Mais sobre Collectors

`static <T> Collector<T,?,List<T>>`	**`toList()`** Returns a `Collector` that accumulates the input elements into a new `List`.
`static <T> Collector<T,?,Set<T>>`	**`toSet()`** Returns a `Collector` that accumulates the input elements into a new `Set`.
`static <T,C extends Collection<T>> Collector<T,?,C>`	**`toCollection(Supplier<C> collectionFactory)`** Returns a `Collector` that accumulates the input elements into a new `Collection`, in encounter order.
`static <T,K,U> Collector<T,?,Map<K,U>>`	**`toMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper)`** Returns a `Collector` that accumulates elements into a `Map` whose keys and values are the result of applying the provided mapping functions to the input elements.
`static <T,K,U,M extends Map<K,U>> Collector<T,?,M>`	**`toMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction, Supplier<M> mapSupplier)`** Returns a `Collector` that accumulates elements into a `Map` whose keys and values are the result of applying the provided mapping functions to the input elements.

Mais sobre *reduce (aka fold)*

reduce pré-definido

```
double sum = alunos.stream().mapToDouble(Aluno::getNota).sum();
```

`Optional<T>`	**`reduce(BinaryOperator<T>`** accumulator**`)`** Performs a **reduction** on the elements of this stream, using an **associative** accumulation function, and returns an `Optional` describing the reduced value, if any.
`T`	**`reduce(T`** identity, **`BinaryOperator<T>`** accumulator**`)`** Performs a **reduction** on the elements of this stream, using the provided identity value and an **associative** accumulation function, and returns the reduced value.
`<U> U`	**`reduce(U`** identity, **`BiFunction<U,? super T,U>`** accumulator, **`BinaryOperator<U>`** combiner**`)`** Performs a **reduction** on the elements of this stream, using the provided identity, accumulation and combining functions.

```
OptionalDouble sum = alunos.stream().mapToDouble(Aluno::getNota).reduce((ac, v) -> ac+v);
```

```
double sum = alunos.stream().mapToDouble(Aluno::getNota).reduce(0.0, (ac, v) -> ac+v);
```

```
double sum = alunos.stream().reduce(0.0,
                                    (ac, al) -> ac+al.getNota(),
                                    (ac1, ac2) -> ac1+ac2);
```

Mais sobre Optional

Optional<T>

- **OptionalDouble**
- **OptionalInt**
- **OptionalLong**

Alguns métodos relevantes...

T	**get**() If a value is present in this Optional, returns the value, otherwise throws NoSuchElementException.
boolean	**isPresent**() Return true if there is a value present, otherwise false.
T	**orElseGet**(**Supplier**<? extends T> other) Return the value if present, otherwise invoke other and return the result of that invocation.
T	**orElse**(T other) Return the value if present, otherwise return other.

Exemplo

```java
/**
 * Média da turma
 *
 * @return um double com a média da turma (zero se turma vazia)
 */
public double media() {
    OptionalDouble res = alunos.stream()
                               .mapToDouble(Aluno::getNota)
                               .average();
    return res.orElse(0.0);
}
```

Simplificando...

```java
/**
 * Média da turma
 *
 * @return um double com a média da turma (zero se turma vazia)
 */
public double media() {
    return alunos.stream()
                 .mapToDouble(Aluno::getNota)
                 .average()
                 .orElse(0.0);
}
```

Mais sobre mutable redutions

```
<R,A> R  collect(Collector<? super T,A,R> collector)
         Performs a mutable reduction operation on the elements of this stream using a Collector.

<R> R    collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner)
         Performs a mutable reduction operation on the elements of this stream.
```

```
public Set<String> getNomes() {
    return alunos.stream()
                 .map(Aluno::getNome)
                 .collect(Collectors.toSet());
}
```

Não temos garantia sobre o tipo de Set

```
public Set<Aluno> getAlunos() {
    return alunos.stream()
                 .map(Aluno::clone)
                 .collect(Collectors.toCollection(TreeSet::new));
}
```

Podemos indicar que tipo de colecção pretendemos

```
public Set<String> getNomes() {
    return alunos.stream()
                 .collect(HashSet::new, (s, al) -> s.add(al.getNome()), HashSet::addAll);
}
```

Fazer a transformação durante o collect... (desnecessariamente complicado, neste caso)

