Ainda sobre ordenações

- Temos visto que podemos ordenar colecções de dados recorrendo:
  - à ordem natural, através do método `compareTo()` (interface `Comparable<T>`)
  - a uma relação de ordem a fornecer, através do método `compare(...)` (interface `Comparator<T>`)

- Temos utilizado o `TreeSet<E>` como mecanismo base para fazer ordenações:
  - tirando partido de que o TreeSet utiliza uma relação de ordem para colocar os objectos
  - como o conjunto não admite repetidos é preciso especial cuidado com comparações que devolvem 0 (são iguais)
    - ou se acrescentam mais critérios de comparação ou então alguns elementos são ignorados

- Uma outra forma é utilizar as `List<E>` para efectuar a ordenação, não tendo que ter de prever a situação de repetição de dados
  - utilizando o método `List.sort(c)`
  - utilizando o `sorted()` e `sorted(c)` das streams

com recurso a List.sort

```java
/**
 * A estratégia de colocar comparators em TreeSet necessita que o método compare
 * , do Comparator, não dê como resultado zero (caso em que o Set não permite ficar
 * com elementos repetidos). Claro que isto nem sempre é possível...
 *
 * Uma forma de permitir continuar a ter repetições é assumir que a estrutura de
 * dados é uma lista e utilizar o método sort (método de classe de Collections)
 * passando como parâmetro um comparator.
 */

public List<Hotel> ordenarHoteisList(Comparator<Hotel> c) {
    List<Hotel> l = new ArrayList<Hotel>();

    l = this.hoteis.values().stream().
                            map(Hotel::clone).
                            collect(Collectors.toList());
    l.sort(c);
    return l;
}
```

com recurso a sorted

```
/**
 * Com a utilização de Stream pode efectuar-se directamente sobre a stream a
 * ordenação. Através do método sorted(), para a ordem natural, ou do método
 * sorted(c) para um comparador definido.
 */


public List<Hotel> ordenarHoteisListStream(Comparator<Hotel> c) {
 return this.hoteis.values().stream()
            .map(Hotel::clone).sorted(c).collect(Collectors.toList());

}
```

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
  - a este nível já vimos 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):

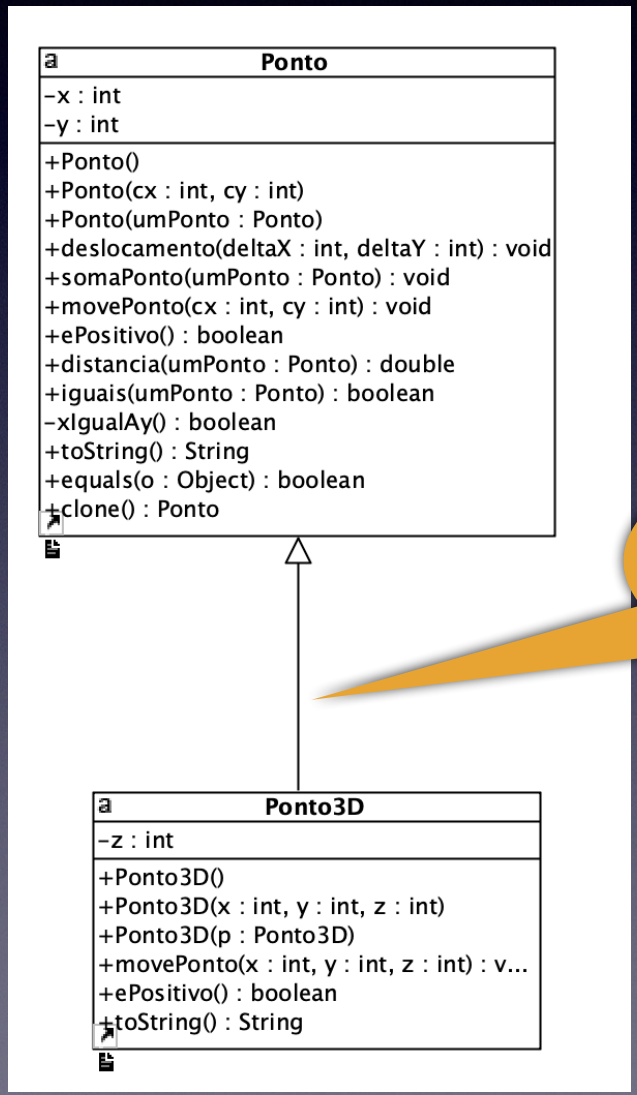
```java
/**
 * Classe que implementa um Ponto num plano2D.
 * As coordenadas do Ponto são inteiras.
 *
 * @author  MaterialPOO
 * @version 20180212
 */
public class Ponto {

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

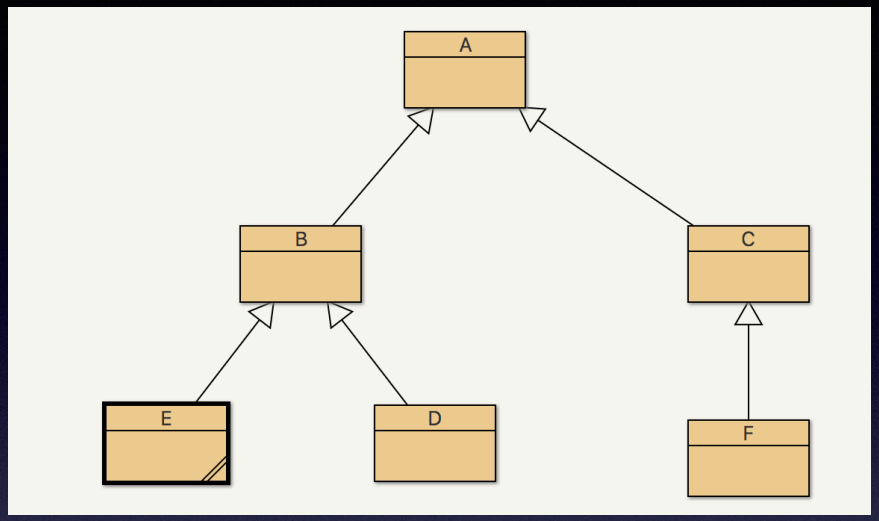
Diagrama de classe com herança



Denota a relação de herança e de especialização

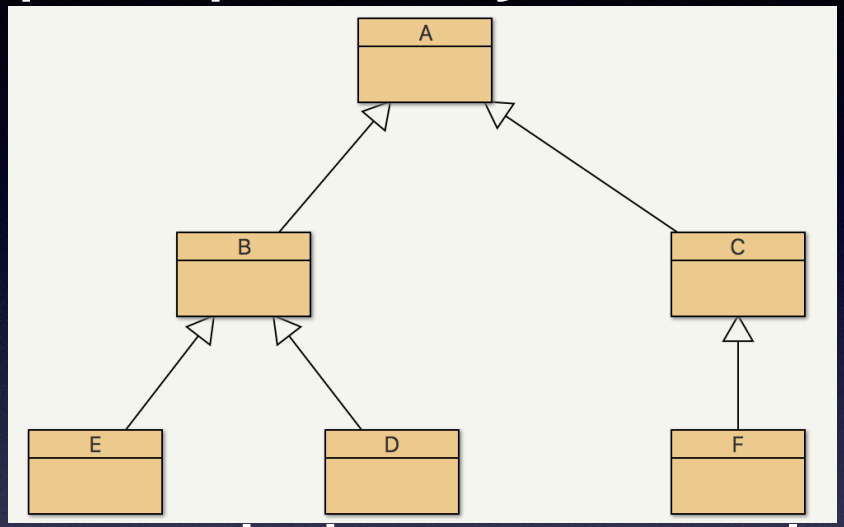
```java
/**
 * Classe que representa um Ponto 3D.
