abstratas vs Interfaces

	Objetos	Herança	Métodos	Atributos	Construtor
Interface	Não pode ter instâncias	Uma classe pode implementar várias	Métodos abstratos, default e static	Somente constantes	Não pode ter
Classe Abstrata	Não pode ter instâncias	Uma classe pode estender apenas uma	Métodos concretos e abstratos	Constantes e atributos	Pode ter

Resumo

- Conceito de interface
- Interfaces em Java
- Atualizações no Java 8
- Classes abstratas
- Classes abstratas vs interfaces

Dúvidas?