Exemplo

```
public Map<Integer,Aluno> getAlunos() {
    Map<Integer, Aluno> copia = new HashMap<>();

    for (Map.Entry<Integer,Aluno> e: alunos.entrySet())
        copia.put(e.getKey(), e.getValue().clone());

    return copia;
}
```

vs

```
public Map<Integer,Aluno> getAlunos() {
    return alunos.entrySet()
                 .stream()
                 .collect(Collectors.toMap((e)->e.getKey(),
                                           (e)->e.getValue().clone()));


}
```

… ainda sobre Collectors

`static <T,K> Collector<T,?,Map<K,List<T>>>`	**`groupingBy(Function<? super T,? extends K> classifier)`** Returns a `Collector` implementing a "group by" operation on input elements of type `T`, grouping elements according to a classification function, and returning the results in a `Map`.
`static <T,K,A,D> Collector<T,?,Map<K,D>>`	**`groupingBy(Function<? super T,? extends K> classifier, Collector<? super T,A,D> downstream)`** Returns a `Collector` implementing a cascaded "group by" operation on input elements of type `T`, grouping elements according to a classification function, and then performing a reduction operation on the values associated with a given key using the specified downstream `Collector`.
`static <T,K,D,A,M extends Map<K,D>> Collector<T,?,M>`	**`groupingBy(Function<? super T,? extends K> classifier, Supplier<M> mapFactory, Collector<? super T,A,D> downstream)`** Returns a `Collector` implementing a cascaded "group by" operation on input elements of type `T`, grouping elements according to a classification function, and then performing a reduction operation on the values associated with a given key using the specified downstream `Collector`.

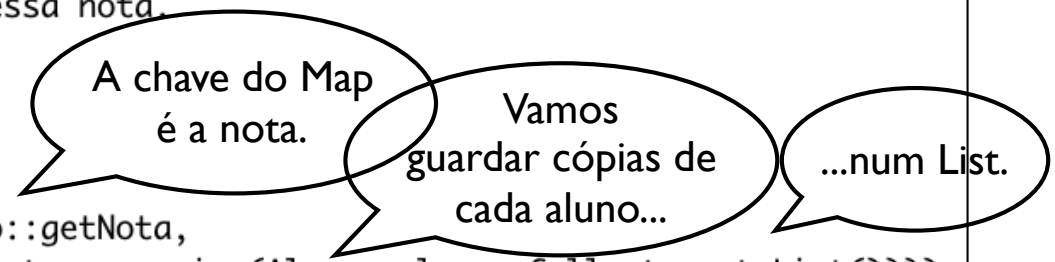
Exemplos

```
/**
 * Calcular um Map de nota para lista de alunos com essa nota.
 */
public Map<Double, List<Aluno>> porNota() {
    Map<Double, List<Aluno>> res = new TreeMap<>();

   for(Aluno a : alunos.values()) {
        double nota = a.getNota();
        if (!res.containsKey(nota))
            res.put(nota, new ArrayList<>());
        res.get(nota).add(a.clone());
    }
    return res;
}
```

vs

```
/**
 * Calcular um Map de nota para lista de alunos com essa nota.
 */
public Map<Double, List<Aluno>> porNota() {
    return alunos.values()
                 .stream()
                 .collect(Collectors.groupingBy(Aluno::getNota,
                                                Collectors.mapping(Aluno::clone, Collectors.toList())));
}
```



Exemplos

A utilização de import static permite simplificar as expressões, eliminando a necessidade de 'Collectors.'.

```
/**
* Calcular um TreeMap de nota para Set de nomes dos alunos com essa nota.
*/
public TreeMap<Double, Set<String>> nomesPorNota() {


    return alunos.values()
                 .stream()
                 .collect(groupingBy(Aluno::getNota,
                                     TreeMap::new,
                                     mapping(Aluno::getNome, toSet())));
}
```

A chave do Map é a nota

Queremos um TreeMap

Queremos guardar o nome dos alunos

Os nomes vão ser guardados num Set

Side effects

Em geral, devem evitar-se efeitos laterais nos métodos/expressões lambda utilizados nas Streams

vão complicar paralelização das streams no futuro

forEach() (e **peek()**) operam via efeitos laterais pelo que deve ser utilizados com cuidado

Em muitos casos, os efeitos laterais podem ser evitados...

```
ArrayList<String> results = new ArrayList<>();
stream.filter(s -> pattern.matcher(s).matches())
      .forEach(s -> results.add(s));  // Unnecessary use of side-effects!
```

```
List<String>results =
    stream.filter(s -> pattern.matcher(s).matches())
          .collect(Collectors.toList());  // No side-effects!
```

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."*


TOY MICE
CLOCKWORK MOTORS
COMPONENT PARTS
ATOMS
ETC. ETC

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

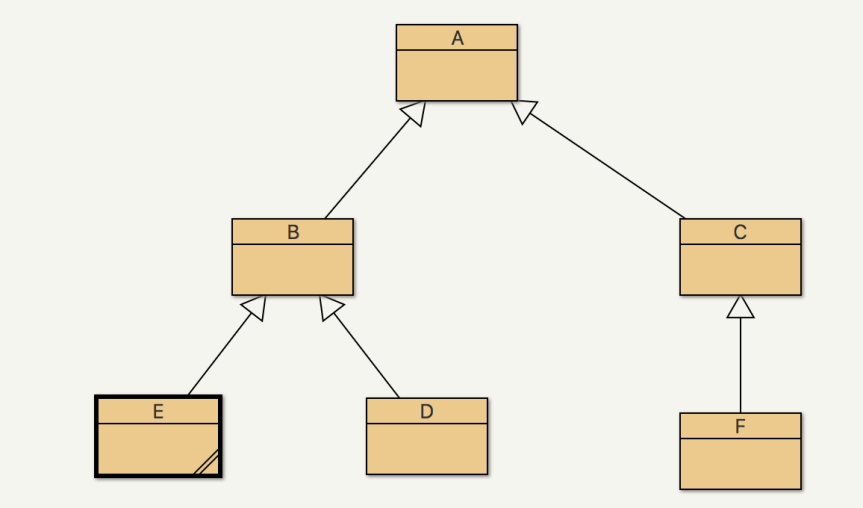
```java
/**
 * Classe que representa um Ponto 3D.
 * @author MaterialPOO
 * @version 20180323
 */
public class Ponto3D extends Ponto {
    private int z;

    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.

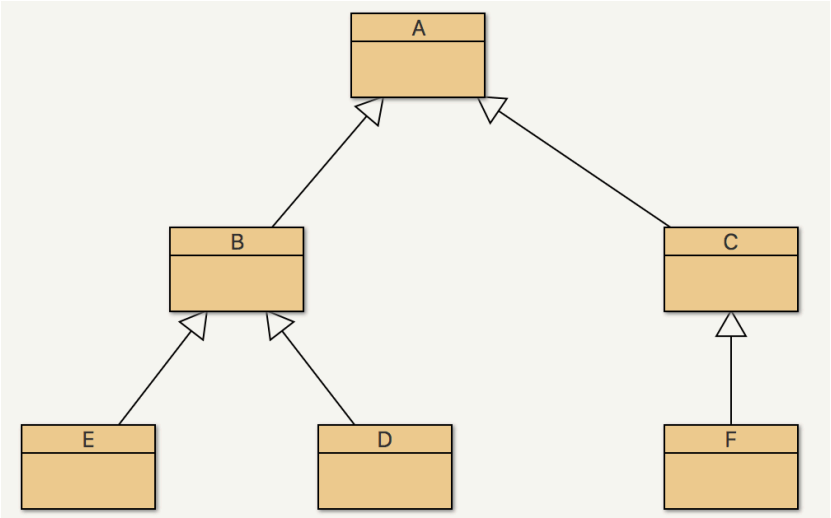
B é superclasse de D e E

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++)

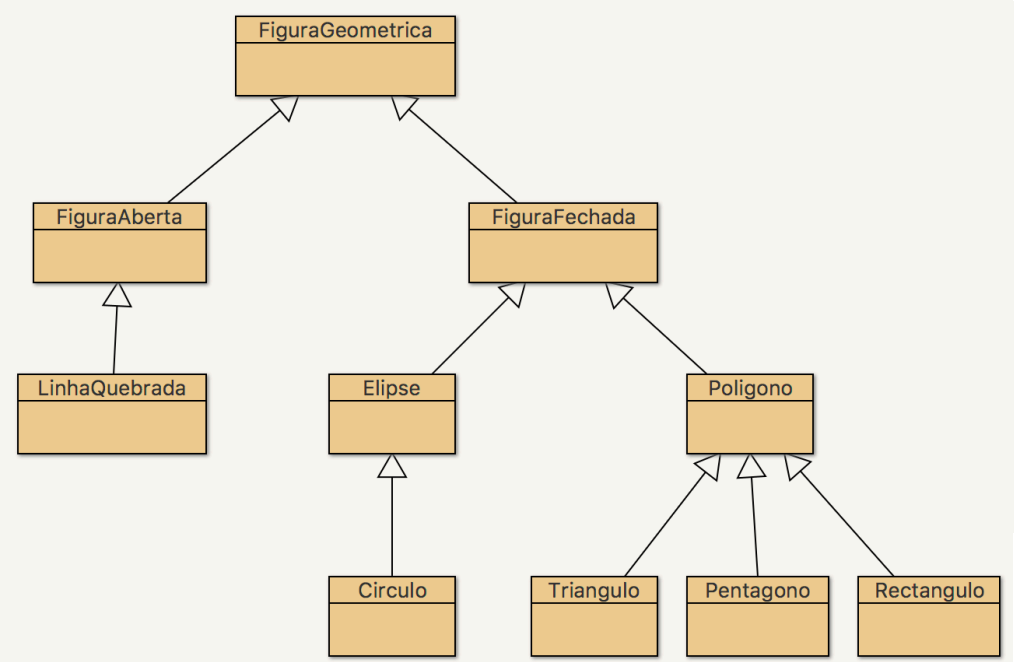
O que significa do ponto de vista semântico dizer que duas classes estão hierarquicamente relacionadas?

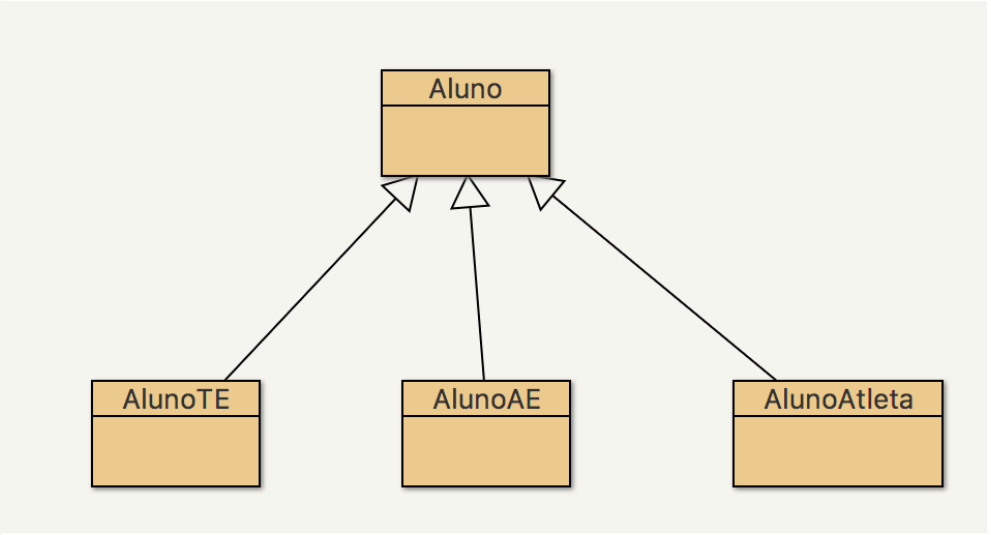
no paradigma dos objectos a hierarquia de classes é uma hierarquia de especialização

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

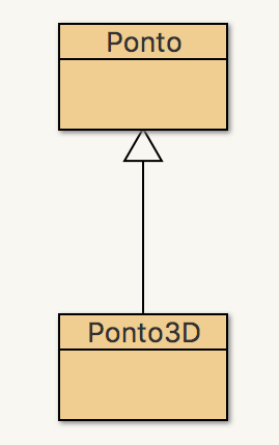
A exemplo de outras taxonomias, a classificação do conhecimento é realizada do geral para o particular





a especialização pode ser feita nas duas vertentes: estrutural e comportamental

voltando ao Ponto3D:

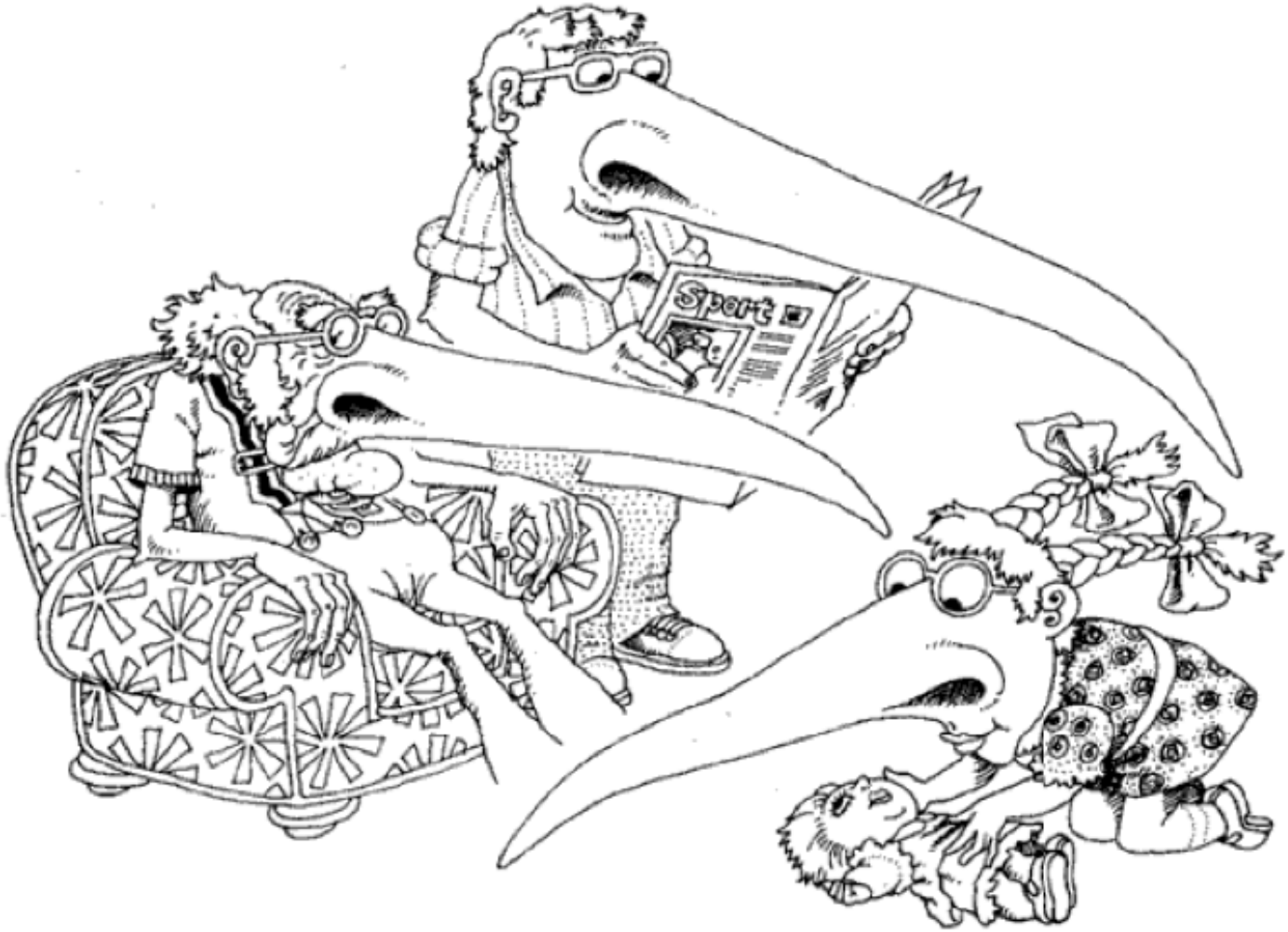


ou seja, Ponto3D é subclasse de Ponto

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

é subclasse!

como foi dito, uma subclasse herda estrutura e comportamento da sua classe:



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 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 nossas 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 de que herdamos pelo nome

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

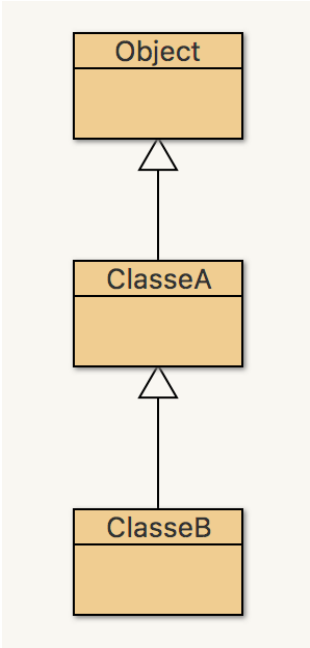
- como vimos atrás os construtores não são herdados

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 os construtores 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: