Java
Interfaces

UA, DETI, Programação III
José Luis Oliveira, Carlos Costa
2016/17

Interfaces

- Uma interface é uma classe abstracta pura (só contém assinaturas*).

```
public interface Desenhavel {
   //...
}
```

- Actua como um protocolo perante as classes que as implementam.

```
public class Grafico implements Desenhavel {
   // …
}
```

- Uma classe pode herdar de uma só classe base e implementar uma ou mais interfaces.

* Java 8: default and static methods

Interfaces - Exemplo

```
interface Desenhavel {
   public void cor(Color c);
   public void corDeFundo(Color cf);
   public void posicao(double x, double y);
   public void desenha(DrawWindow dw);
}

class CirculoGrafico extends Circulo implements Desenhavel {
   public void cor(Color c) {...}
   public void corDeFundo(Color cf) {...}
   public void posicao(double x, double y) {...}
   public void desenha(DrawWindow dw) {...}
}
```

56

Características principais

- Todos os seus métodos são, implicitamente, abstractos.
  - Os únicos modificadores permitidos são public e abstract.
- Uma interface pode herdar (extends) mais do que uma interface.
- Não são permitidos construtores.
- As variáveis são implicitamente estáticas e constantes
  - static final ..
- Uma classe (não abstracta) que implemente uma interface deve implementar todos os seus métodos.
- Uma interface pode ser vazia
  - Cloneable, Serializable
- Não se pode criar uma instância da interface
- Pode criar-se uma referência para uma interface

57

Interfaces em Java 8

- **Default Methods**
  - Oferecem um implementação por defeito
  - Podem ser reescritos nas classes que implementam a interface

```
public interface Interface1 {
  default void defMeth(){//... do something }
}
public class MyClass implements Interface1 {
  @Override
  public void defMeth(){ // ... do something       }
}
```

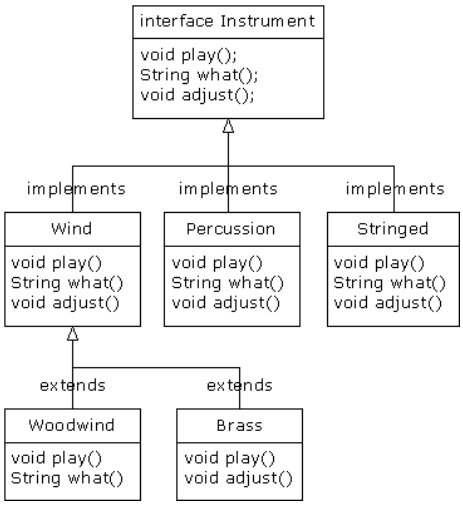
- **Static Methods**
  - Similares aos default methods
  - Não podem ser reescritos nas classes que implementam a interface

```
public interface Interface2 {
  static void stMeth(){//... do something }
}
public class MyClass implements Interface2 {
  @Override
  public void defMeth(){ // ... do something       }
}
```

58

Interfaces - Exemplos

- Depois de implementada uma interface passam a actuar as regras sobre classes



59

Interfaces - Exemplos

```
interface Instrument {
    // Compile-time constant:
  int i = 5; // static & final
    // Cannot have method definitions:
  void play(); // Automatically public
  String what();
  void adjust();
}

class Wind implements Instrument {
  public void play() {
    System.out.println("Wind.play()");
  }
  public String what() { return "Wind"; }
  public void adjust() {}
}
```

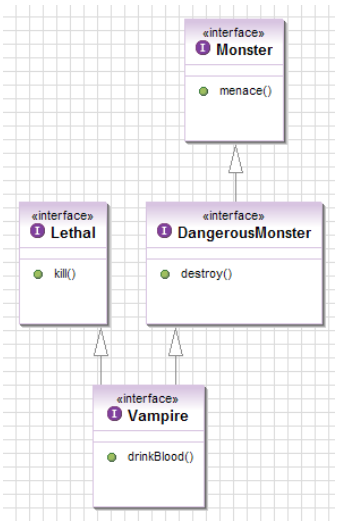

Herança em Interfaces

```
interface Monster {
  void menace();
}

interface DangerousMonster
   extends Monster {
  void destroy();
}

interface Lethal {
  void kill();
}

interface Vampire
   extends DangerousMonster,
           Lethal {
  void drinkBlood();
}
```



Classes Abstractas versus Interfaces

Classes Abstractas

- pode não ser 100% abstracta
- escrever software genérico, parametrizável e extensível
- relacionamento na hierarquia simples de classes

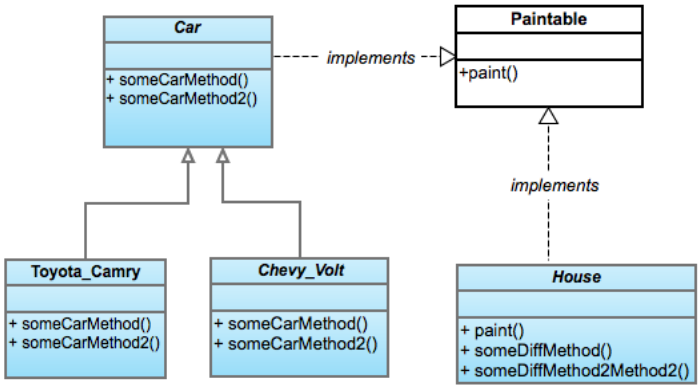
Interfaces

- 100% abstractas
  - Java 8: default and static methods
- especificar um conjunto adicional de comportamentos / propriedades funcionais
- implementação horizontal na hierarquia

Não há regras ou metodologia: "... neste caso usa-se interfaces, no outro ..."

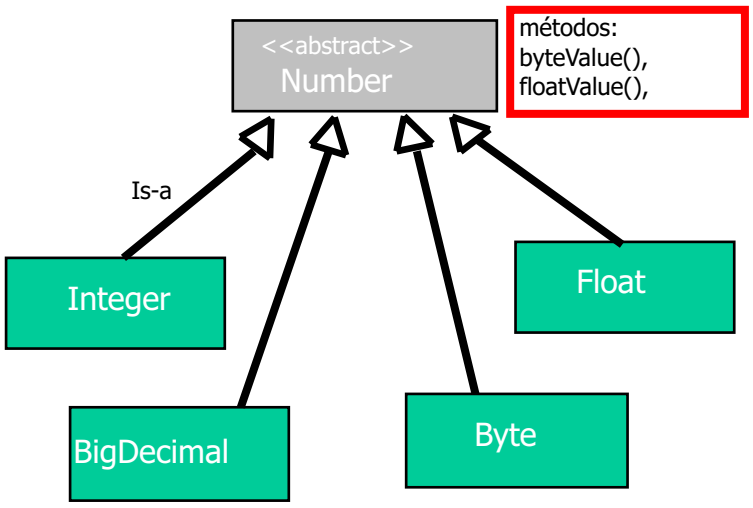
62

Classes Abstractas versus Interfaces



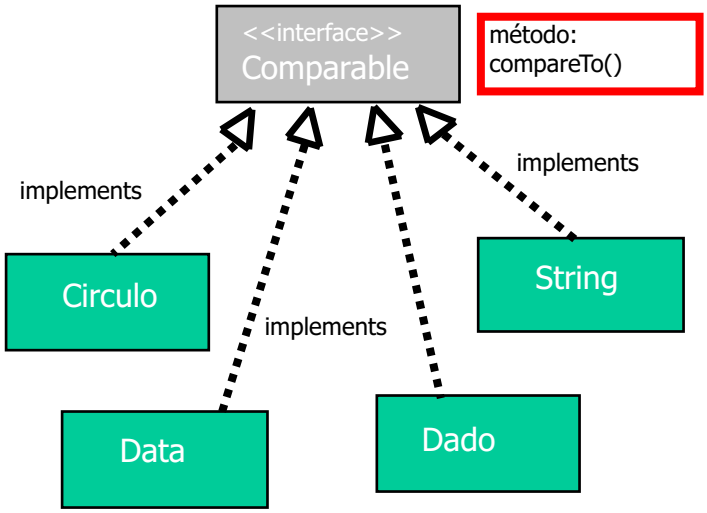
63

Classes Abstractas versus Interfaces



64

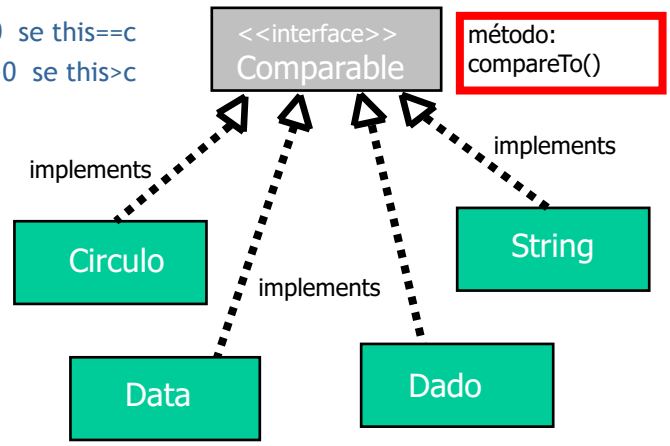
Classes Abstractas versus Interfaces



65

Questões?

- Qual o interesse de usar uma interface neste caso?
- Note que o método *int compareTo(Object c)* retorna:
  - <0 se this<c
  - 0 se this==c
  - >0 se this>c



Interface *Comparable*

```
public interface Comparable<T> { // package java.lang;
  int compareTo(T other);
}

public abstract class Shape implements Comparable<Shape> {
    public abstract double area( );
    public abstract double perimeter( );

    public int compareTo( Shape irhs )  {
      if(irhs==null)
            throw new NullPointerException("....");

      return area() - rhs.area();
  }
}
```

```
class FindMaxDemo {

public static <T> Comparable<T> findMax(Comparable<T>[] a)  {
  int maxIndex = 0;

  for ( int i = 1; i < a.length; i++ )
     if (a[i] != null && a[i].compareTo((T) a[maxIndex]) > 0)
        maxIndex = i;

        return a[maxIndex];
}

    public static void main( String[] args )  {
        Shape[] sh1 = { new Circle(  2.0 ),
                        new Square(  3.0 ),
                        new Rectangle( 3.0, 4.0 ) };

        String[] st1 = { "Joe", "Bob", "Bill", "Zeke" };

        System.out.println( findMax( sh1 ) );
        System.out.println( findMax( st1 ) );
    }
}
```

68

instanceof

- Instrução que indica se uma referência é membro de uma classe ou interface
- Exemplo, considerando

```
class Dog extends Animal implements Pet {...}
Animal fido = new Dog();
```

- as instruções seguintes são true:

```
if (fido instanceof Dog) ..
if (fido instanceof Animal) ..
if (fido instanceof Pet) ..
```

69

Copiar objetos (clone)

- protected Object clone()
  - Retorna um novo objeto cujo estado inicial é uma cópia do objeto sobre o qual o método foi invocado.
  - As alterações subsequente na réplica não afetarão o original.
  - Este método realiza uma cópia simples de todos os campos. Nem sempre é adequado.
- Construtor de cópia
  - Construtor cujo argumento é um objeto da mesma classe

```
public Figura(Figura original) {
...
}
```

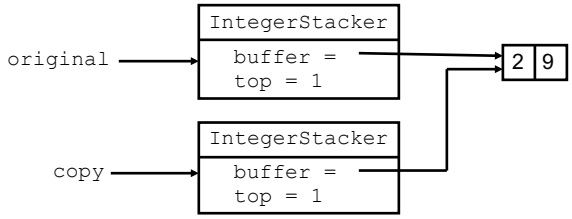
Não é comum em Java

É preferível usar o método clone()

Shallow cloning

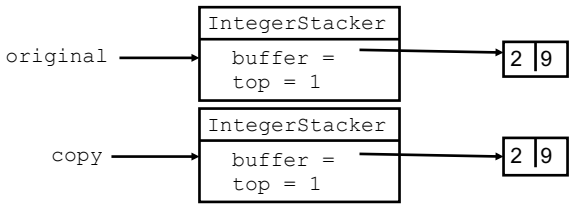
- Cópia campo a campo.
  - This might be wrong if it duplicates a reference to an object that shouldn't be shared.

```
public class IntegerStack {
  private int[] buffer; // a stack of integers
  private int top;      // largest index in the stacker
                        // (starting from 0)
  . . .
}
```



Deep cloning

- Cria uma réplica de todos os objectos que podem ser alcançados a partir do objeto que estamos a replicar



72

Interface java.lang.Cloneable

- Se quisermos fazer uso de Objecto.clone() temos de implementar a interface Cloneable
  - não tem métodos nem constantes (vazia) e funciona como um marcador

```
public class Rectangle implements Cloneable{
...
```

  - Shallow copy

```
@Override protected Rectangle clone() throws
CloneNotSupportedException {
      return (Rectangle) super.clone();
}
```

  - Deep copy – temos de ser nós a garantir a implementação local de clone()

```
@Override protected Rectangle clone() throws
CloneNotSupportedException {
      return new Rectangle(...);
}
```

73

Java
Classes internas

UA, DETI, Programação III
José Luis Oliveira, Carlos Costa
2016/17

Classes internas

- Classes podem ser membros de classes, de objetos ou locais a métodos. Podem até serem criadas sem nome, apenas com corpo no momento em que instanciam um objeto
  - Há poucas situações onde classes internas podem ou devem ser usadas. Devido à complexidade do código que as utiliza, deve evitar-se usos não convencionais
  - Usos típicos incluem tratamento de eventos em GUIs, criação de threads, manipulação de coleções e sockets
- Classes internas podem ser classificadas em quatro tipos
  - Classes estáticas - classes membros de classe (*nested classes*)
  - Classes de instância – classes membros de objetos
  - Classes locais – classes dentro de métodos
  - Classes anónimas - classes dentro de instruções

Classes estáticas

- São declaradas como *static* dentro de uma classe
- A classe externa age como um pacote para uma ou mais classes internas estáticas
  - *Externa.Coisa, Externa.InternaUm, ..*
- O compilador gera arquivos tipo *Externa$InternaUm.class*

```
class Externa {
   private static class InternaUm {
      public int campo;
      public void metodoInterno() {...}
   }
   public static class InternaDois
                        extends InternaUm {
      public int campo2;
      public void metodoInterno() {...}
   }
   public static interface Coisa {
      void existe();
   }
   public void metodoExterno() {...}
}
```

76

Classes de instância

- São membros do objeto, como métodos e atributos
- Requerem que objeto exista antes que possam ser usadas.
  - Externamente usa-se *referência.new* para criar objetos
- Deve usar-se *NomeDaClasse.this* para aceder a campos internos

```
class Externa {
    public int campoUm;
    public class Interna {
        public int campoUm;
        public int campoDois;
        public void metodoInterno() {
            this.campoUm = 10;          // Externa.campoUm
            Interna.this.campoUm = 15;
        }
    }
    public static void main(String[] args){
        Interna e = (new Externa()).new Interna();
    }
}
```

77

Classes locais

- Servem para tarefas temporárias já que deixam de existir quando o método acaba
  - Têm o mesmo alcance de variáveis locais.

```
public Multiplicavel calcular(final int a, final int b) {
   class Interna implements Multiplicavel {
      public int produto() {
         return a * b; // usa a e b, que são constantes
      }
   }
   return new Interna();
}
public static void main(String[] args){
   Multiplicavel mul = (new Externa()).calcular(3,4);
   int prod = mul.produto();
}
```

78

Classes anónimas

- Servem para criar um único objeto
  - A classe abaixo estende ou implementa SuperClasse, que pode ser uma interface ou classe abstracta (o new, neste caso, indica a criação da classe entre chavetas, não da SuperClasse)

```
Object i = new SuperClasse() { implementação };
```

  - O compilador gera arquivo Externa$1.class, Externa$2.class,

```
public Multiplicavel calcular(final int a, final int b) {
   return new Multiplicavel()  {
      public int produto() {
         return a * b;
      }
   };
}
public static void main(String[] args){
   Multiplicavel mul = (new Externa()).calcular(3,4);
   int prod = mul.produto();
}
```

Compare com parte em preto e vermelho do slide anterior!

A classe está dentro da instrução: preste atenção no ponto-e-vírgula!

79

Classes internas

- São sempre classes dentro de classes. Exemplo:

```
class Externa {
   private class Interna {
      public int campo;
      public void metodoInterno() {...}
   }
   public void metodoExterno() {...}
}
```

- Podem ser *private*, *protected*, *public* ou *package-private*
  - Excepto as que aparecem dentro de métodos, que são locais
- Podem ser estáticas:
  - E chamadas usando a notação Externa.Interna
- Podem ser de instância e depender da existência de objetos:

```
Externa e = new Externa();
Externa.Interna ei = e.new Externa.Interna();
```

- Podem ser locais (dentro de métodos)
  - E nas suas instruções podem não ter nome (anónimas)

80

Sumário

- Polimorfismo
- Generalização
- Classes abstractas
- Interfaces
- Classes internas

81