 * @author MaterialPOO
 * @version 20180323
 */
public class Ponto3D extends Ponto {
    private int z;

    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();
    }
```

- 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:

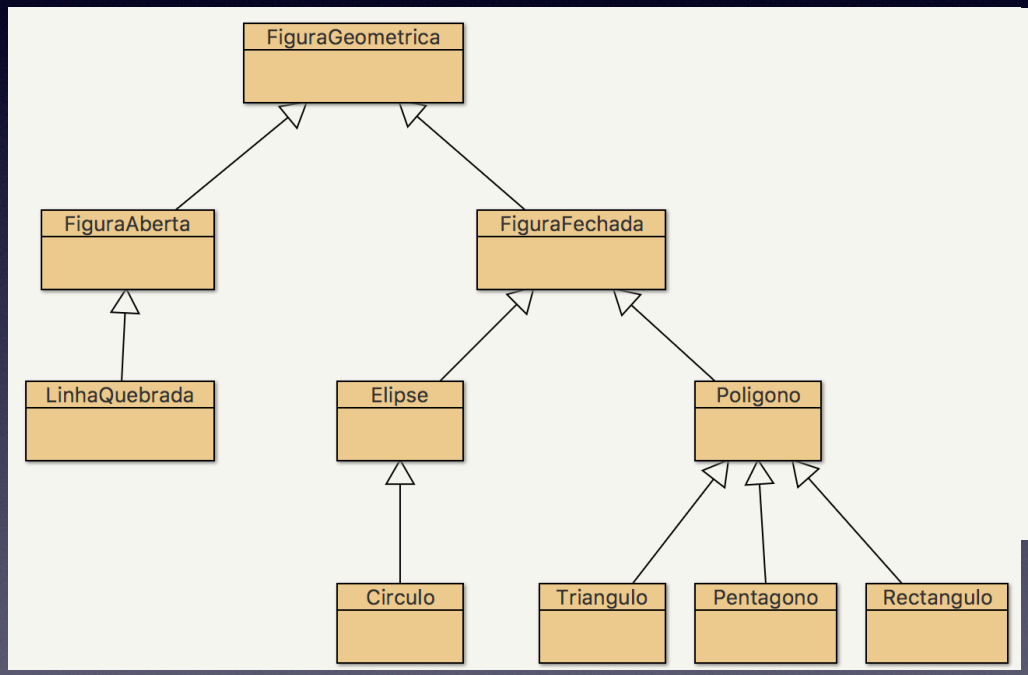


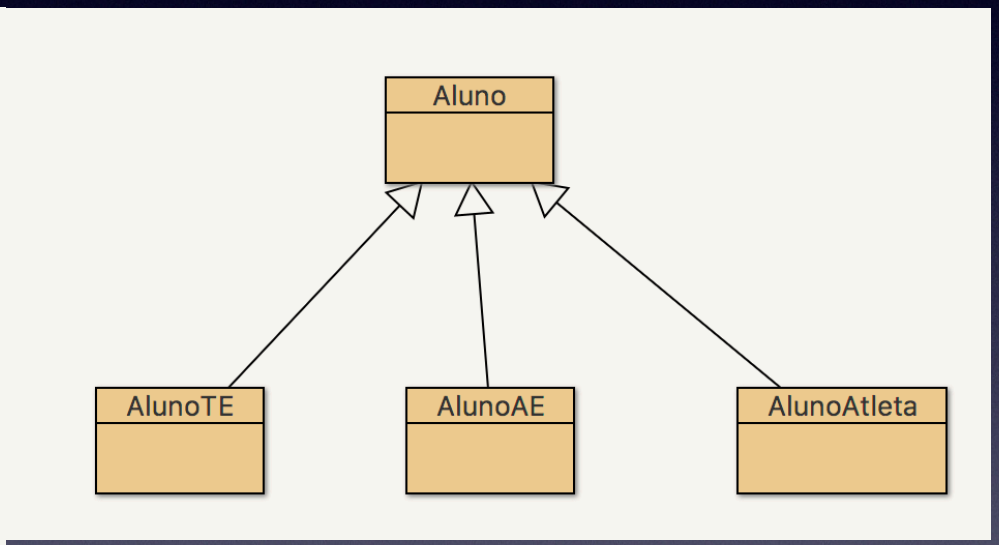
  - hierarquia de herança simples (por oposição, p.ex., a C++)

