

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
- variáveis e métodos de classe são herdados mas...
 - se forem redefinidos são *hidden*, não *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?
- total, vamos deixar de poder referir as v.i. da superclasse de que herdamos pelo nome
- vamos utilizar os métodos de acesso, *getX()*, para aceder aos seus valores

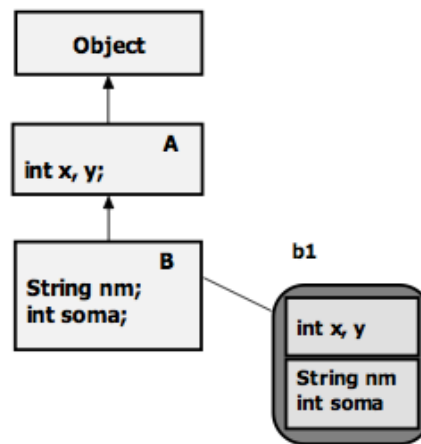
- Para percebermos a dinâmica do mecanismo de herança, vamos prestar especial atenção aos seguintes aspectos:
- criação de instâncias das subclasses
- redefinição de variáveis e métodos
- procura de métodos

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:



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

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

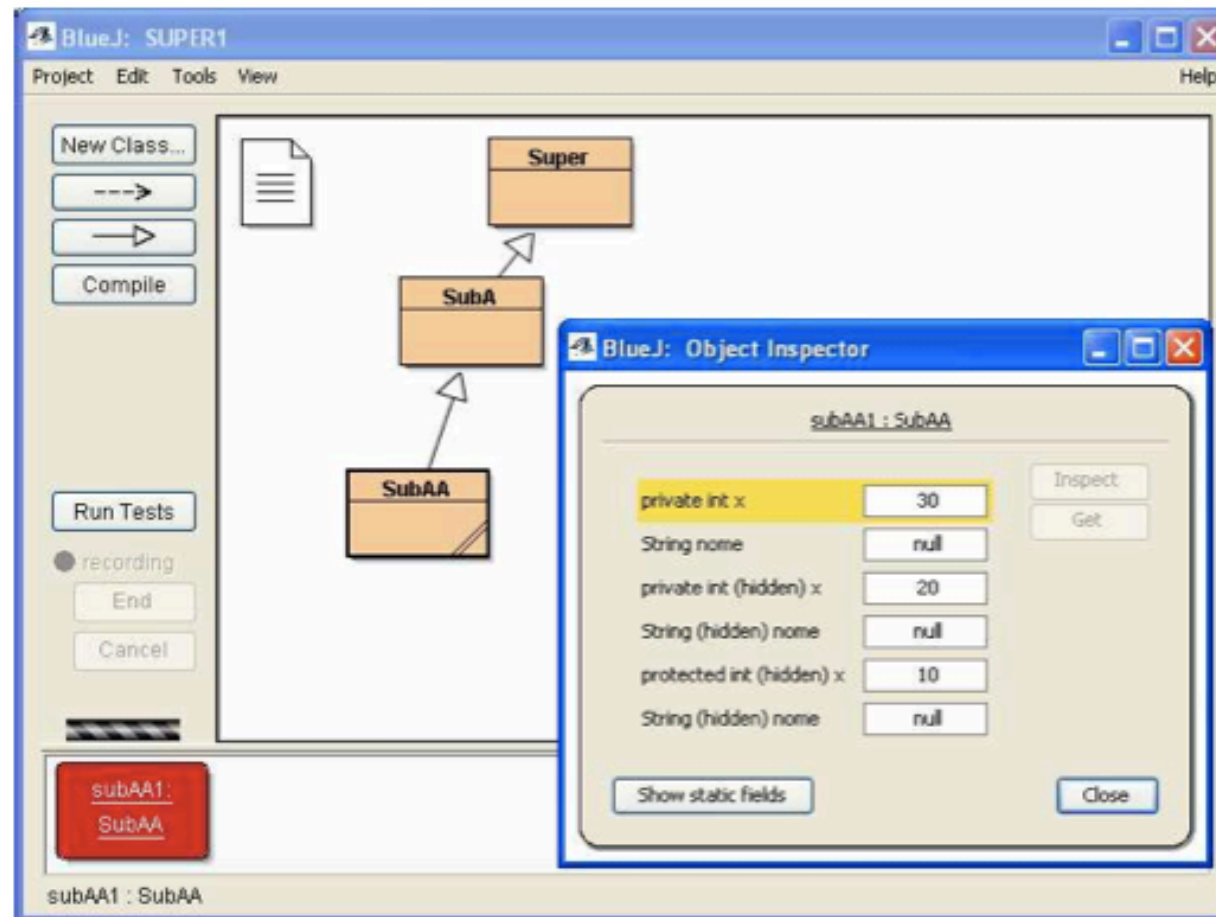
- em Java, para que seja possível a invocação do construtor de uma superclasse, esta deve ser feita logo no início do construtor da subclasse
- recorrendo a **super(...)**, em que a verificação do construtor a invocar se faz pelo matching dos parâmetros e respectivos tipos de dados
- de facto a invocação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

