Polimorfismo
Classes abstratas
Interfaces

Polimorfismo

- Polimorfismo é a terceira caraterística fundamental de programação orientada a objetos, conjuntamente com o encapsulamento e a herança.

- Polimorfismo permite tratar vários tipos (derivados do mesmo tipo base) como se fossem um só tipo.

A chamada dum método polimórfico permite que um tipo expresse a sua distinção de outro tipo, desde que ambos sejam derivados do mesmo tipo base.

- Polimorfismo é também conhecido sob o nome de ligação dinâmica (*dynamic binding*, *late binding*, *run-time binding*).

Upcasting e *downcasting*

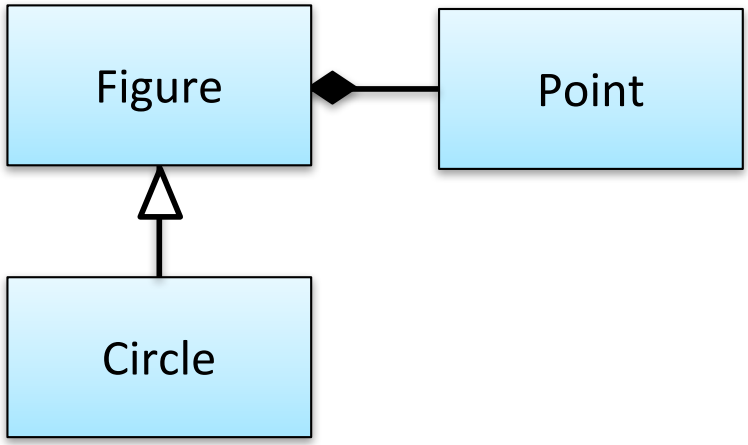
O processo de conversão duma referência da classe derivada para uma referência da classe base chama-se ***upcasting***.

O processo de conversão duma referência da classe base para uma referência da classe derivada chama-se ***downcasting***.

O tipo do objeto pode ser testado com o operador de comparação de tipos **instanceof**.

Exemplo:

```
//upcast:
Figure f = new Circle("amarelo", new Point(1.0, 1.0), 2.2);
//downcast:
Circle c1 = f; //erro
Circle c2 = (Circle)f;

if (f instanceof Circle)
  c2 = (Circle) f;
```



Ligação dinâmica

Ligação dinâmica permite determinar o tipo do objeto durante o correr do programa (em *run-time*) e chamar métodos certos que correspondem ao tipo real.

Os programas tornam-se extensíveis porque podemos adicionar funcionalidades novas herdando tipos novos de uma classe base comum.

As funções que manipulam a interface da classe base não precisam de ser modificadas para poderem suportar as classes novas!

Exemplo de comportamento polimórfico

```
class Pessoa {
  private String nome;
  private int idade;

  public Pessoa(String nome, int age) { this.nome = nome; idade = age; }

  @Override
  public String toString(){ return nome + " " + idade; }
}

public class Aluno extends Pessoa {
  private int nMec;

  public Aluno(String nome, int age, int num)
  {  super(nome, age); nMec = num; }

  @Override
  public String toString(){ return super.toString() + " " + nMec; }

  public static void main(String[] args) {
     Aluno a = new Aluno("Paulo Ferreira", 20, 77888);
     System.out.println(a);
     Pessoa p = a;
     System.out.println(p);
  }
}
```

```
Paulo Ferreira 20 77888
Paulo Ferreira 20 77888
```

O compilador vai gerar código que determinará o tipo de objeto em *run-time* e invocará o método toString() para este tipo (Aluno).

Uso de comportamento polimórfico

O comportamento polimórfico é normalmente usado com coleções de objetos.

```
public class AlunoPosGr extends Aluno {
   private Pessoa orientador;

   public AlunoPosGr(String nome, int age, int num, Pessoa or) {
      super(nome, age, num);
      orientador = or;
   }

   @Override
   public String toString(){
      return super.toString() + " " + "Orientador: " + orientador;
   }

   public static void main(String[] args) {
      Aluno[] turma = new Aluno[3];
      turma[0] = new Aluno("Ana", 20, 77777);
      turma[1] = new AlunoPosGr("João", 27, 44777,
                        new Pessoa("Carlos Bastos", 44));
      turma[2] = new Aluno("Pedro", 20, 77123);

      for (Aluno a : turma)
         System.out.println(a);
   }
}
```

```
Ana 20 77777
João 27 44777 Orientador: Carlos Bastos 44
Pedro 20 77123
```

Ligação dinâmica vs. ligação estática

Em Java todos os métodos (à exceção dos construtores, **final** e **static**) têm comportamento polimórfico e resultam na ligação dinâmica implementada pelo compilador.

Métodos privados, bem como métodos explicitamente declarados como **final**, não podem ser redefinidos. Logo, para estes o compilador implementa ligação estática (*early binding*), i.e. determina o método a chamar ainda durante a compilação e não durante o correr do programa.

Ligação estática resulta num código ligeiramente mais eficiente (porque não há necessidade de determinar o tipo de objeto e localizar o método a chamar em *run-time*).

Métodos polimórficos em construtores

Nunca chame métodos polimórficos em construtores porque pode referir a objetos que ainda não foram construídos!

Exemplo:

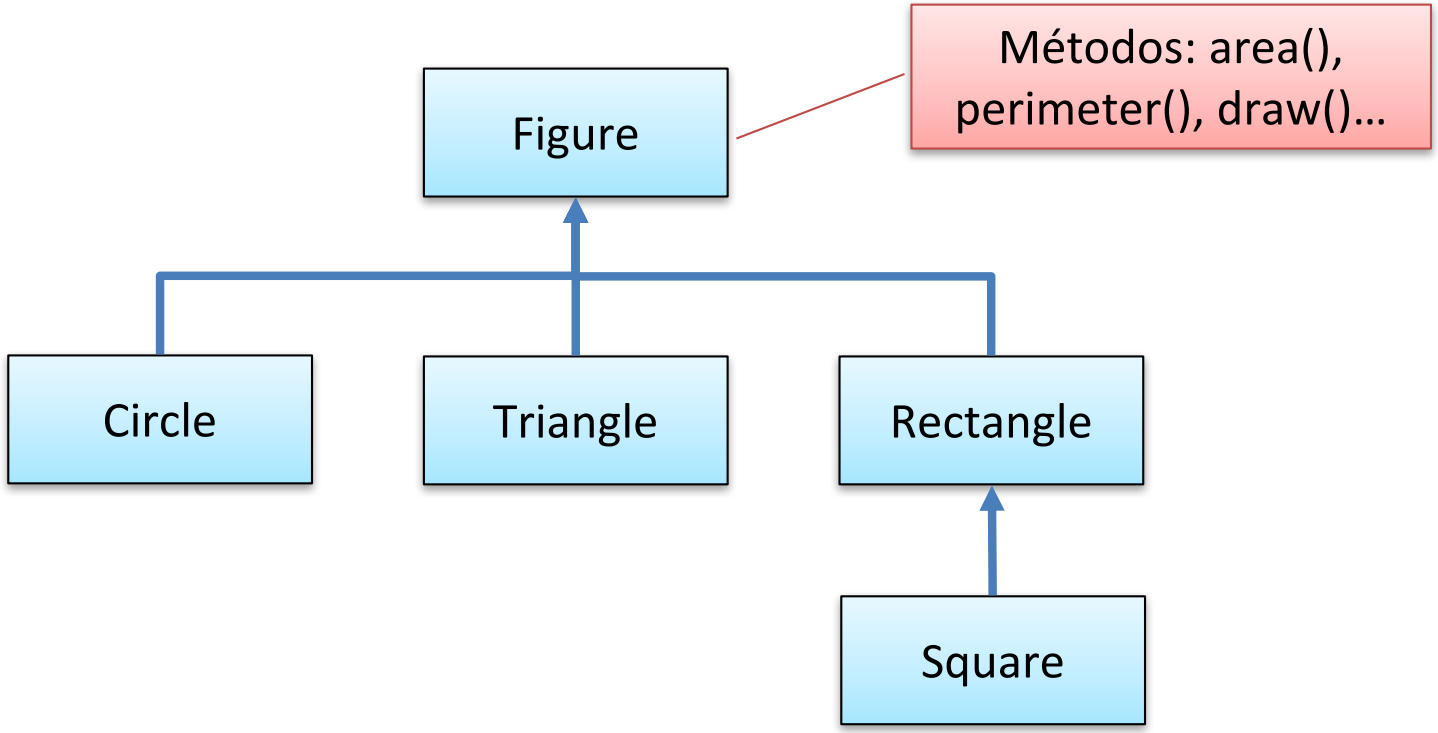
```
public Pessoa(String nome, int age){
  this.nome = nome;
  idade = age;
  System.out.println(this); --erro
}
…
Aluno[] turma = new Aluno[2];
turma[0] = new Aluno("Ana", 20, 77777);
turma[1] = new AlunoPosGr("João", 27, 44777,
        new Pessoa("Carlos Bastos", 44));
```

```
Ana 20 0
Carlos Bastos 44
João 27 0 Orientador: null
```

Generalização

Normalmente, tentamos garantir que a classe base seja mais abrangente possível incluindo nela todos os métodos que sejam precisos nas classes derivadas.

Estes métodos são posteriormente redefinidos em classes derivadas.



Como implementar os métodos para o tipo Figure?

Classes abstratas

No exemplo anterior, a única razão de criar a classe Figure consiste em garantir a interface comum para classes derivadas.

Os objetos de tipo Figure não fazem sentido e nunca serão criados na prática, pois o programa vai manipular figuras específicas tais como círculos e quadrados, mas não figuras abstratas.

Sendo assim, a classe Figure deve ser abstrata para expressamente proibir que o utilizador crie objetos do tipo Figure.

Uma classe é abstrata se esta incluir pelo menos um método abstrato. Uma classe abstrata não é instanciável (não se pode criar objetos dela).

Um método abstrato é um método cujo corpo não é definido.

Exemplo de classe abstrata

```
abstract class Figure {
   // métodos abstratos
   public abstract double area();
   public abstract double perimeter();

   // pode incluir métodos não abstratos
   public String toString() { return "Figura"; }
}

public class Circle extends Figure {
   private double radius;

   public Circle(double d) { radius = d; }

   public double area() { return Math.PI * radius * radius; }
   public double perimeter() { return 2 * Math.PI * radius; }

   public static void main(String[] args) {
    Figure f; //é permitido criar referências para Figure
    f = new Figure(); //erro, não podemos criar objetos
    f = new Circle(2.2);
    Circle c = new Circle(1.0);
    System.out.printf("%4.2f\n", c.perimeter());
    System.out.printf("%4.2f\n", f.perimeter());
    System.out.println(f);
}
}
```

```
6.28
13.82
Figura
```

Classes abstratas e herança

Se derivar da classe abstrata deve implementar todos os métodos abstratos da classe base.

Caso não o faça, a classe derivada também será abstrata e o compilador vai forçar a inserção da palavra-chave **abstract** antes da definição da classe.

Exercícios

Identifique erros no programa seguinte. Qual é a saída deste programa (depois de corrigidos os erros)?

```
abstract class X {
   private final static int a;
   X (int i) { a = i; }
   public abstract void f();
}

final class Y extends X {
   private char c;
   Y(int i, char c) { this.c = c; super(i); }

   public void f() { System.out.println("Y.f()"); }
}

public class Teste extends Y {
   @Override
   public void f() { System.out.println("Teste.f()"); }

   public static void main(String[] args) {
       Teste t = new Teste();
       Y y = new Y(10, 'a');
       t.f();
       y.f();
       X x = t;
       x.f();
       t = x;
   }
}
```

```
Teste.f()
Y.f()
Teste.f()
```

Exercícios (cont.)

Identifique erros no programa seguinte. Qual é a saída deste programa (depois de corrigidos os erros)?

```
abstract class A {}

public class Teste extends A {
  public void f() { System.out.println("Teste.f()"); }

  public static void m(A a) {
     if (a instanceof Teste)
        ((Teste)a).f();
  }

  public static void main(String[] args) {
     m(new A());
     m(new Teste());
  }
}
```

Interfaces

- Uma interface funciona como uma classe abstracta pura (podemos descrever apenas assinaturas).

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

- Atua 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.

Interfaces - Exemplo

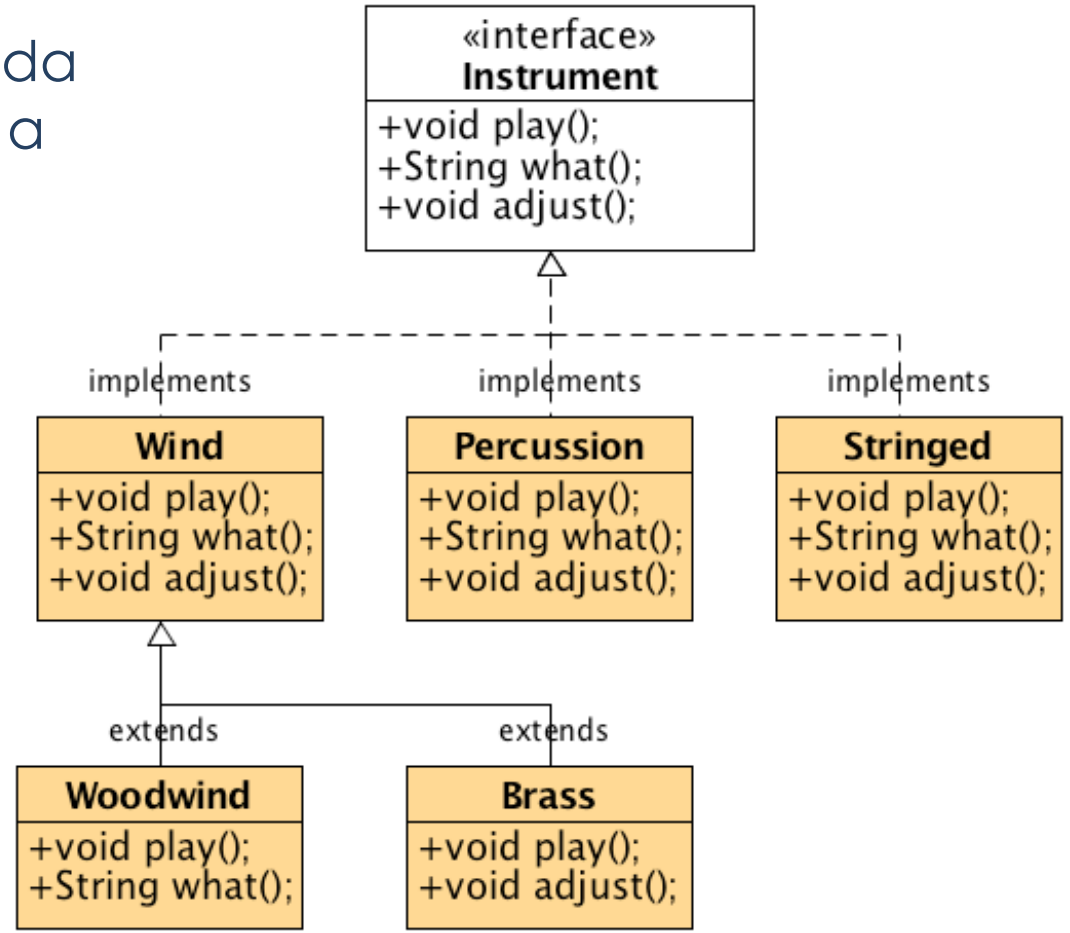
```
interface Desenhavel {
   public void cor(Color c);
   public void corDeFundo(Color cf);
   public void posicao(double x, double y);
   public void desenha(DrawWindow dw);
}

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) {...}
}
```

Características principais

- Os seus métodos são, implicitamente, abstractos.
  - Os únicos modificadores permitidos são *public* e *abstract*.
  - A partir do Java 8, passaram a ser também *static* e *default*.
- Uma interface pode herdar (extends) mais do que uma interface.
- Não são permitidos construtores.
- Os atributos são implicitamente estáticos 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

Interfaces - Exemplos

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

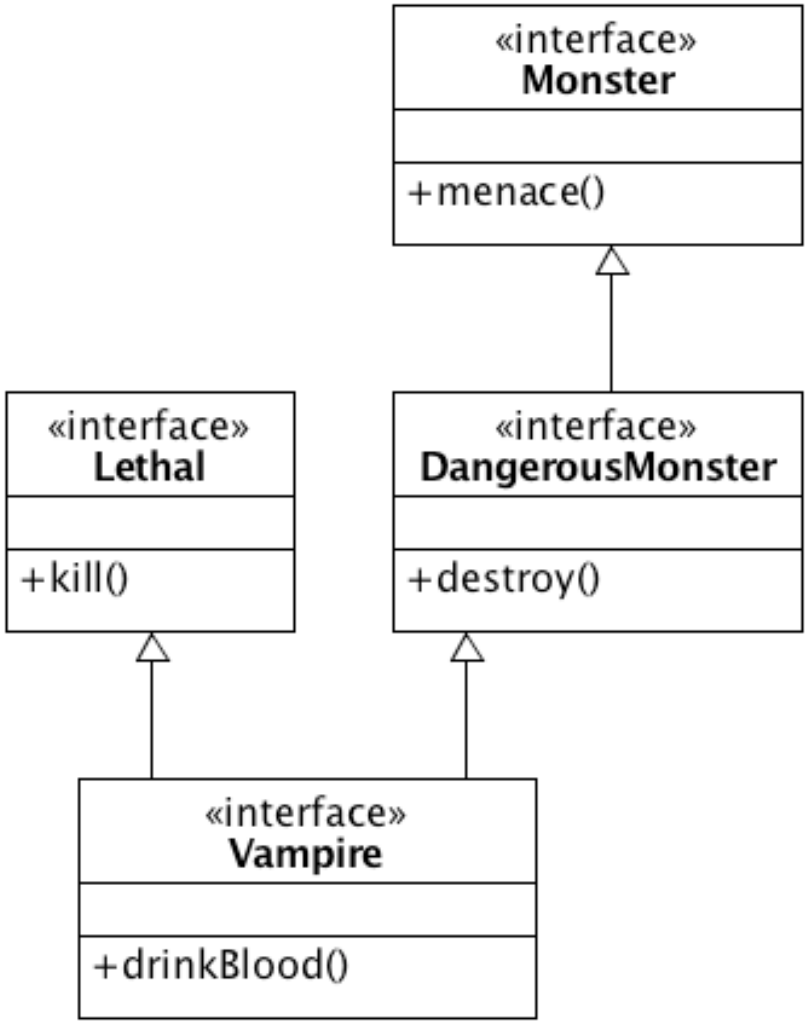


Interfaces - Exemplos

```
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();
}
```



Interfaces em Java 8

- Default methods
  - Podemos definir o corpo dos métodos na interface
- Static methods
  - Podemos definir o corpo de métodos estáticos na interface. Devem ser invocados sobre a interface (Métodos de Interface)
- Functional interfaces
  - *(vamos falar nisto mais tarde…)*

- **porquê** (complicar com) **estas novas funcionalidades?**

Default methods

```
interface X {
    default void foo() {
        System.out.println("foo");
    }
}

class Y implements X {
    // ...
}

public class Testes {
    public static void main(String[] args) {
        Y myY = new Y();
        myY.foo();
        // ...
    }
}
```

Static methods

```
interface X {
    static void foo() {
        System.out.println("foo");
    }
}

class Y implements X {
    // ...
}

public class Testes {
    public static void main(String[] args) {
        X.foo();
        // Y.foo(); // won't compile
    }
}
```

Classes Abstractas versus Interfaces

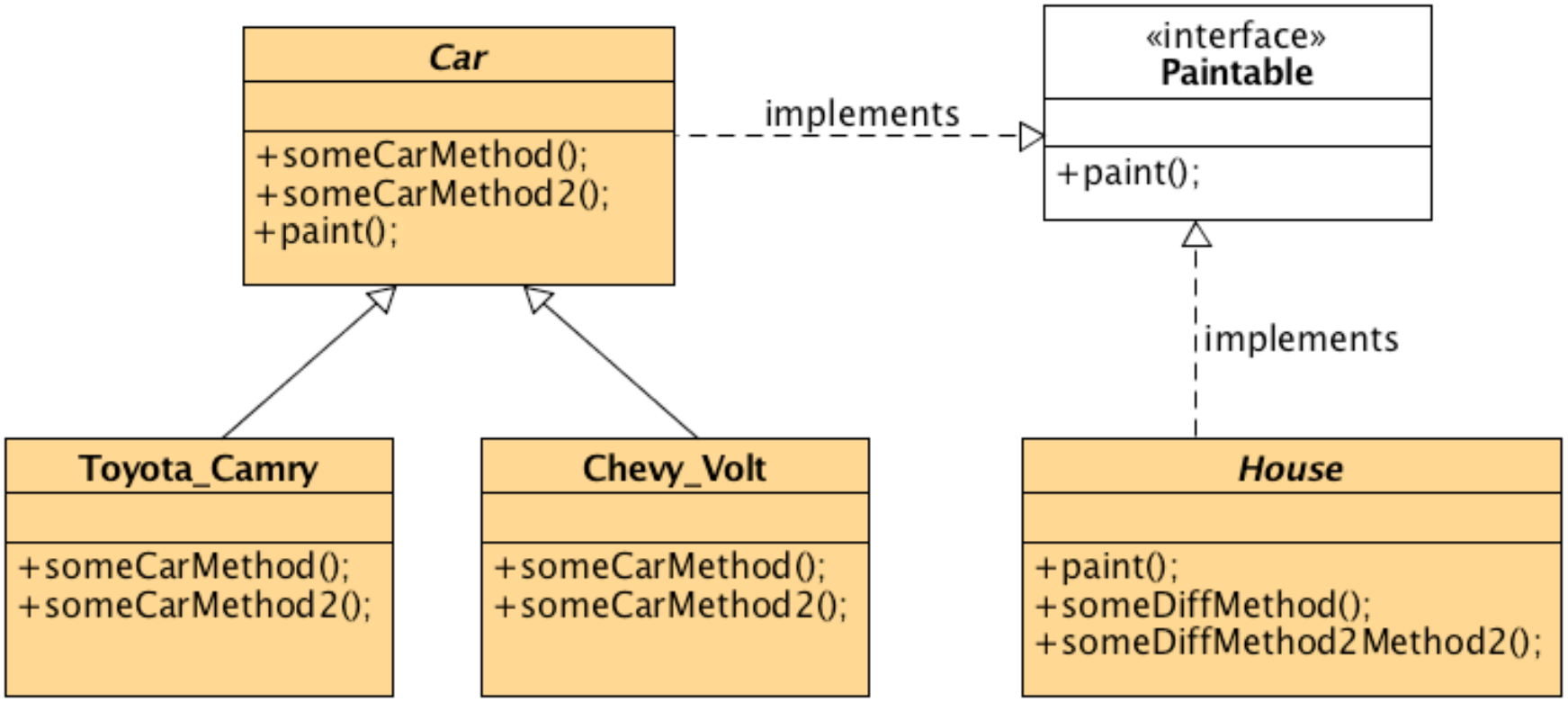
Classes Abstractas

- escrever software genérico, parametrizável e extensível
- relacionamento na hierarquia simples de classes

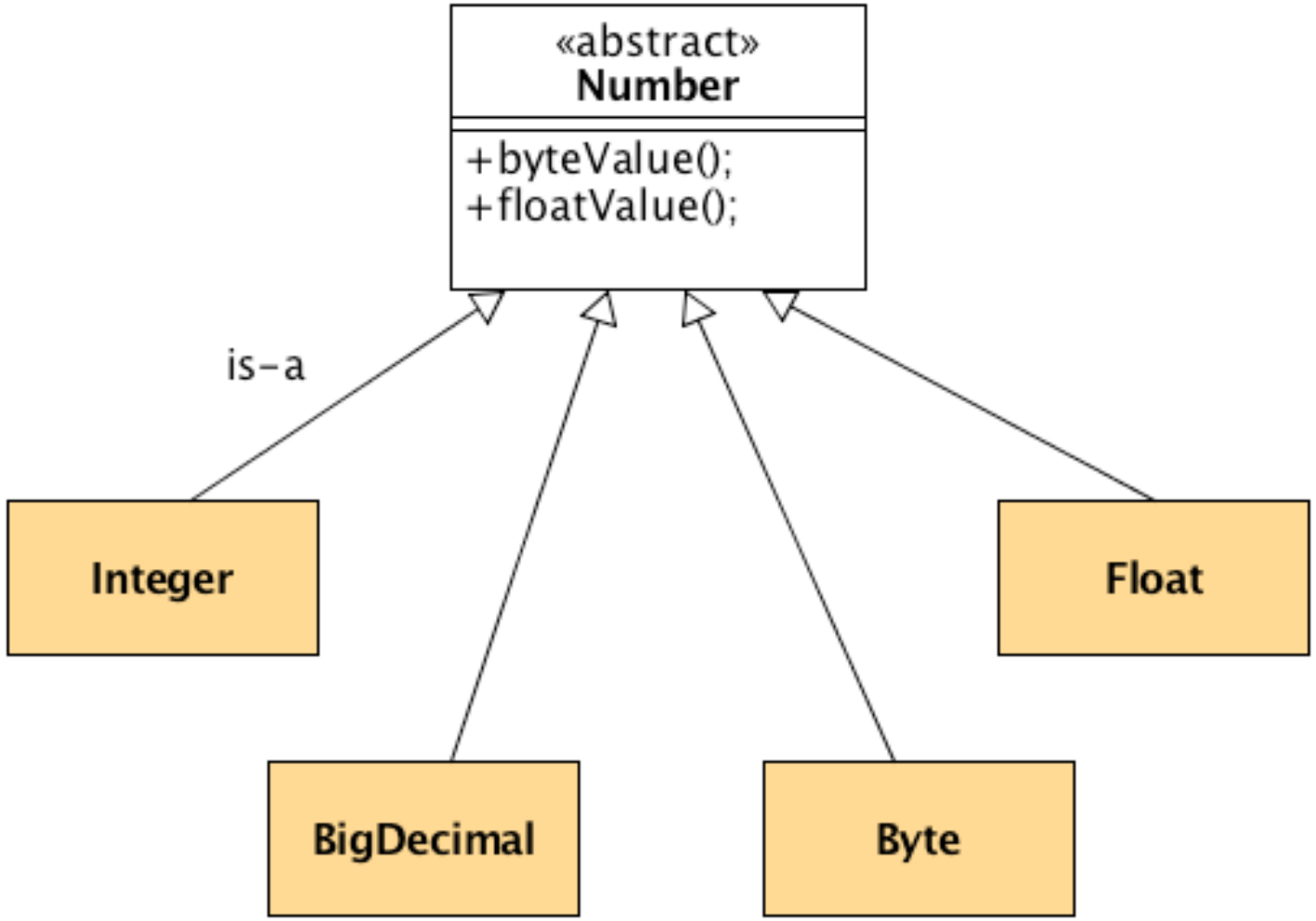
Interfaces

- especificar um conjunto adicional de propriedades funcionais
- implementação horizontal na hierarquia

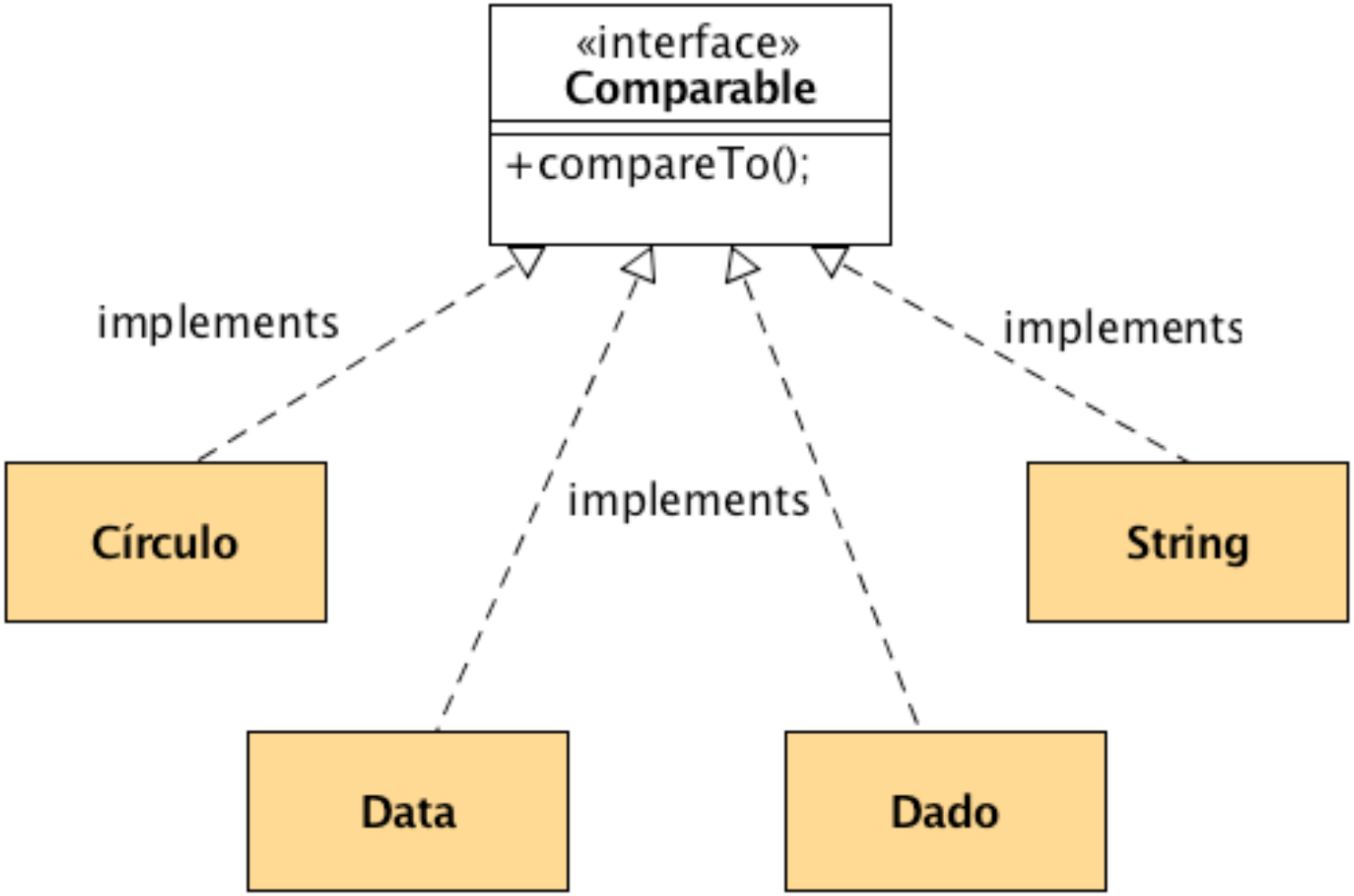
Classes Abstractas versus Interfaces



Classes Abstractas versus Interfaces

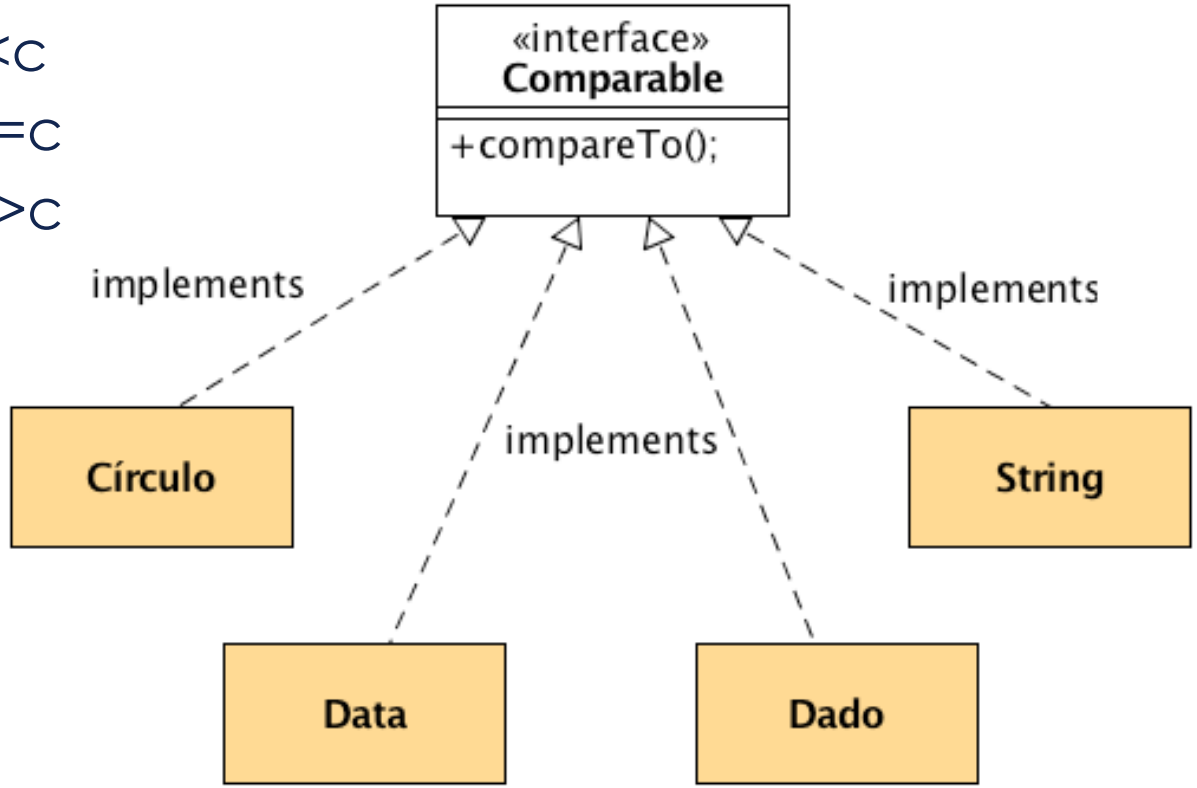


Classes Abstractas versus Interfaces



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 )  {
        double res = area() - irhs.area();
        if (res > 0) return 1;
        else if (res < 0) return -1;
            else return 0;
    }
}
```

Interface Comparable

```
public class UtilCompare {
   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 <T> void sortArray(Comparable<T>[] a)      {
      // ...
   }
}
```

Interface Comparable

```
class FindMaxDemo {
   public static void main( String [ ] args )  {
      Figura[] sh1 = {
            new Circulo(1, 3, 1),
            new Quadrado(3, 4, 2),
            new Rectangulo(1, 1, 5, 6), };
      String[] st1 = { "Joe", "Bob", "Bill", "Zeke" };

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

```
Rectangulo de Centro (1.0,1.0), altura 6.0, comprimento 5.0
Zeke
```

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) ..
```

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

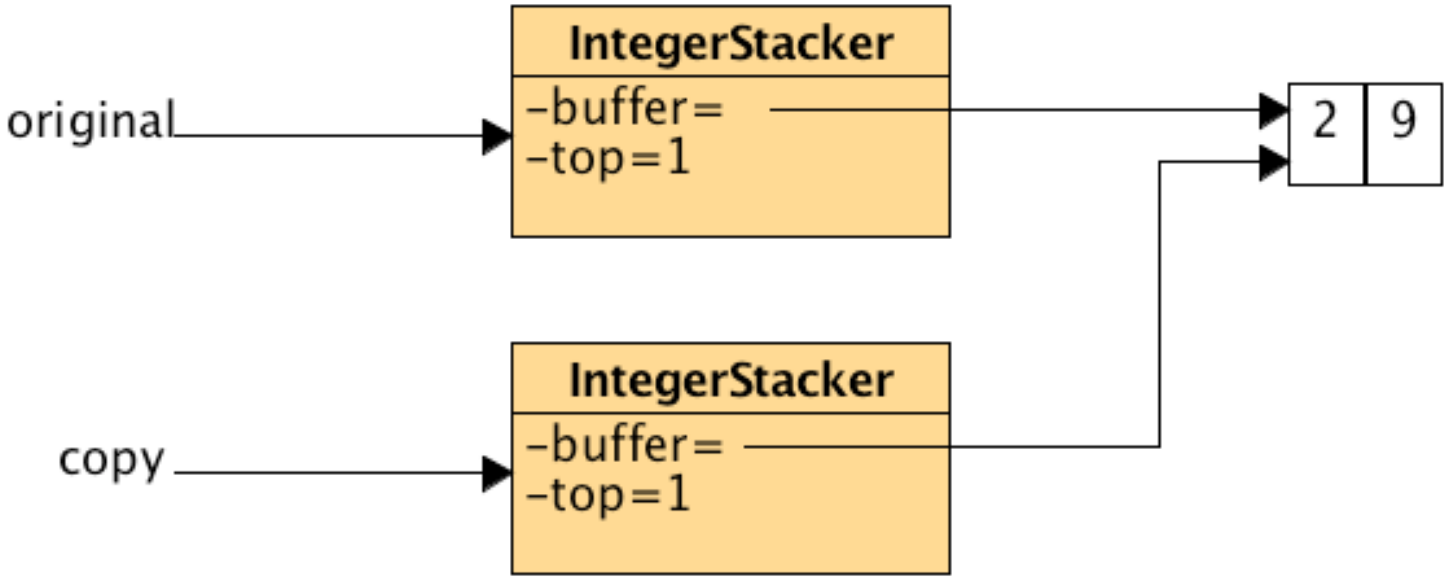
```
public Figura(Figura original) {
...
}
```

Solução muito usada em C++, mas pouco comum em Java.
→ Object:clone()

Shallow cloning

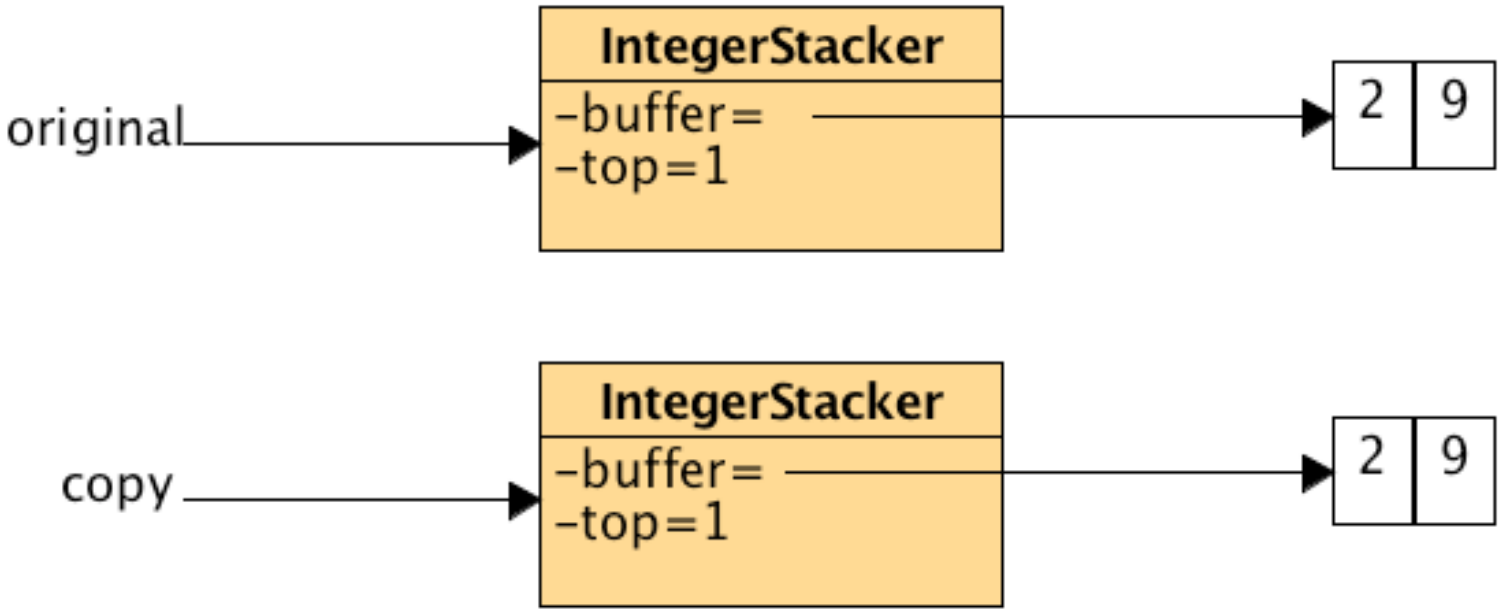
- Cópia campo a campo.

```
public class IntegerStack {
   private int[] buffer; // a stacker 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



Interface java.lang.Cloneable

- Se quisermos fazer uso de *Object.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(...);
}
```