- 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

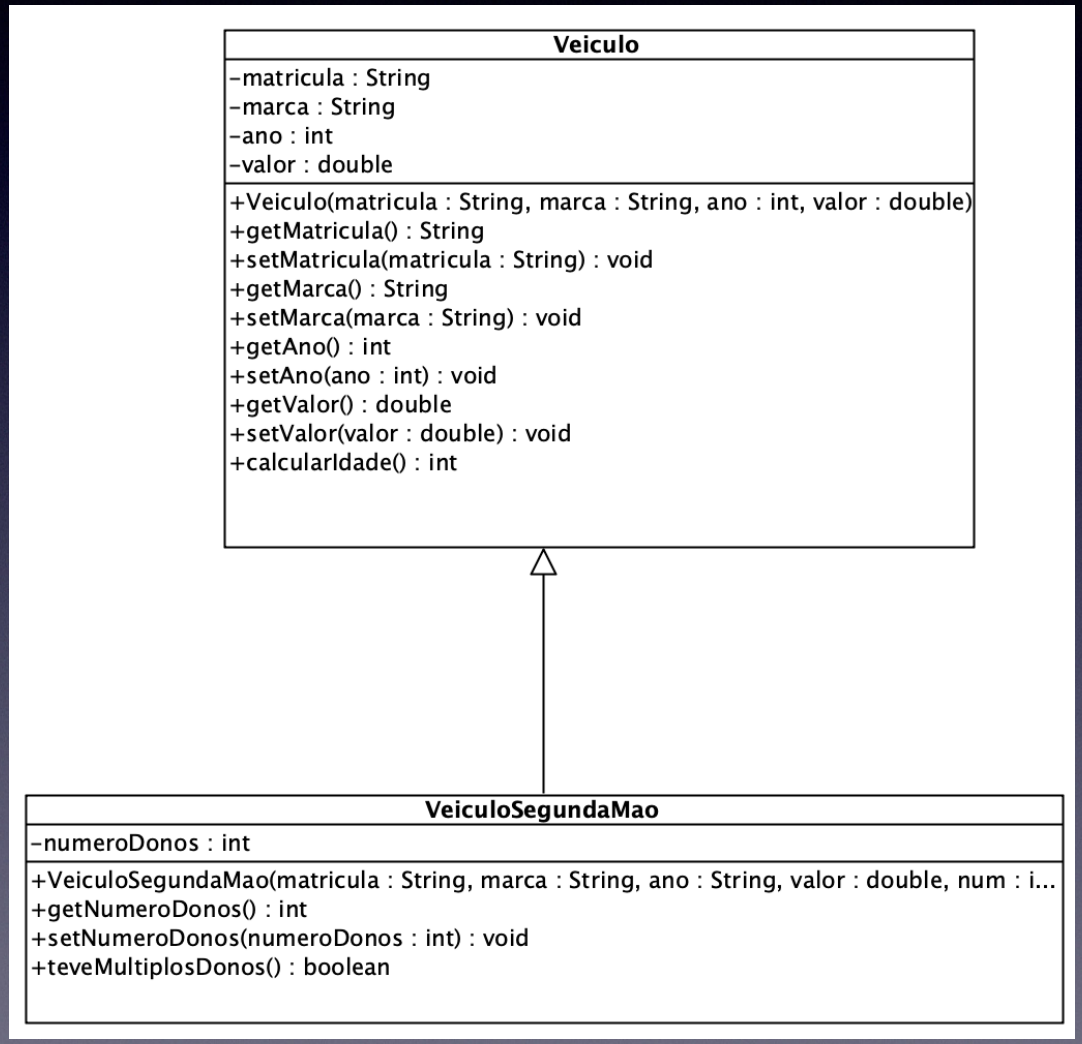
- A exemplo de outras taxonomias, a classificação do conhecimento é realizada do geral para o particular

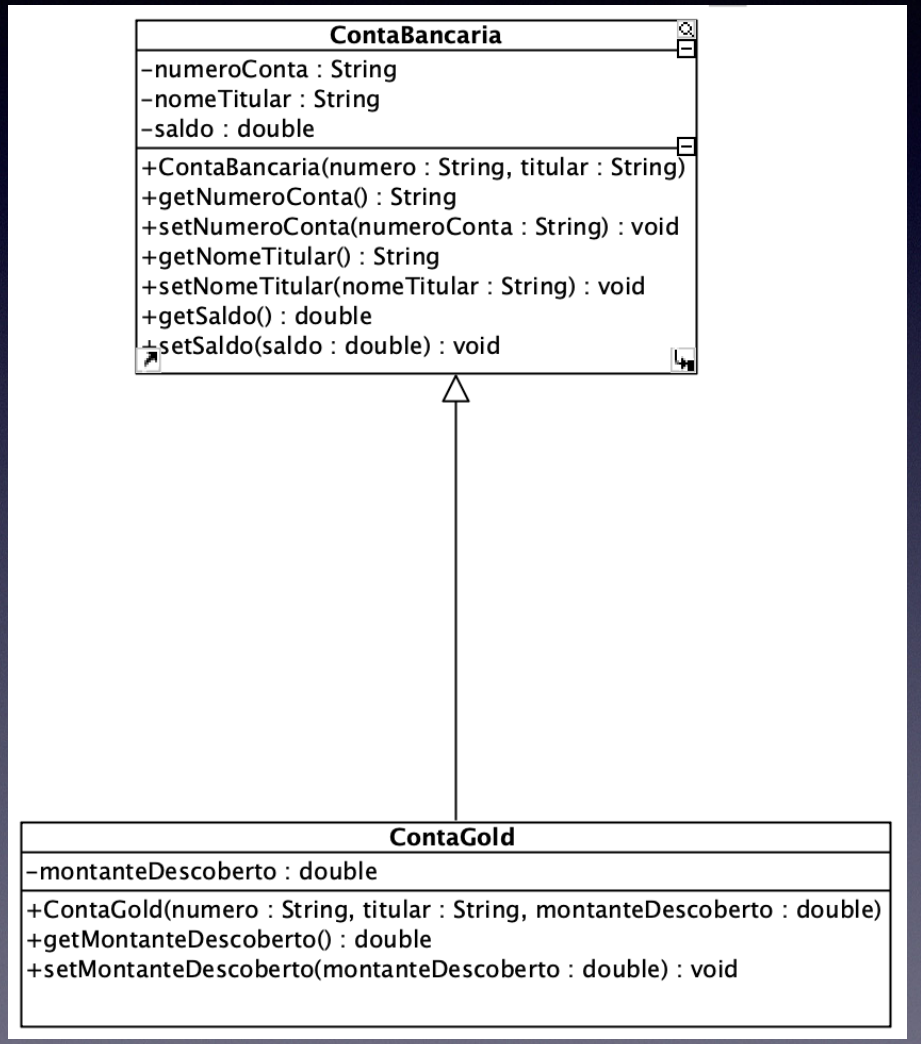




- a especialização pode ser feita nas duas vertentes: estrutural (variáveis) e comportamental (métodos)

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