- Exemplo classe Pixel, subclasse de Ponto2D
- os construtores de Pixel delegam nos construtores de Ponto2D a inicialização das v.i. declaradas em Ponto2D

```
public class Pixel extends Ponto2D {  
    // Variáveis de Instância  
    private int cor;  
    // Construtores  
    public Pixel() { super(0, 0); cor = 0; }  
    public Pixel(int cor) { this.cor = cor%100; }  
    public Pixel(int x; int y; int cor) {  
        super(x, y); this.cor = cor%100;  
    }  
}
```

- 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

- Veja-se o exemplo:



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 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 Super

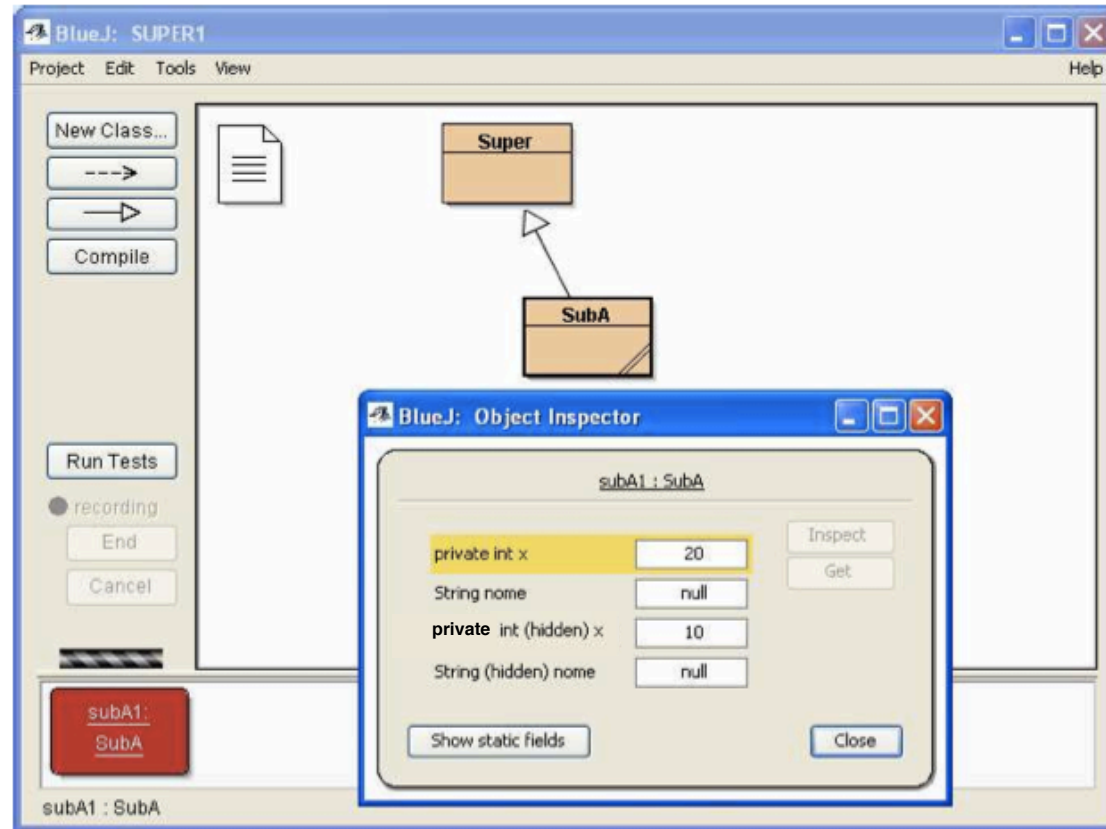
```
public class Super {  
    private    int x = 10;  
    private    String nome;  
    // Métodos  
    public int getX() { return x; }  
    public String classe() { return "Super"; }  
    public int teste() { return this.getX(); }  
}
```

- e uma subclasse SubA

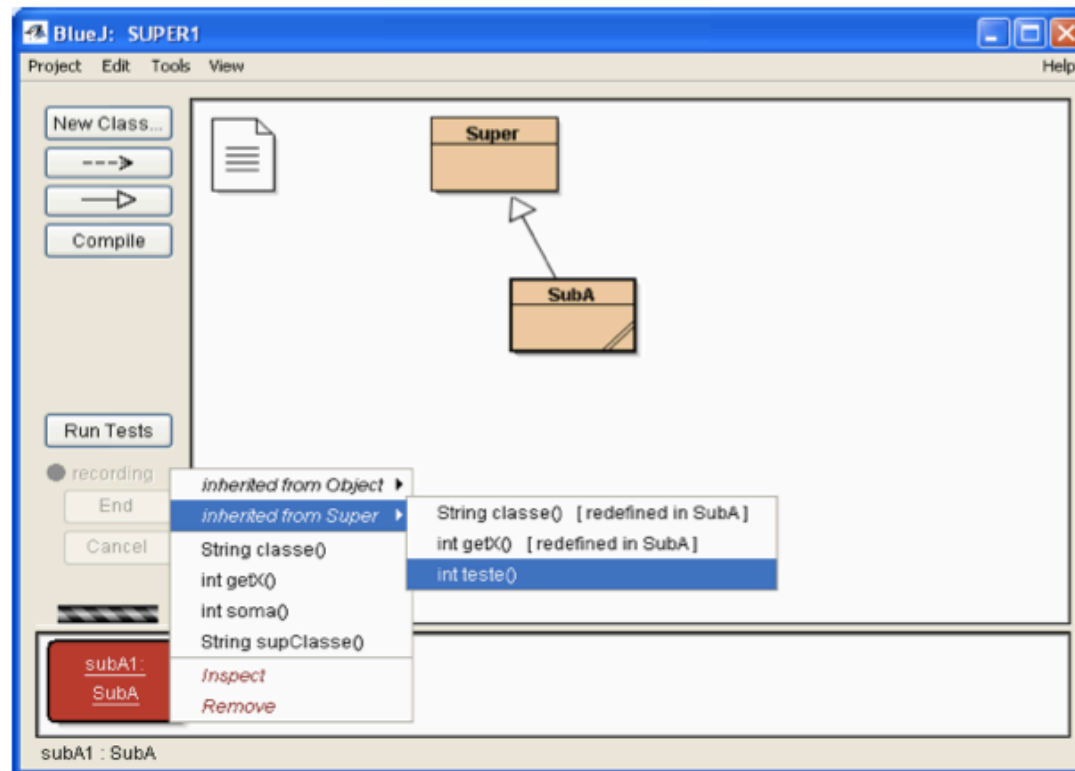
```
public class SubA extends Super {  
    private int x = 20; // "shadow"  
    private String nome; // "shadow"  
    // Métodos  
    public int getX() { return x; }  
    public String classe() { return "SubA"; }  
    public String superClass() { return super.classe(); }  
    public int soma()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


- 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



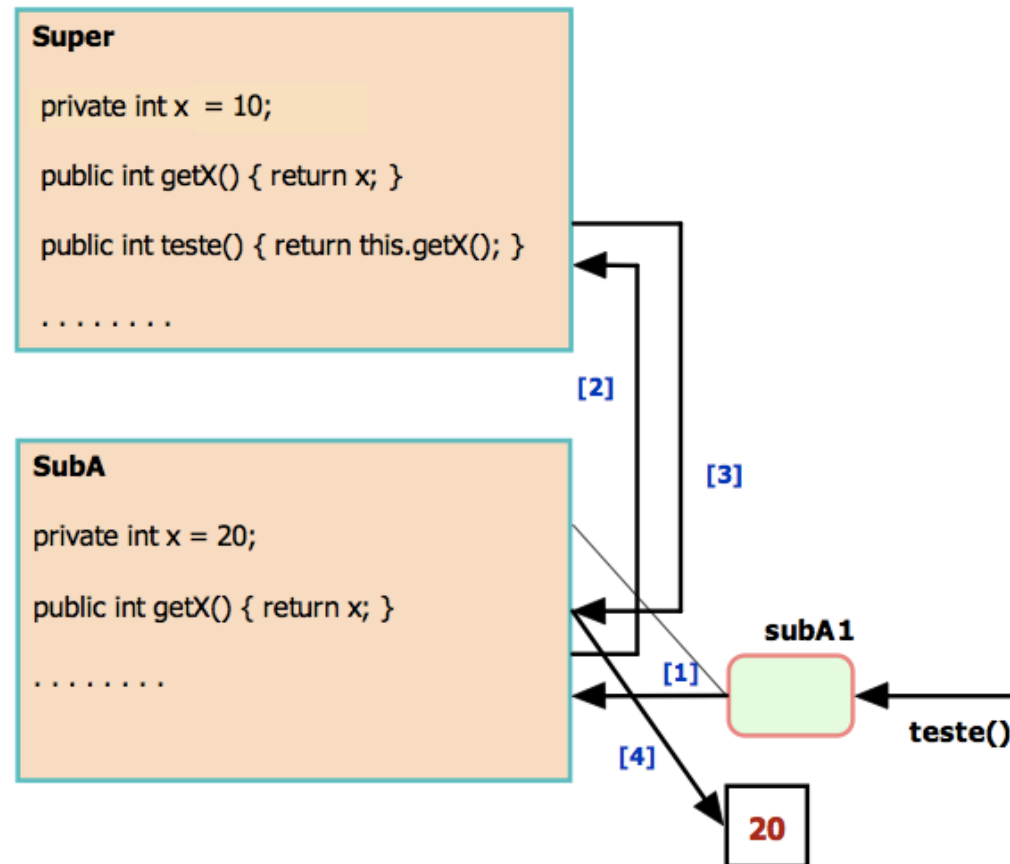
- no BlueJ é possível ver os métodos definidos na classe e os herdados



- o que acontece quando enviamos à instância subAI (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, `subA`
- 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 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

Compatibilidade entre classes e subclasses

- uma das vantagens da construção de uma hierarquia é a reutilização de código, mas...
- os aspectos relacionados com a criação de tipos de dados são também não negligenciáveis
- as classes são associadas estaticamente a tipos
 - uma classe é um tipo de dados

- é preciso saber qual a compatibilidade entre os tipos das diferentes classes (superclasses e subclasses)
- a questão importante é saber que uma classe é compatível com as suas subclasses!
- é importante reter o princípio da substituição que diz que...

- “se uma variável é declarada como sendo de uma dada classe (tipo), é legal que lhe seja atribuído um valor (instância) dessa classe ou de qualquer das suas subclasses”
- existe compatibilidade de tipos no sentido ascendente da hierarquia (eixo da generalização)
- ou seja, uma instância de uma subclasse pode ser atribuída a uma instância da superclasse (`Forma f = new Triangulo()`)

- seja o código

```
A a, al;  
a = new A(); al = new B();
```

- ambas as declarações estão correctas, tendo em atenção a declaração de variável e a atribuição de valor
- B é uma subclasse de A, pelo que está correcto
- mas o que acontece quando se executa `a1.m()`?

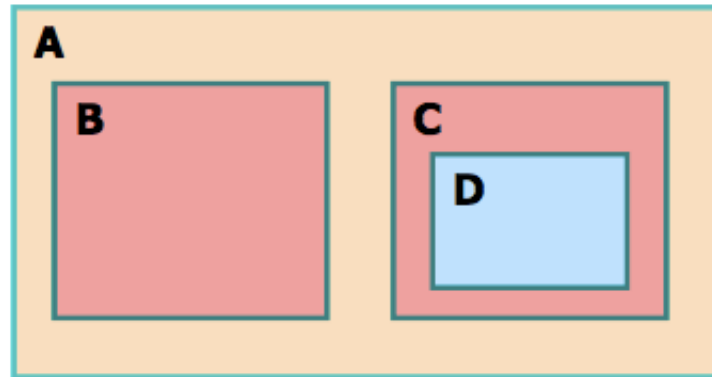
- o compilador tem de verificar se `m()` existe em `A` ou numa sua superclasse
- se existir é como se estivesse declarado em `B`
- a expressão é correcta do ponto de vista do compilador
- em tempo de execução terá de ser determinado qual é o método a ser invocado.

- o interpretador, em tempo de execução, faz o ***dynamic binding*** procurando determinar em função do valor contido qual é o método que deve invocar
- se várias classes da hierarquia implementarem o método m(), então o interpretador executa o método associado ao tipo de dados da **classe do objecto**

- Seja o seguinte código

```
public class A {
    public A() { a = 1; }
    public int daVal() { return a; }
    public void metd() { a += 10; }
}
public class B extends A {
    public B() { b = 2; }
    private int b;
    public int daVal() { return b; }
    public void metd() { b += 20 ; }
}
public class C extends A {
    public C() { c = 3; }
    private int c;
    public int daVal() { return c; }
    public void metd() { c += 30 ; }
}
public class D extends C {
    public D() { d = 33; }
    private int d;
    public int daVal() { return d; }
    public void metd() { d = d * 10 + 3 ; } }
```

- do ponto de vista dos tipos de dados especificados e da relação entre eles, podemos estabelecer as seguintes relações de inclusão:



- ou seja, um D pode ser visto como um C ou um A. Um C pode ser visto como um A, etc...

- e as seguintes inicializações de variáveis

```
import static java.lang.System.out;
public class TesteABCD {
    public static void main(String args[]) {
        A a1, a2, a3, a4;
        a1 = new A(); a2 = new B(); a3 = new C(); a4 = new D();
        a1.metd(); a2.metd(); a3.metd(); a4.metd();
        out.println("a1.metd() = " + a1.daVal());
        out.println("a2.metd() = " + a2.daVal());
        out.println("a3.metd() = " + a3.daVal());
        out.println("a4.metd() = " + a4.daVal());
    }
}
```

- qual é o resultado deste programa?

- importa agora distinguir dois conceitos muito importantes:
- tipo *estático* da variável
 - é o tipo de dados da declaração, tal como foi aceite pelo compilador
- tipo *dinâmico* da variável
 - corresponde ao tipo de dados associado ao construtor que criou a instância

- como o interpretador executa o algoritmo de procura dinâmica de métodos, executando `metd()` em cada uma das classes, então o resultado é:

