# 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 das características mais importantes 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 (exemplo da Ficha 3) é composto por duas instâncias de Ponto2D
  - 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 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 Ponto2D?

    - mais uma v.i. e métodos associados

- A classe Ponto2D (incompleta):

```
public class Ponto2D {
   // Construtores
   public Ponto2D(int cx, int cy) { x = cx; y = cy; }
   public Ponto2D(){ this(0, 0); }
   // Variáveis de Instância
   private int x, y;
   // Métodos de Instância
   public int getX() { return x; }
   public int getY() { return y; }
   public void desloca(int dx, int dy) {
      x += dx; y += dy;
   }
   public void somaPonto(Ponto2D p) {
      x += p.getX(); y += p.getY();
   }
   public Ponto2D somaPonto(int dx, int dy) {
      return new Ponto2D(x += dx, y+= dy);
   }
   public String toString() {
      return new String("Pt= " +x + ", " + y);
   } }
```

- 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

$$\text{Ponto3D} = \text{Ponto2D} + \Delta_{prog}$$
$$1\ \text{ponto3D} \Leftrightarrow 1\ \text{ponto2D} + \Delta_{var} + \Delta_{met}$$

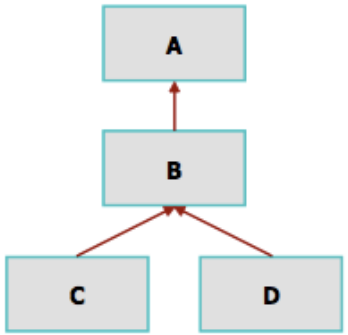
- ou seja, para ter um Ponto3D precisamos de tudo o que existe em Ponto2D e acrescentar um *delta* que consiste nas características novas

- A classe Ponto3D aumenta, refina, detalha, especializa, a classe Ponto2D

- 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 Ponto2D
  - através de um mecanismo existente de base nas linguagens por objectos que é a noção de hierarquia e herança

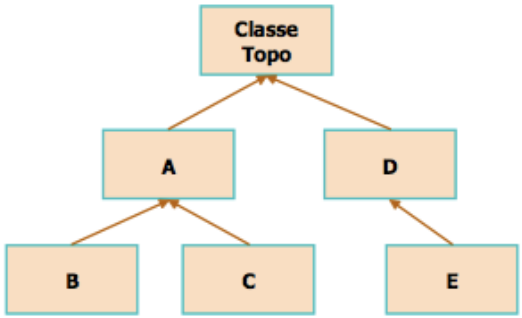
```java
/**
 * Ponto3D através de composição.
 */

public class Ponto3D {
  private Ponto2D p;
  private int z;

  public Ponto3D() {
    this.p = new Ponto2D();
    this.z = 0;
  }

  public Ponto3D(int x, int y, int z) {
    this.p = new Ponto2D(x,y);
    this.z = z;
  }

  //...
  public int getX() {return this.p.getX();}
  public int getY() {return this.p.getY();}
  public int getZ() {return this.z;}
```

- Exemplo:




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

- Hierarquia típica em Java:

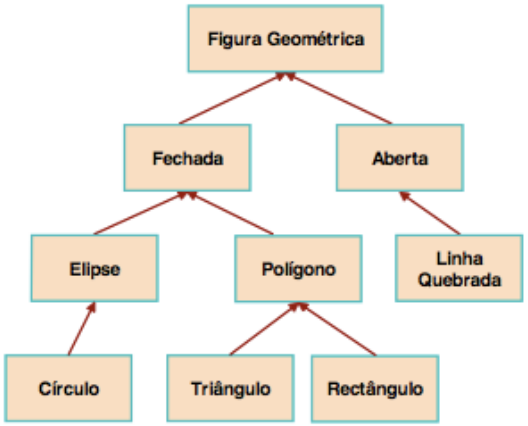



  - hierarquia de herança simples (por oposição, p.ex., a C++)

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

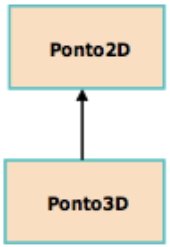
- no paradigma dos objectos a hierarquia de classes é uma hierarquia de especialização
  - uma subclasse de uma dada classe constitui uma especialização, sendo por definição mais detalhada que a classe que lhe deu origem
  - isto é, possui **mais** estado e **mais** comportamento

- 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 Ponto2D

```
public class B extends A { …
public class Ponto3D extends Ponto2D { …
```

- 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 todas as variáveis e métodos de instância que não são private.
  - se B **não pertence** ao mesmo package de A, B herda 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
- variáveis e métodos de classe não são herdados: podem ser redefinidos.
- 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?

  - total, vamos deixar de poder referir as v.i. da superclasse que herdamos pelo nome

  - vamos utilizar os métodos de acesso, *getX()*, para aceder aos seus valores

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

# 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 transitivas 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 declarada na subclasse diz-se que 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 Super

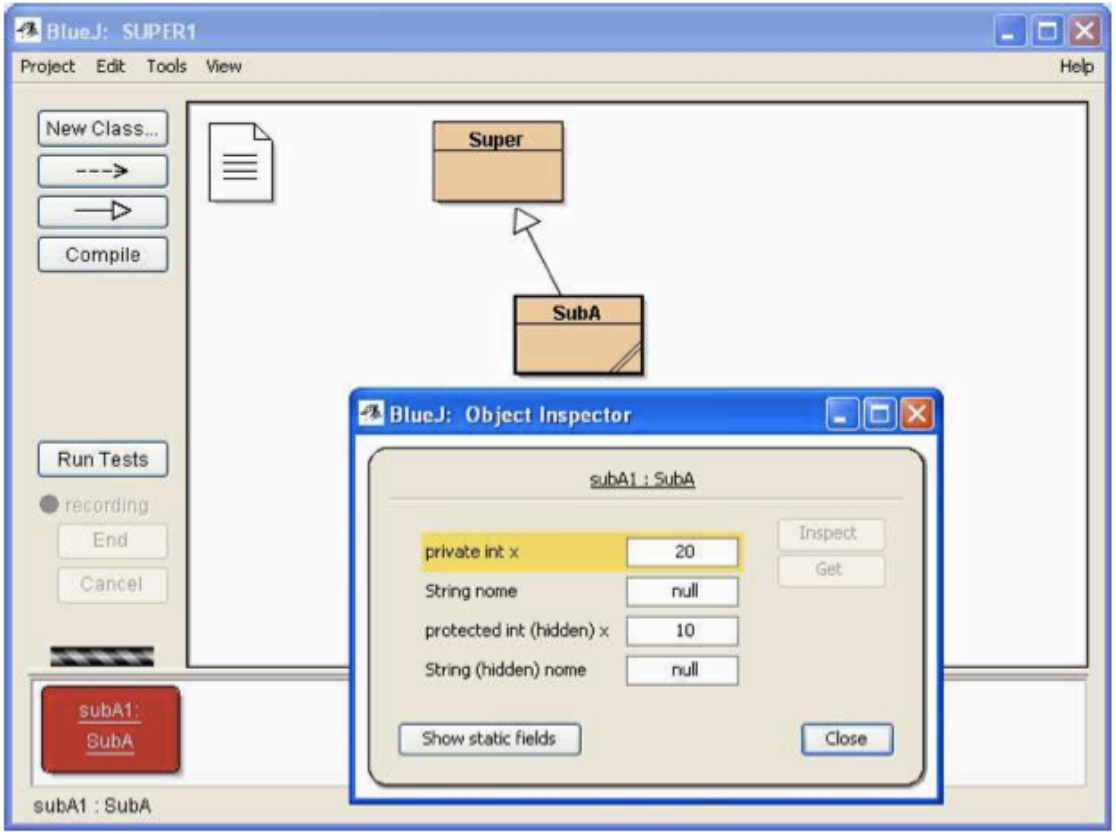
```
public class Super {
   protected int x = 10;
   String nome;
   // Métodos
   public int getX() { return x; }
   public String classe() { return "Super"; }
   public int teste() { return this.getX(); }
}
```

- e uma subclasse SubA

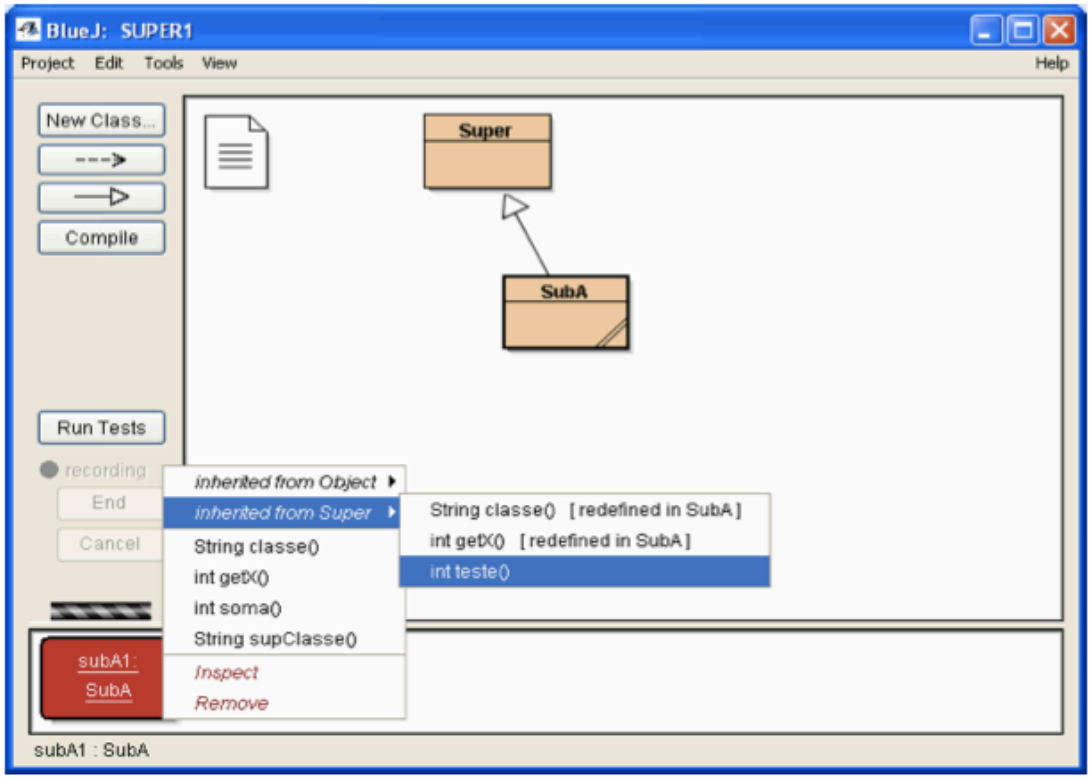
```
public class SubA extends Super {
   private int x = 20; // "shadow"
   String nome;        // "shadow"
   // Métodos
   public int getX() { return x; }
   public String classe() { return "SubA"; }
   public String supClass() { return super.classe(); }
   public int soma() { return x + super.x; }
}
```

- 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 de um objecto no BlueJ



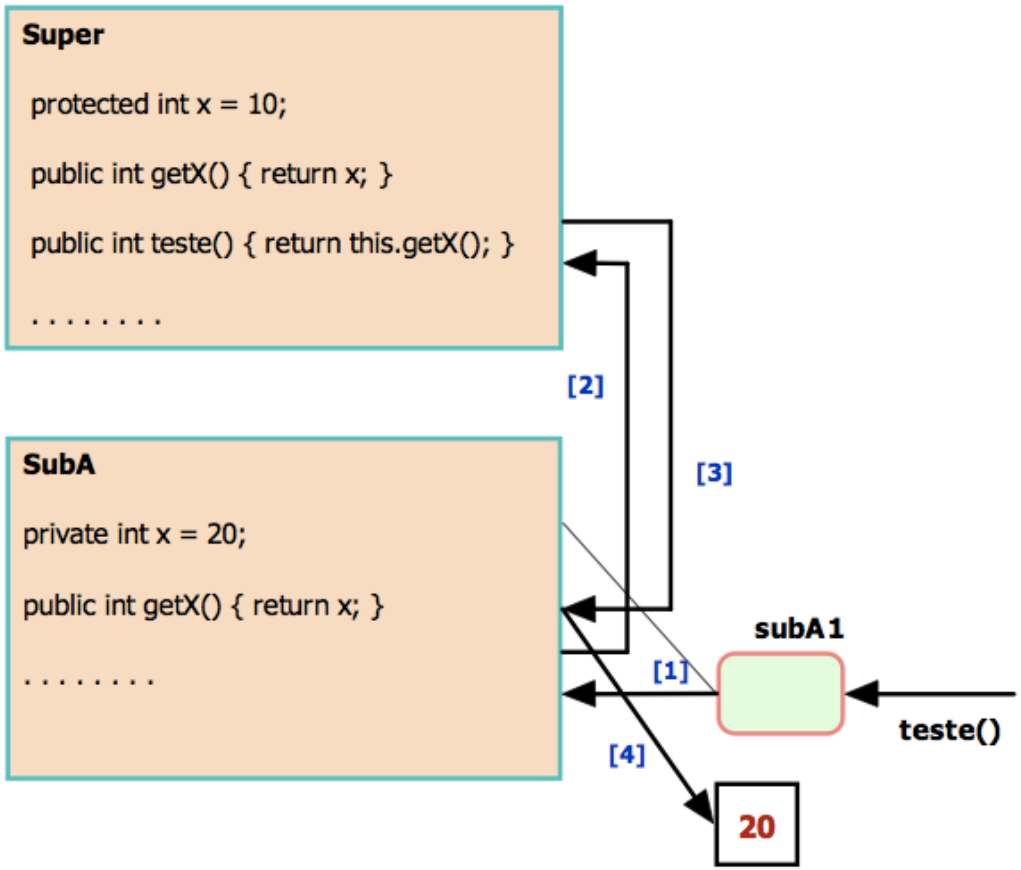
- no BlueJ é possível ver os métodos definidos na classe e os herdados



- o que acontece quando enviamos à instância subA1 (imagem anterior) a mensagem teste()?
  - teste() 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.getX()`

- em Super o valor de x é 10, enquanto que em SubA o valor de x é 20.
  - qual é o contexto de execução de this.getX()?
  - a que instância é que o this se refere?
- Vejamos o algoritmo de procura e execução de métodos...

- execução da invocação de teste()



- na execução do código, a referência a this corresponde sempre ao objecto que recebeu a mensagem
  - neste caso, subA1
- sendo assim, o método getX() é o método de SubA que é a classe do receptor da mensagem
  - independentemente do contexto "subir e descer", o `this` refere sempre o receptor da mensagem!

# Regra para avaliaçao 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