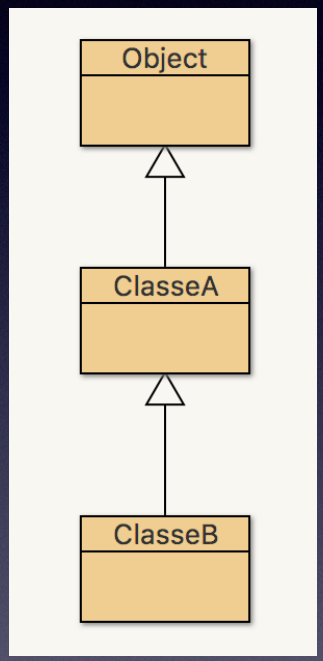
- Para percebermos a dinâmica do mecanismo de herança, vamos prestar especial atenção aos seguintes aspectos:
  - criação de instâncias das subclasses
  - redefinição de variáveis e métodos
  - procura de métodos

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

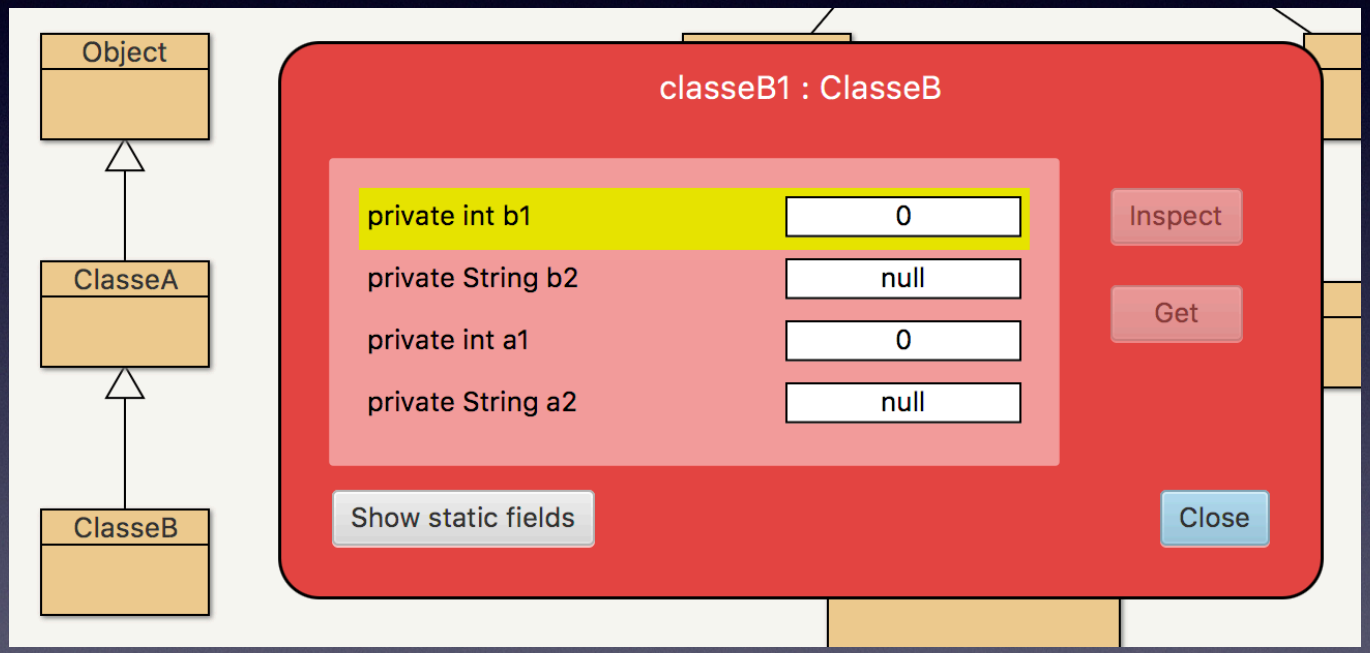
- em Java, para que seja possível a invocação do construtor de uma superclasse, esta deve ser feita logo no início do construtor da subclasse
  - recorrendo a **super(...)**, em que a verificação do construtor a invocar se faz pelo matching dos parâmetros e respectivos tipos de dados
- de facto a invocação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

- 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:



- Exemplo de criação de um objecto da classe Ponto3D