

Hierarquia de Classes e Herança

- **(Grady Booch) *The Meaning of Hierarchy:***
 - *“Abstraction is a good thing, but in all except the most trivial applications, we may find many more different abstractions than we can comprehend at one time. Encapsulation helps manage this complexity by hiding the inside view of our abstractions. Modularity helps also, by giving us a way to cluster logically related abstractions. Still, this is not enough. A set of abstractions often forms a hierarchy, and by identifying these hierarchies in our design, we greatly simplify our understanding of the problem.”*
- Logo, *“Hierarchy is a ranking or ordering of abstractions.”*

- Até agora só temos visto classes que estão ao mesmo nível hierárquico. No entanto...
- A colocação das classes numa hierarquia de especialização (do mais genérico ao mais concreto) é uma característica de muitas linguagens da POO
- Esta hierarquia é importante:
 - ao nível da **reutilização** de variáveis e métodos
 - da compatibilidade de tipos

- No entanto, a tarefa de criação de uma hierarquia de conceitos (classes) é complexa, porque exige que se **classifiquem** os conceitos envolvidos
- A criação de uma hierarquia é do ponto de vista operacional um dos mecanismos que temos para criar novos conceitos a partir de conceitos existentes
- de notar que a este propósito já utilizamos a composição de classes

- Exemplos de composição de classes
 - um segmento de recta é composto por duas instâncias de Ponto
 - um Triângulo pode ser definido como composto por três segmentos de recta ou por um segmento e um ponto central, ou ainda por três pontos
 - uma Turma é composta por uma colecção de alunos

- Uma outra forma de criar classes a partir de classes já existentes é através do mecanismo de herança.
- Considere-se que se pretende criar uma classe que represente um Ponto 3D
- quais são as alterações em relação ao Ponto que codificamos anteriormente?
 - mais uma v.i. e métodos associados

- A classe Ponto (incompleta):

```
/**
 * Classe que implementa um Ponto num plano2D.
 * As coordenadas do Ponto são inteiras.
 *
 * @author MaterialP00
 * @version 20180212
 */
public class Ponto {

    //variáveis de instância
    private int x;
    private int y;

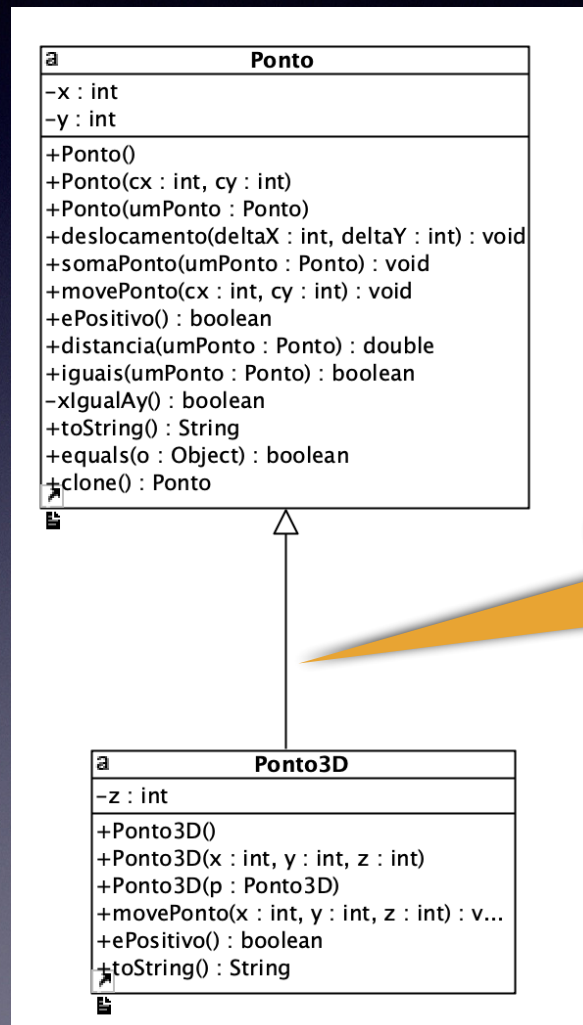
    /**
     * Construtores da classe Ponto.
     * Declaração dos construtores por omissão (vazio),
     * parametrizado e de cópia.
     */
}
```

- o esforço de codificação consiste em acrescentar uma v.i. (z) e getZ() e setZ()

- O mecanismo de herança proporciona um esforço de programação diferencial
- ou seja, para ter um Ponto3D precisamos de tudo o que existe em Ponto e acrescentar um *delta de informação* que consiste nas características novas
- logo, a classe Ponto3D aumenta, refina, detalha, especializa a classe Ponto

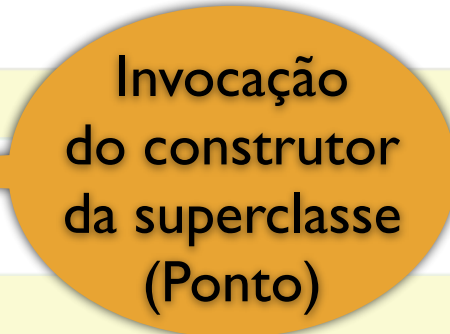
- Como se faz isto?
 1. de forma ad-hoc, sem suporte, através de um mecanismo de copy&paste 🤔😱
 2. usando composição, isto é, tendo como v.i. de Ponto3D um Ponto 🤔
 3. *mais importante*, através de um mecanismo existente de base nas linguagens por objectos que é a noção de hierarquia e herança 👍

Diagrama de classe com herança



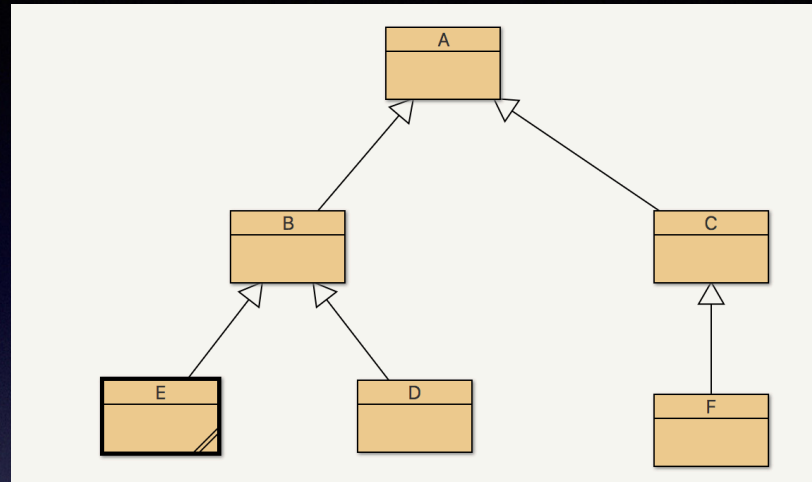
Denota a
relação de herança e de
especialização


```
public class Ponto3D extends Ponto {  
    private int z;  
  
    public Ponto3D() {  
        super();  
        this.z = 0;  
    }  
  
    public Ponto3D(int x, int y, int z) {  
        super(x,y);  
        this.z = z;  
    }  
  
    public Ponto3D(Ponto3D p) {  
        super(p);  
        this.z = p.getZ();  
    }  
}
```



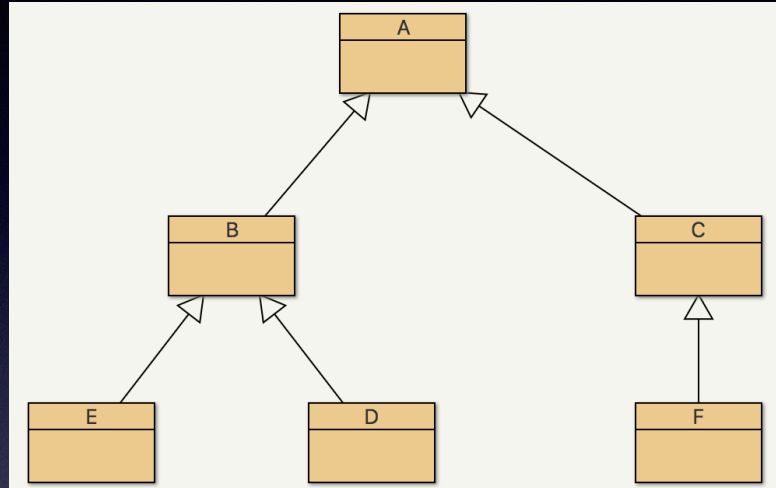
Invocação
do construtor
da superclasse
(Ponto)

- Hierarquia:



- A é superclasse de B.
- A é superclasse de C.
- B é superclasse de D e E
- D e E são subclasses de B
- F é subclasse de C
- B especializa A, D e E especializam B (e A!)

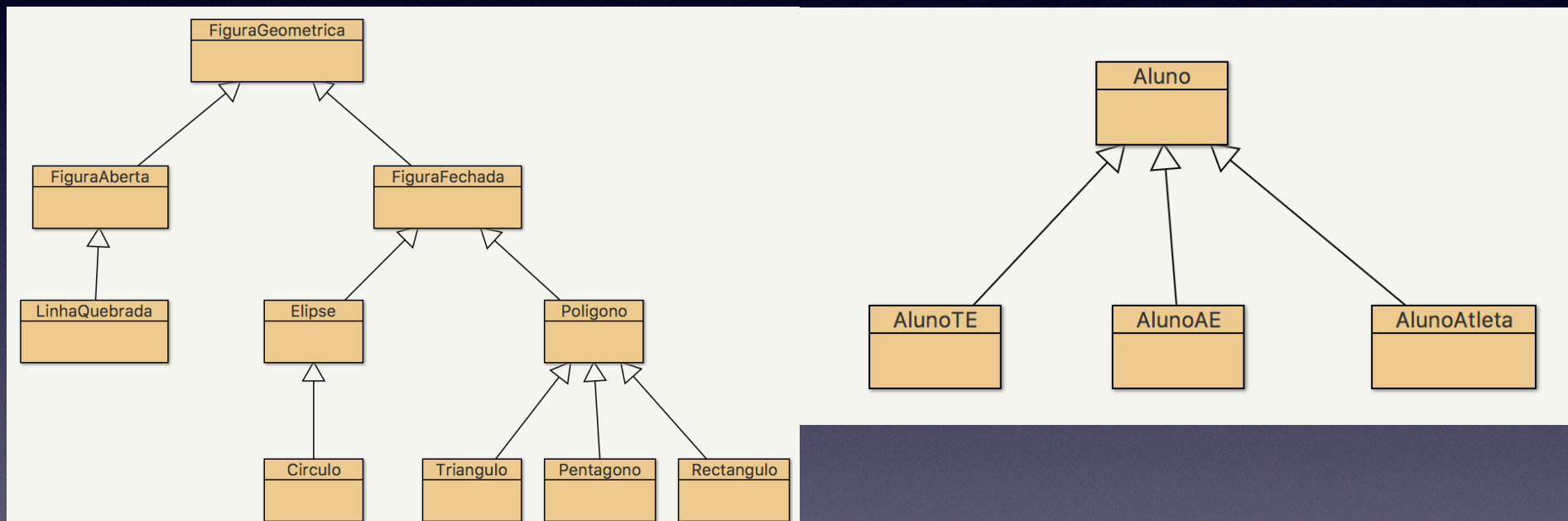
- Hierarquia típica em Java:



- hierarquia de herança simples (por oposição, p.ex., a C++)
- O que significa do ponto de vista semântico dizer que duas classes estão hierarquicamente relacionadas?

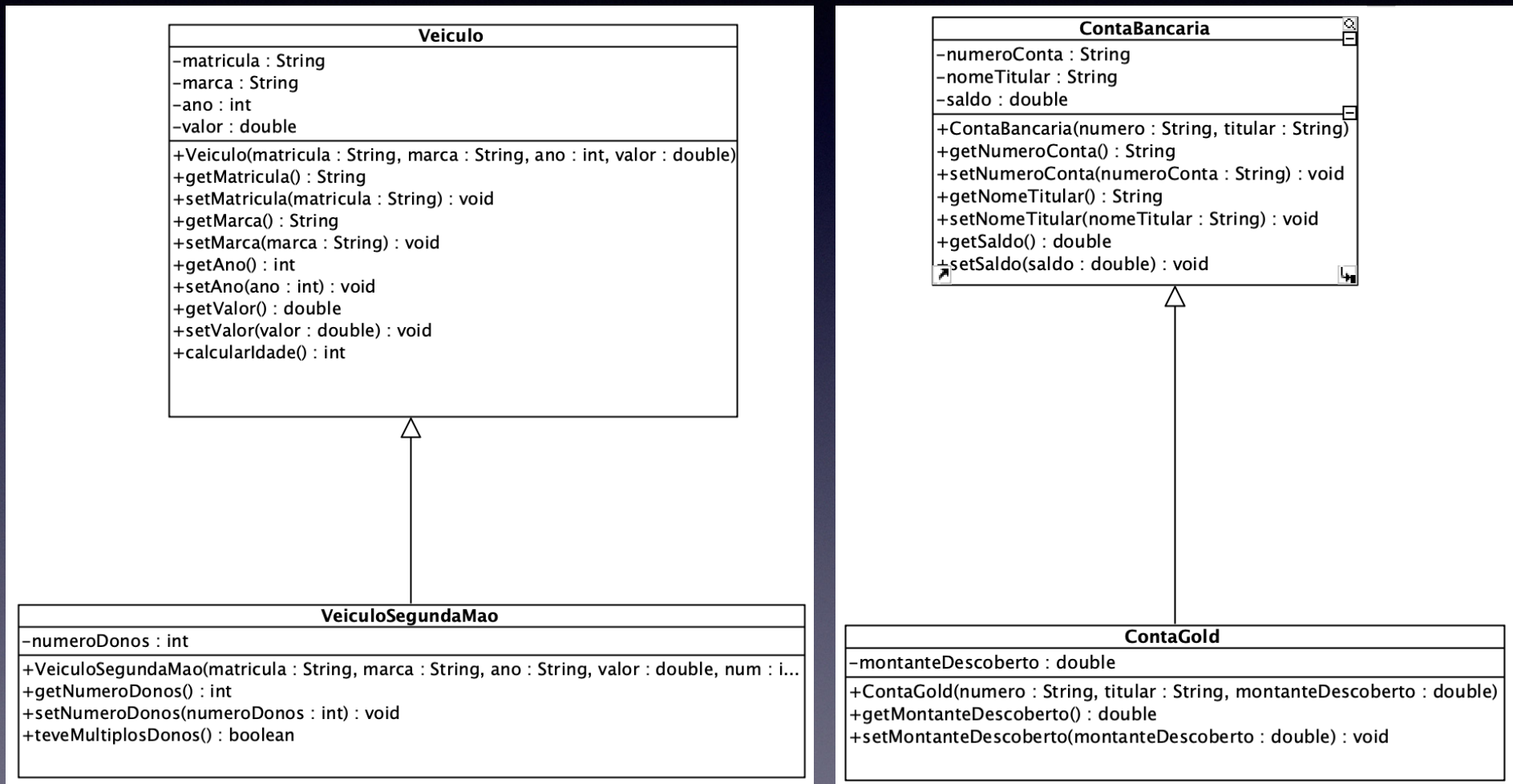
- no paradigma dos objectos a hierarquia de classes é uma hierarquia de especialização
- uma subclasse de uma dada classe constitui uma especialização, sendo por definição mais detalhada que a classe que lhe deu origem
- isto é, possui **mais** estado e **mais** comportamento

- A exemplo de outras taxonomias, a classificação do conhecimento é realizada do geral para o particular



- a especialização pode ser feita nas duas vertentes: estrutural (variáveis) e comportamental (métodos)

Mais exemplos...



(adaptado de “Java in Two Semesters”, Q. Charatan, A. Kans)

O mecanismo de herança

- se uma classe B é subclasse de A, então:
 - B é uma especialização de A
 - este relacionamento designa-se por “é *um*” ou “é *do tipo*”, isto é, uma instância de B pode ser designada como sendo um A
 - implica que aos atributos e métodos de A se acrescentou mais informação

- Se uma classe B é subclasse de A:
 - se B **pertence** ao mesmo package de A, B herda e pode aceder directamente a todas as variáveis e métodos de instância que não são private.
 - se B **não pertence** ao mesmo package de A, B herda e pode aceder directamente a todas as variáveis e métodos de instância que não são private ou package. Herda automaticamente tudo o que é public ou protected.

- B pode **definir** novas variáveis e métodos de instância próprios
- B pode **redefinir** variáveis e métodos de instância herdados (fazer override)
- variáveis e métodos de classe são herdados mas...
 - se forem redefinidos são *hidden*, não são *overridden*.
- métodos construtores não são herdados

- na definição que temos utilizado nesta unidade curricular, as variáveis de instância são declaradas como **private**
- que impacto é que isto tem no mecanismo de herança?
- vamos deixar de poder referir as v.i. da superclasse que herdamos pelo nome
- ..., mas vamos utilizar os métodos de acesso, *getX()* (no caso do Ponto), para aceder aos seus valores

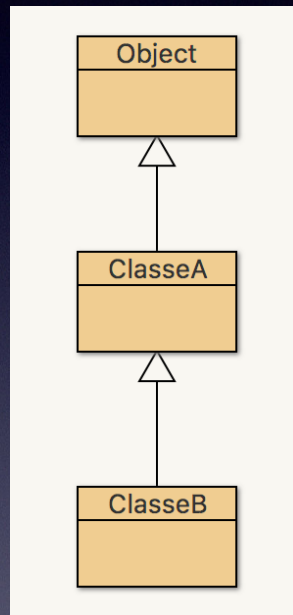
- Para percebermos a dinâmica do mecanismo de herança, vamos prestar especial atenção aos seguintes aspectos:
 - criação de instâncias das subclasses
 - redefinição de variáveis e métodos
 - procura de métodos

Criação das instâncias das subclasses

- em Java é possível definir um construtor à custa de um construtor da mesma classe, ou seja, à custa de **this()**
- fica agora a questão de saber se é possível a um construtor de uma subclasse invocar os construtores da superclasse
 - como vimos atrás os construtores não são herdados

- quando temos uma subclasse B de A, sabe-se que B herda todas as v.i. de A a que tem acesso.
- assim cada instância de B é constituída pela “soma” das partes:
 - as v.i. declaradas em B
 - as v.i. herdadas de A

- em termos de estrutura interna, podemos dizer que temos:



```
public class ClasseA extends Object {  
  
    private int a1;  
    private String a2;  
}
```

```
public class ClasseB extends ClasseA {  
  
    private int b1;  
    private String b2;  
}
```

- como sabemos que B tem pelo menos um construtor definido, B(), as v.i. declaradas em B (b1 e b2) são inicializadas

- ... mas quem inicializa as variáveis que foram declaradas em A?
- resposta evidente: os métodos encarregues de fazer isso em A, ou seja, os construtores de A
- dessa forma, o construtor de B deve invocar o construtor de A para inicializar as v.i. declaradas em A

- em Java, para que seja possível a invocação do construtor de uma superclasse, esta deve ser feita logo no início do construtor da subclasse
- recorrendo a **super(...)**, em que a verificação do construtor a invocar se faz pelo matching dos parâmetros e respectivos tipos de dados
- de facto a invocação de um construtor numa subclasse, cria uma cadeia transitiva de invocações de construtores

- Exemplo classe Ponto3D, subclasse de Ponto
- os construtores de Ponto3D delegam nos construtores de Ponto a inicialização das v.i. declaradas em Ponto

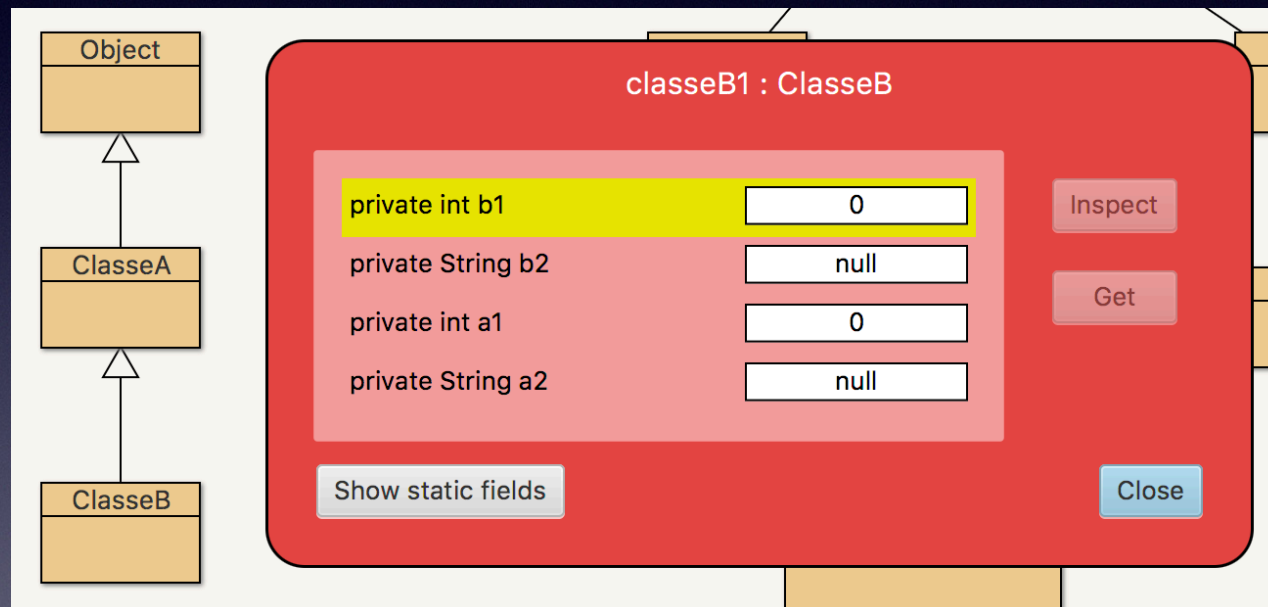
```
public Ponto3D() {  
    super();  
    this.z = 0;  
}  
  
public Ponto3D(int x, int y, int z) {  
    super(x,y);  
    this.z = z;  
}  
  
public Ponto3D(Ponto3D p) {  
    super(p);  
    this.z = p.getZ();  
}
```

construtor
parametrizado de
Ponto

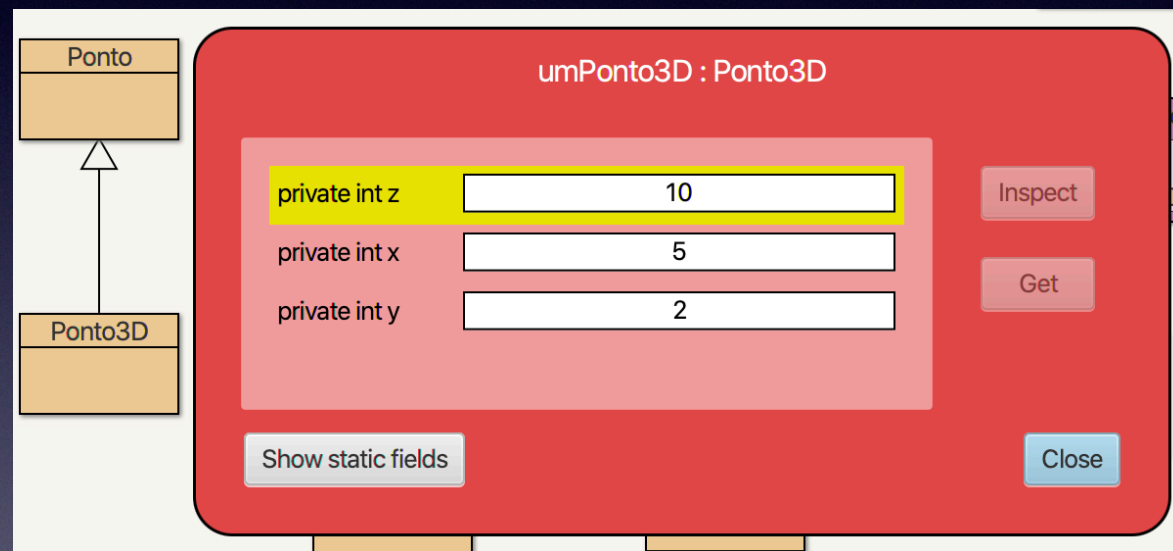
construtor de
cópia de Ponto

- a cadeia de construtores é implícita e na pior das hipóteses usa os construtores que por omissão são definidos em Java.
- por isso em Java são disponibilizados por omissão construtores vazios
- por aqui se percebe o que Java faz quando cria uma instância: aloca espaço e inicializa todas as v.i. que são criadas pelas diversas classes até Object

- Exemplo de criação de uma instância da classe ClasseB, recorrendo ao construtor vazio:



- Exemplo de criação de um objecto da classe Ponto3D



Redefinição variáveis e métodos

- o mecanismo de herança é automático e total, o que significa que uma classe herda obrigatoriamente da sua superclasse directa, e superclasses por transitividade, um conjunto de variáveis e métodos
- no entanto, uma determinada subclasse pode pretender modificar localmente uma definição herdada
 - a definição local é sempre a prioritária

- na literatura quando um método é redefinido, é comum dizer que ele é reescrito ou *overriden*
- quando uma variável de instância é re-declarada na subclasse diz-se que a da superclasse é escondida (*hidden* ou *shadowed*)
- A questão é saber se ao redefinir estes conceitos se perdemos, ou não, o acesso ao que foi herdado!

- considere-se a classe ClasseA

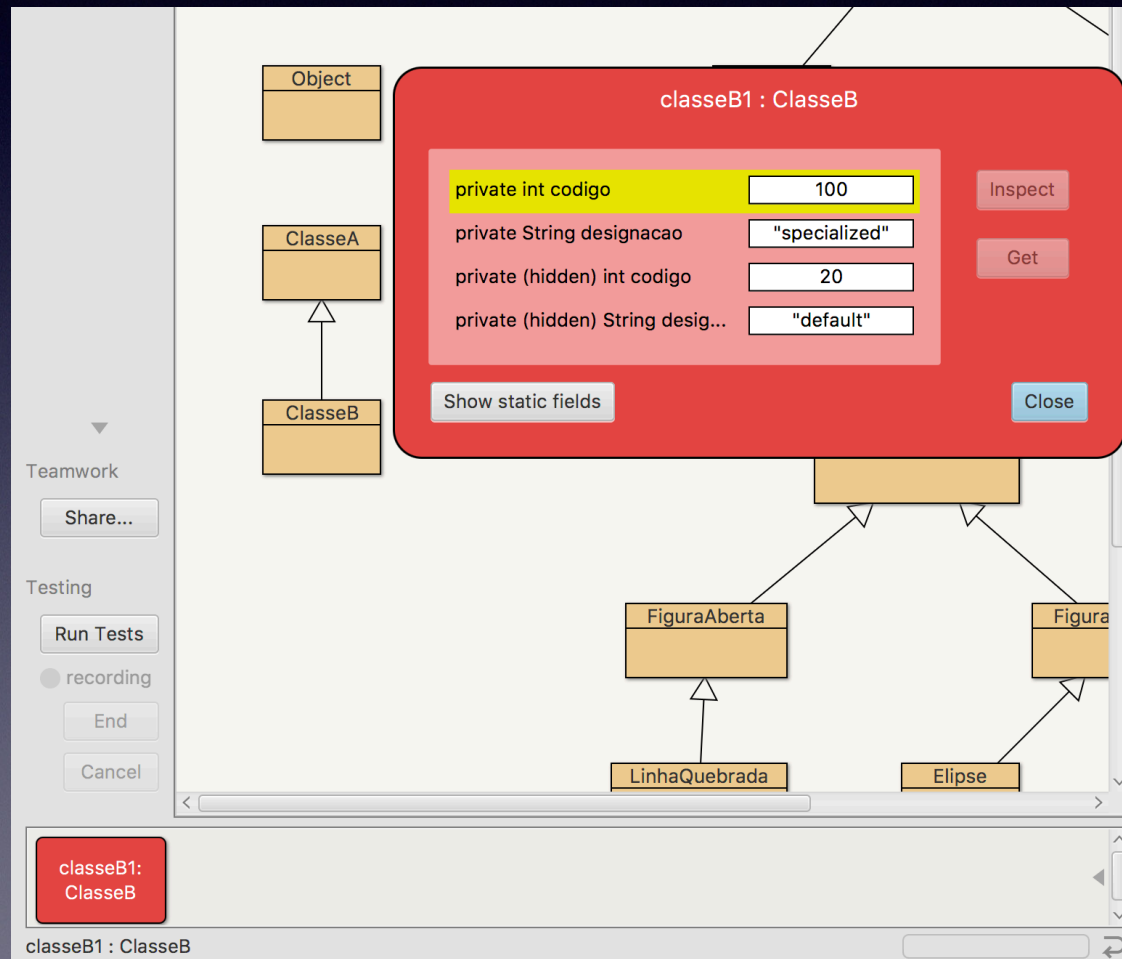
```
public class ClasseA {  
    private int codigo;  
    private String designacao;  
  
    public ClasseA() {  
        this.codigo = 20;  
        this.designacao = "default";  
    }  
    public int getCodigo() { return this.codigo;}  
    public String getDesignacao() {return this.designacao;}  
    public int metodo() {return this.getCodigo();}  
    public int resultado() {return this.getCodigo();}  
}
```

- e uma sua subclasse, ClasseB

```
public class ClasseB extends ClasseA {  
  
    private int codigo; // esconde a v.i. de ClasseA  
    private String designacao; //esconde a v.i. de ClasseA  
  
    public ClasseB() {  
        this.codigo = 100;  
        this.designacao = "specialized";  
    }  
    public int getCodigo() {return this.codigo;}  
    public String getDesignacao() {return this.designacao;}  
    public int metodo() {return this.getCodigo();}  
    public int metodoA() {return super.metodo();}  
    public int metodoB() {return metodoA();}  
}
```


- o que é a referência **super**?
 - um identificador que permite que a procura seja remetida para a superclasse
 - ao fazer **super.m()**, a procura do método **m()** é feita na superclasse e não na classe da instância que recebeu a mensagem
 - apesar da sobreposição (*override*), tanto o método local como o da superclasse estão disponíveis

- veja-se o inspector BlueJ de um objecto da Classe B



- é também possível visualizar os métodos definidos na classe e os herdados da(s) superclasse(s)

The screenshot shows a Java IDE interface. On the left, a class hierarchy is displayed with `ClasseA` as the superclass and `ClasseB` as the subclass, indicated by a hollow triangle arrow. In the center, a red-bordered window titled `classeB1 : ClasseB` displays the instance variables of a `ClasseB` object. The variables are:

- `private int codigo` with value `100`
- `private String designacao` with value `"specialized"`
- `private (hidden) int codigo` with value `20`
- `private (hidden) String desig...` with value `"default"`

Buttons for `Inspect`, `Get`, `Show static fields`, and `Close` are visible in the window. Below the class hierarchy, a context menu is open, showing the following items:

- inherited from Object
- inherited from ClasseA
- `int getCodigo()`
- `String getDesignacao()`
- `int metodo()`
- `int metodoA()`
- `int metodoB()`

Below the context menu, a list of methods is shown, indicating where they are redefined:

- `int getCodigo() [redefined in ClasseB]`
- `String getDesignacao() [redefined in ClasseB]`
- `int metodo() [redefined in ClasseB]`
- `int resultado()`

An orange speech bubble points to the context menu with the text: **Métodos herdados de ClasseA**

- o que acontece quando enviamos à instância classeBI (imagem anterior) a mensagem `resultado()`?
- **`resultado()`** é uma mensagem que não foi definida na subclasse
- o algoritmo de procura vai encontrar a definição na superclasse
- o código a executar é
`return this.getCodigo()`

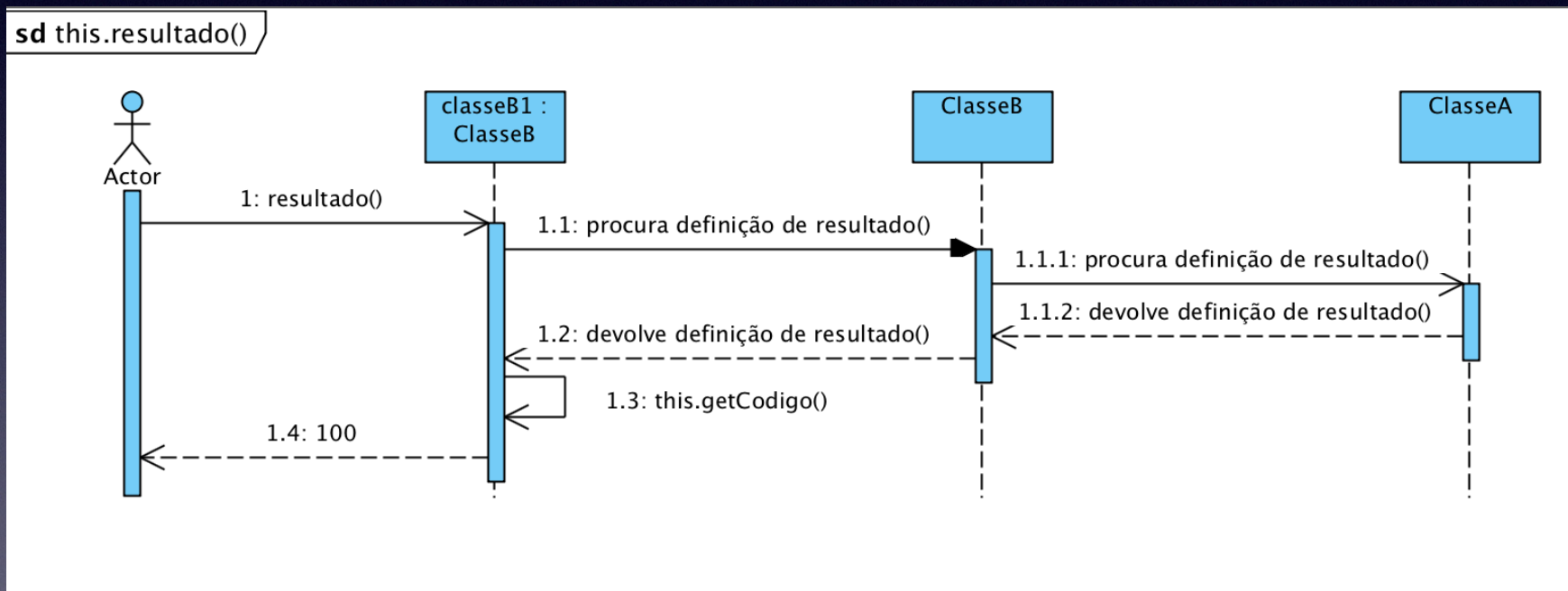
- em ClasseA o valor de codigo é 20, enquanto que em ClasseB o valor é 100.
- qual é o contexto de execução de **this.getCodigo()**?
- a que instância é que o **this** se refere?
- Vejamos o algoritmo de procura e execução de métodos...

- qual o resultado de invocar resultado() numa instância de ClasseB?

```
public class ClasseA {  
    private int codigo;  
    private String designacao;  
  
    public ClasseA() {  
        this.codigo = 20;  
        this.designacao = "default";  
    }  
  
    public int getCodigo() { return this.codigo;}  
    public String getDesignacao() {return this.designacao;}  
    public int metodo() {return this.getCodigo();}  
    public int resultado() {return this.getCodigo();}  
}
```

```
public class ClasseB extends ClasseA {  
  
    private int codigo; // esconde a v.i. de ClasseA  
    private String designacao; //esconde a v.i. de ClasseA  
  
    public ClasseB() {  
        this.codigo = 100;  
        this.designacao = "specialized";  
    }  
  
    public int getCodigo() {return this.codigo;}  
    public String getDesignacao() {return this.designacao;}  
    public int metodo() {return this.getCodigo();}  
    public int metodoA() {return super.metodo();}  
    public int metodoB() {return metodoA();}  
}
```


- algoritmo de execução da invocação de resultado() no objecto classeB1:



- na execução do código, a referência a **this** corresponde sempre ao objecto que recebeu a mensagem
- neste caso, a instância *classeB*
- sendo assim, o método **getCodigo()** é o método de ClasseB, que é a classe do receptor da mensagem
- logo, independentemente do contexto “subir e descer”, o this refere sempre o receptor da mensagem!

- E qual o resultado da invocação em classeB1 (instância de ClasseB) dos seguintes métodos?

```
public int metodo() {return this.getCodigo();}  
public int metodoA() {return super.metodo();}  
public int metodoB() {return metodoA();}
```


Regra para avaliação de `this.m()`

- de forma geral, a expressão **`this.m()`**, onde quer que seja encontrada no código de um método de uma classe (independentemente da localização na hierarquia), corresponde sempre à execução do método **`m()`** da classe do receptor da mensagem

Modificadores e redefinição de métodos

- a possibilidade de redefinição de métodos está condicionada pelo tipo de modificadores de acesso do método da superclasse (private, public, protected, package) e do método redefinidor
- o método redefinidor não pode diminuir o nível de acessibilidade do método redefinido

- os métodos public podem ser redefinidos por métodos public
- métodos protected por public ou protected
- métodos package por public ou protected ou package