- Para percebermos a dinâmica do mecânismo 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

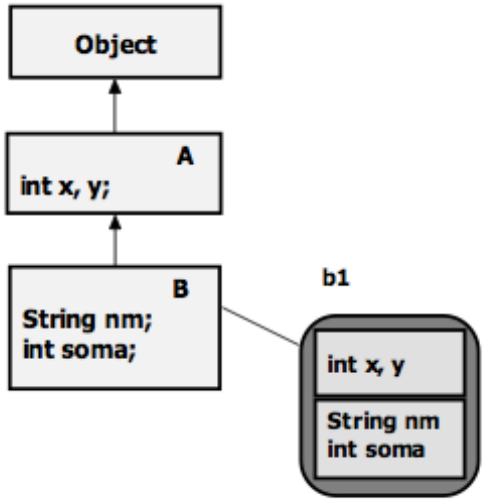
- quando temos uma subclasse B de A, sabe--se que B herda todas as v.i. de A a que tem acesso.

- 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

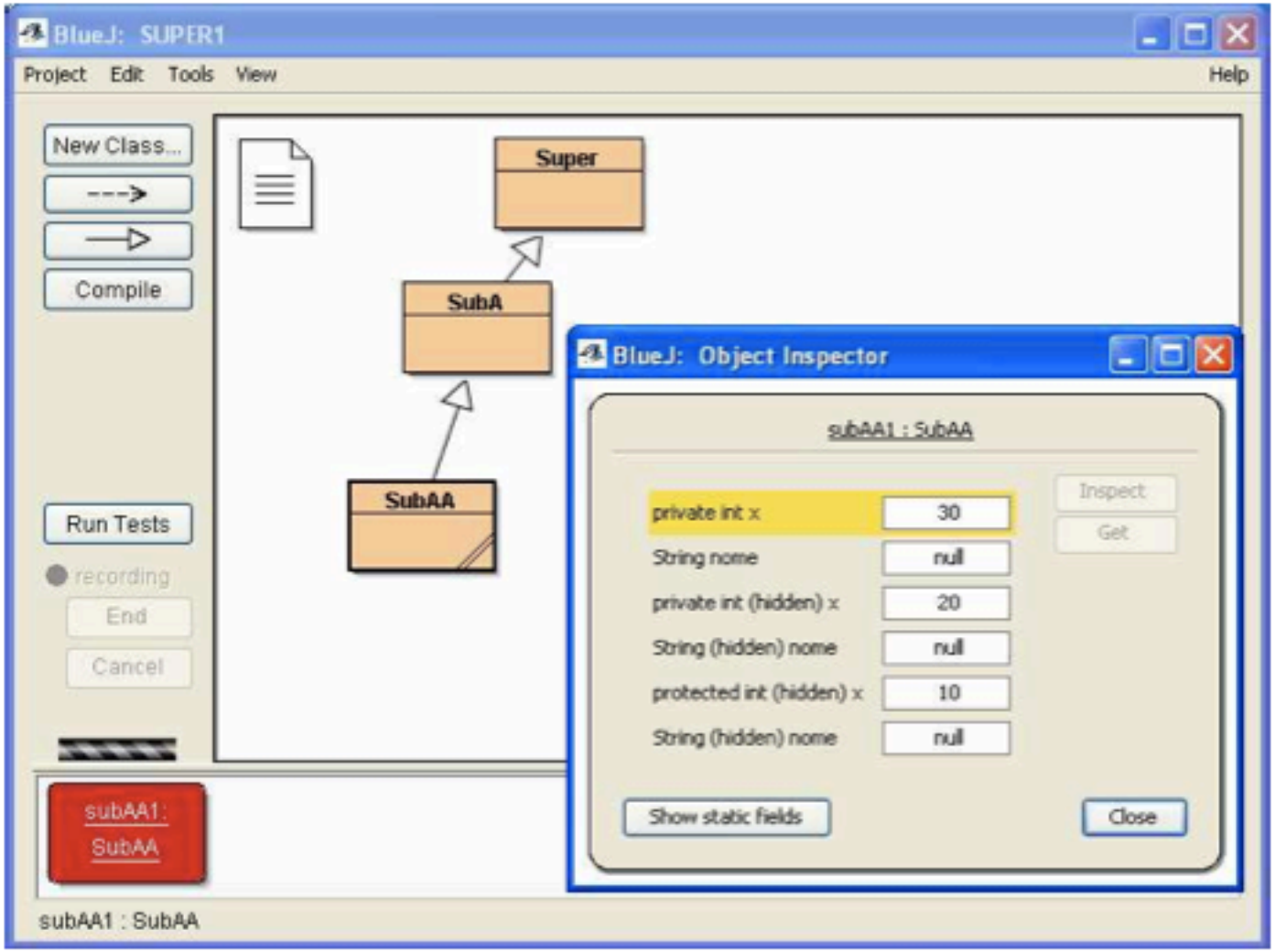
- Exemplo classe Pixel, subclasse de Ponto2D

- 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 classe 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

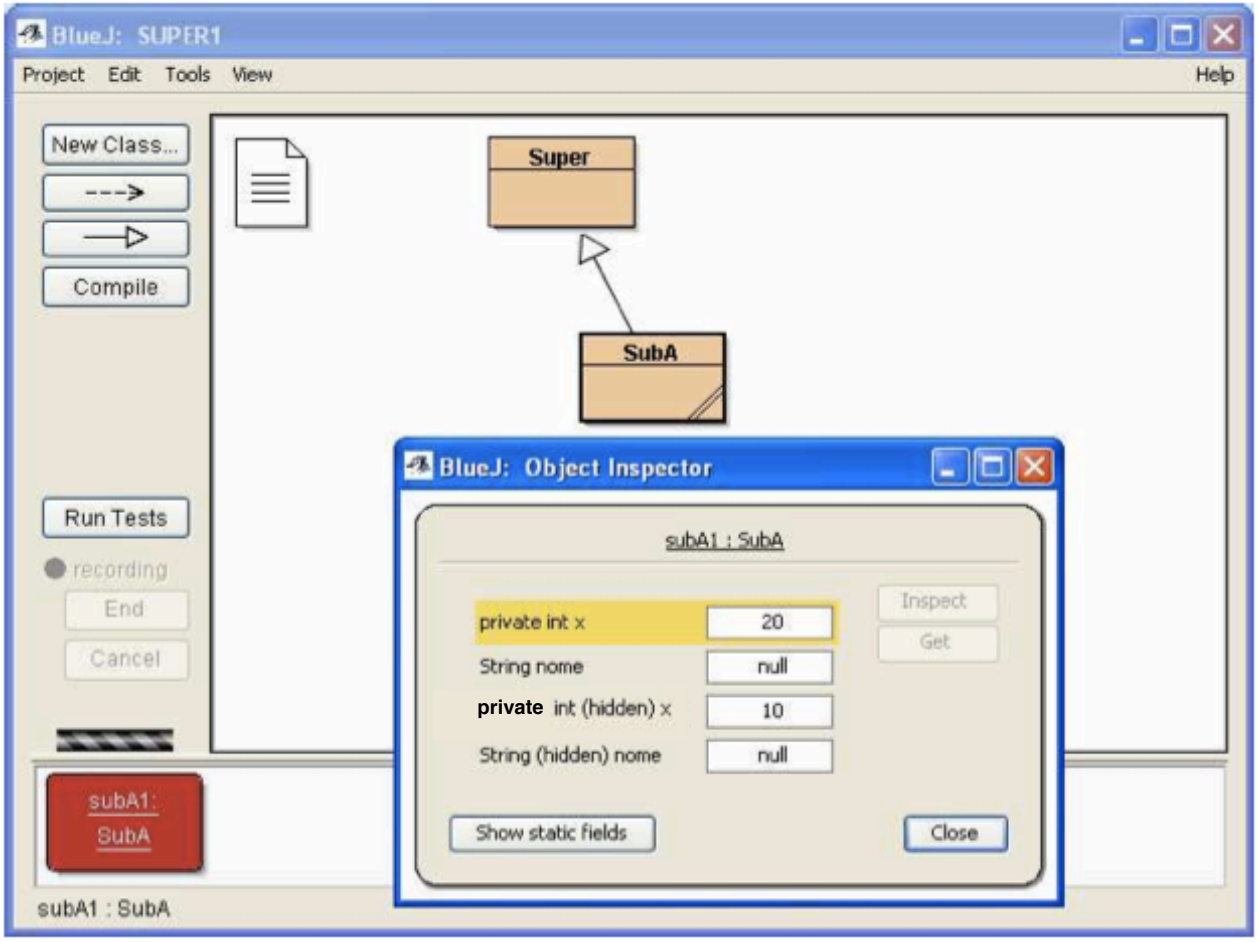
```
public class Super {
   private   int x = 10;
   private   String nome;
   // Métodos
   public int getX() { return x; }
   public String classe() { return "Super"; }
   public int teste() { return this.getX(); }
}
```

- e uma subclasse SubA

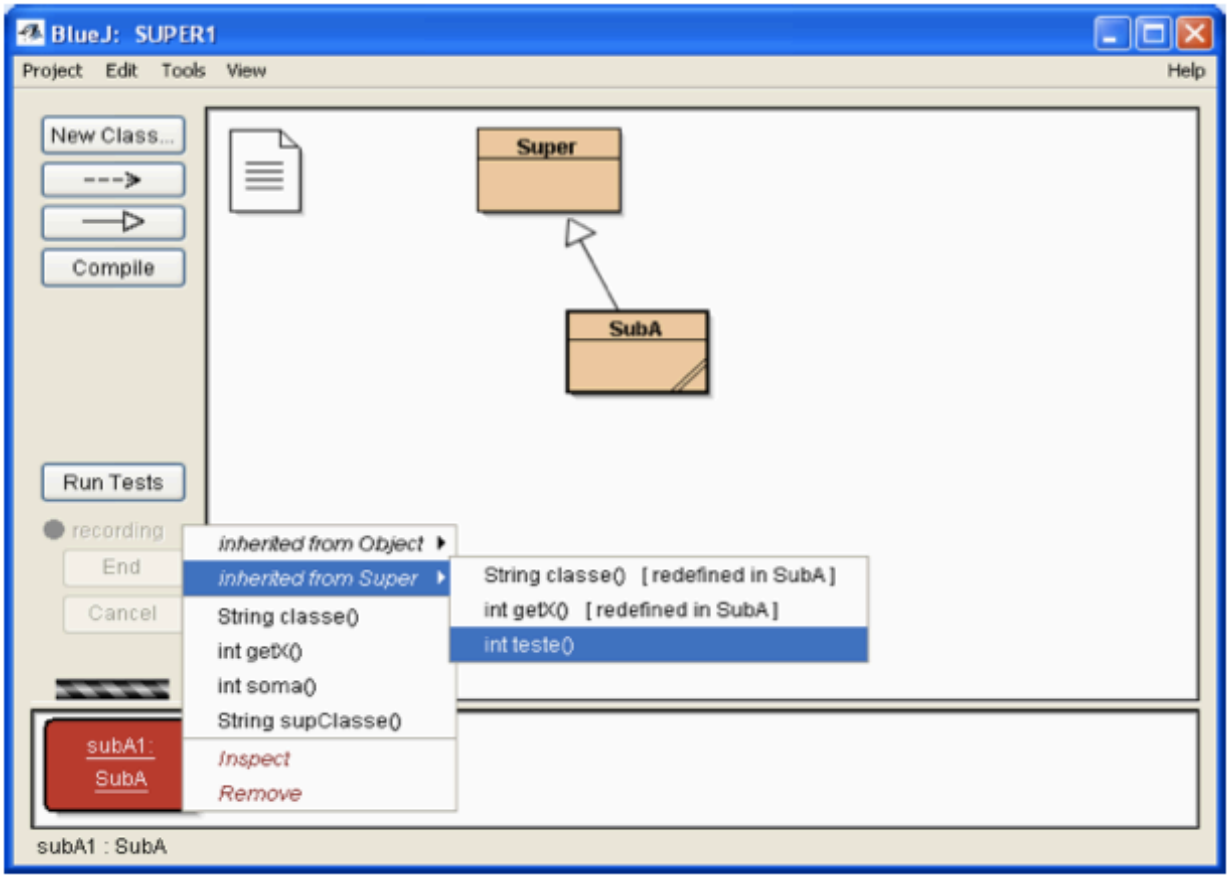
```
public class SubA extends Super {
   private int x = 20; // "shadow"
   private String nome;// "shadow"
   // Métodos
   public int getX() { return x; }
   public String classe() { return "SubA"; }
   public String supClass() { return super.classe(); }
   public int soma() { return x + super.getX();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

- 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



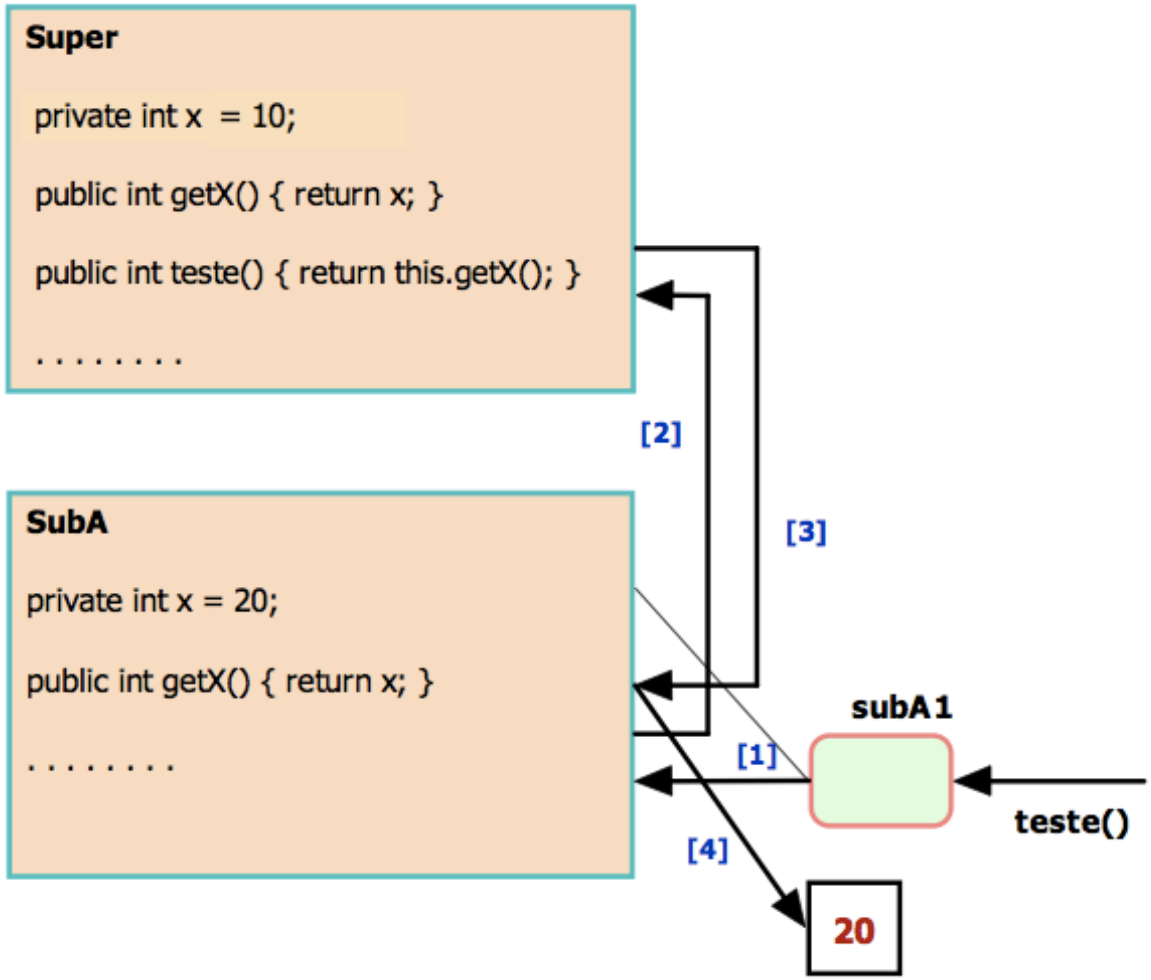
- no BlueJ é possível ver os métodos definidos na classe e os herdados



- o que acontece quando enviamos à instância 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

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, a1;
a = new A(); a1 = new B();
```

- ambas as declarações estão correctas, tendo em atenção a declaração de varíavel 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.

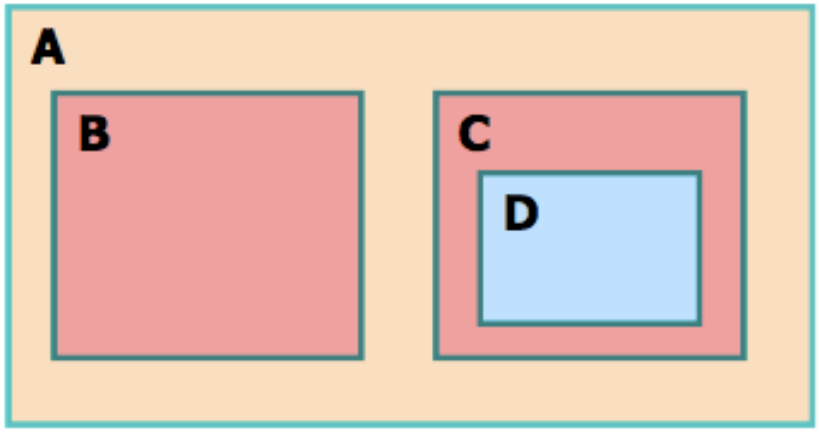
- o interpretador, em tempo de execução, faz o ***dynamic binding*** procurando determinar em função do valor contido qual é o método que deve invocar

- 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...

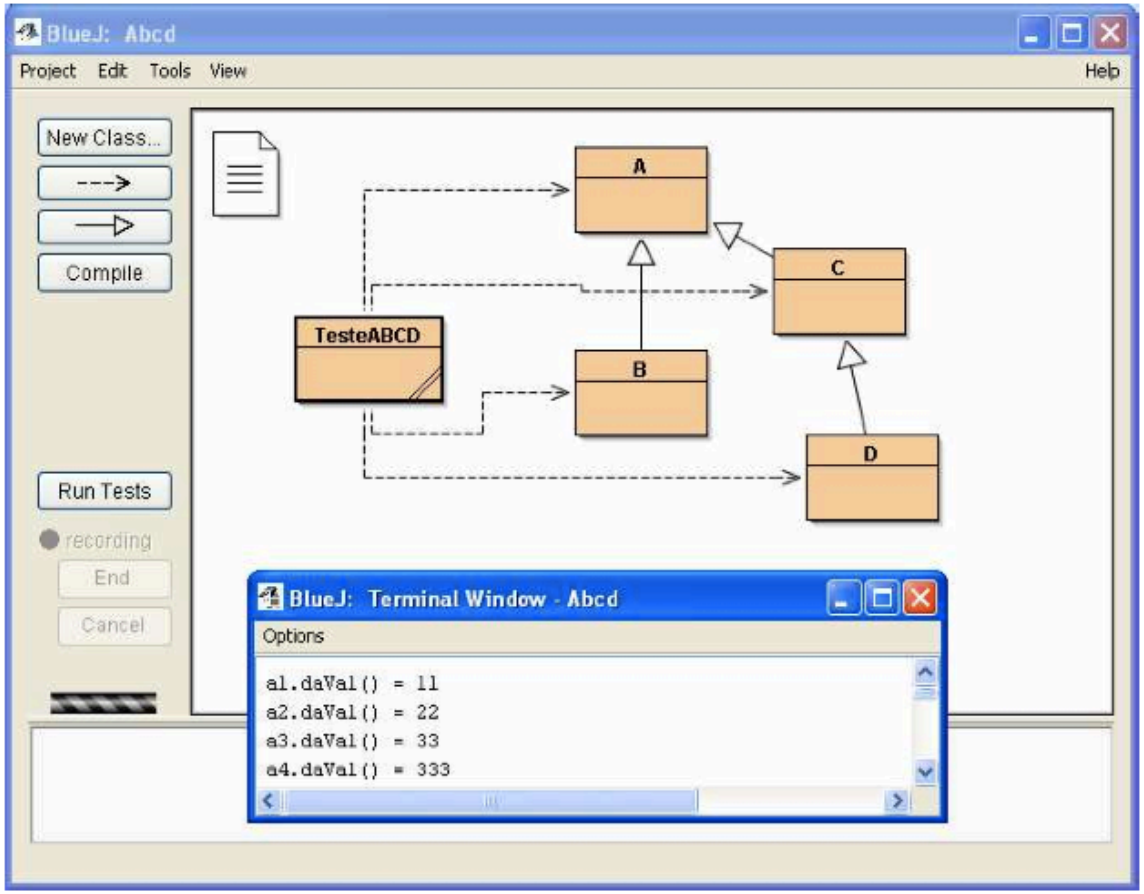
- e as seguintes inicializações de variáveis

```
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 e*stático* 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:



O equals, novamente...

- como vimos anteriormente o método equals de uma subclasse deve invocar o método equals da superclasse, para nesse contexto comparar os valores das v.i. lá declaradas.
  - utilização de **super.equals()**

- seja o método equals da classe Aluno (já conhecido de todos)

```
/**
 * Implementação do método de igualdade entre dois Aluno
 *
 * @param  umAluno   aluno que é comparado com o receptor
 * ** * @return      booleano true ou false
 * ** * */
public boolean equals(Object umAluno) {
    if (this == umAluno)
        return true;

    if ((umAluno == null) || (this.getClass() != umAluno.getClass()))
        return false;
     Aluno a =  (Aluno) umAluno;
     return(this.nome.equals(a.getNome()) && this.nota == a.getNota()
            && this.numero == a.getNumero());
 }
```

- seja agora o método equals da classe AlunoTE, que é subclasse de Aluno:

```
/**
 * Implementação do método de igualdade entre dois Alunos do tipo T-E
 *
 * @param  umAluno   aluno que é comparado com o receptor
 * ** * @return      booleano true ou false
 * ** * */

public boolean equals(Object umAluno) {
    if (this == umAluno)
        return true;

    if ((umAluno == null) || (this.getClass() != umAluno.getClass()))
        return false;
    AlunoTE a =  (AlunoTE) umAluno;
    return(super.equals(a) & this.nomeEmpresa.equals(a.getNomeEmpresa());
}
```

- considerando o que se sabe sobre os tipos de dados, a invocação **this.getClass()** continua a dar os resultados pretendidos?

Polimorfismo

- capacidade de tratar da mesma forma objectos de tipo diferente
  - desde que sejam compatíveis a nível de API
  - ou seja, desde que exista um tipo de dados que os inclua

- no caso das formas geométricas:

```
public double totalArea() {
  double total = 0.0;
  for (Forma f: this.formas)
    total += f.area();
  return total;
}

public int qtsCirculos() {
  int total = 0;
  for (Forma f: this.formas)
    if (f instanceof Circulo) total++;
  return total;
}

public int qtsDeTipo(String tipo) {
  int total = 0;
  for (Forma f: this.formas)
    if ((f.getClass().getSimpleName()).equals(tipo))
      total++;
  return total;
}
```

- no caso do projecto dos Empregados:

```
public double totalSalarios() {
    double total = 0.0;
    for(Empregado emp : emps) total += emp.salario();
    return total;
}

public int totalGestores() {
    int total = 0;
    for(Empregado emp : emps) if(emp instanceof Gestor) total++;
    return total;
}

public int totalDe(String Tipo) {
   int total = 0;
    for(Empregado emp : emps)
      if(emp.getClass().getName().equals(Tipo)) total++;
    return total;
}

public double totalKms() {
    double totalKm = 0.0;
    for(Empregado emp : emps)
      if(emp instanceof Motorista) totalKm += ((Motorista) emp).getKms();
    return totalKm;
}
```

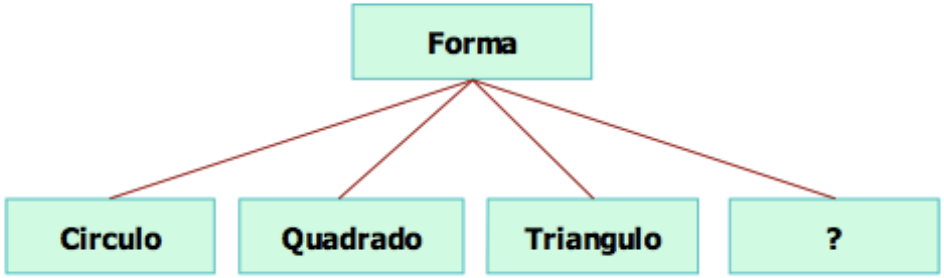
- todos respondem a salario(), embora com implementações diferentes

Classes Abstractas

- até ao momento todas as classes definiram completamente todo o seu estado e comportamento
- no entanto, na concepção de soluções por vezes temos situações em que o código de uma classe pode não estar completamente definido
  - esta é uma situação comum em POO e podemos tirar partido dela para criar soluções mais interessantes

- consideremos que precisamos de manipular forma geométricas (triângulos, quadrados e círculos)
  - no entanto podemos acrescentar, com o evoluir da solução, mais formas geométricas
  - torna-se necessário uniformizar a API que estas classes tem de respeitar
    - todos tem de ter **`area()`** e **`perimetro()`**

- Seja então a seguinte hierarquia:



  - conceptualmente correcta e com capacidade de extensão através da inclusão de novas subclasses de forma

  - mas qual é o estado e comportamento de Forma?

- A classe Forma pode definir algumas v.i., como um ponto central (um Ponto2D), mas se quiser definir os métodos area() e perímetro() como é que pode fazer?
  - Solução 1: não os definir deixando isso para as subclasses
    - as subclasses podem nunca definir estes métodos e aí perde-se a capacidade de dizer que todas as formas respondem a esses métodos

- Solução 2: definir os métodos area() e perimetro() com um resultado inútil, para que sejam herdados e redefinidos
- Solução 3: aceitar que nada pode ser escrito que possa ser aproveitado pelas subclasses e que a única declaração que interessa é a assinatura do método a implementar
    - a maioria das linguagens por objectos aceitam que as definições possam ser incompletas

- em POO designam-se por **classes abstractas** as classes nas quais, pelo menos, um método de instância não se encontra implementado, mas apenas declarado

    - são designados por **métodos abstractos** ou virtuais

    - uma classe 100% abstracta tem apenas assinaturas de métodos

- no caso da classe Forma não faz sentido definir os métodos area() e perímetro, pelo que escrevemos apenas:

```
public abstract class Forma {
//
   public abstract double area();
   public abstract double perimetro();
}
```

- como os métodos não estão definidos, não é possível criar instâncias de classes abstractas

- apesar de ser uma classe abstracta, o mecanismo de herança mantém-se e dessa forma uma classe abstracta é também um (novo) tipo de dados
    - compatível com as instâncias das suas subclasses
    - torna válido que se faça
      `Forma f = new Triangulo()`

- uma classe abstracta ao não implementar determinados métodos, **obriga** a que as suas subclasses os implementem
  - se não o fizerem, ficam como abstractas
- para que servem métodos abstractos?
  - para garantir que as subclasses respondem àquelas mensagens de acordo com a implementação desejada

- Em resumo, as classes abstractas são um mecanismo muito importante em POO, dado que permitem:
  - escrever especificações sintácticas para as quais são possíveis múltiplas implementações
  - fazer com que futuras subclasses decidam como querem implementar esses métodos

- Classe Circulo

```
public class Circulo extends Forma {
   // variáveis de instância
   private double raio;
   // construtores
   public Circulo() {  raio = 1.0; }
   public Circulo(double r) {  raio = (r <= 0.0 ? 1.0 : r); }
   // métodos de instância
   public double area() { return PI*raio*raio; }
   public double perimetro() { return 2*PI*raio; }
   public double raio() {  return raio; }
}
```

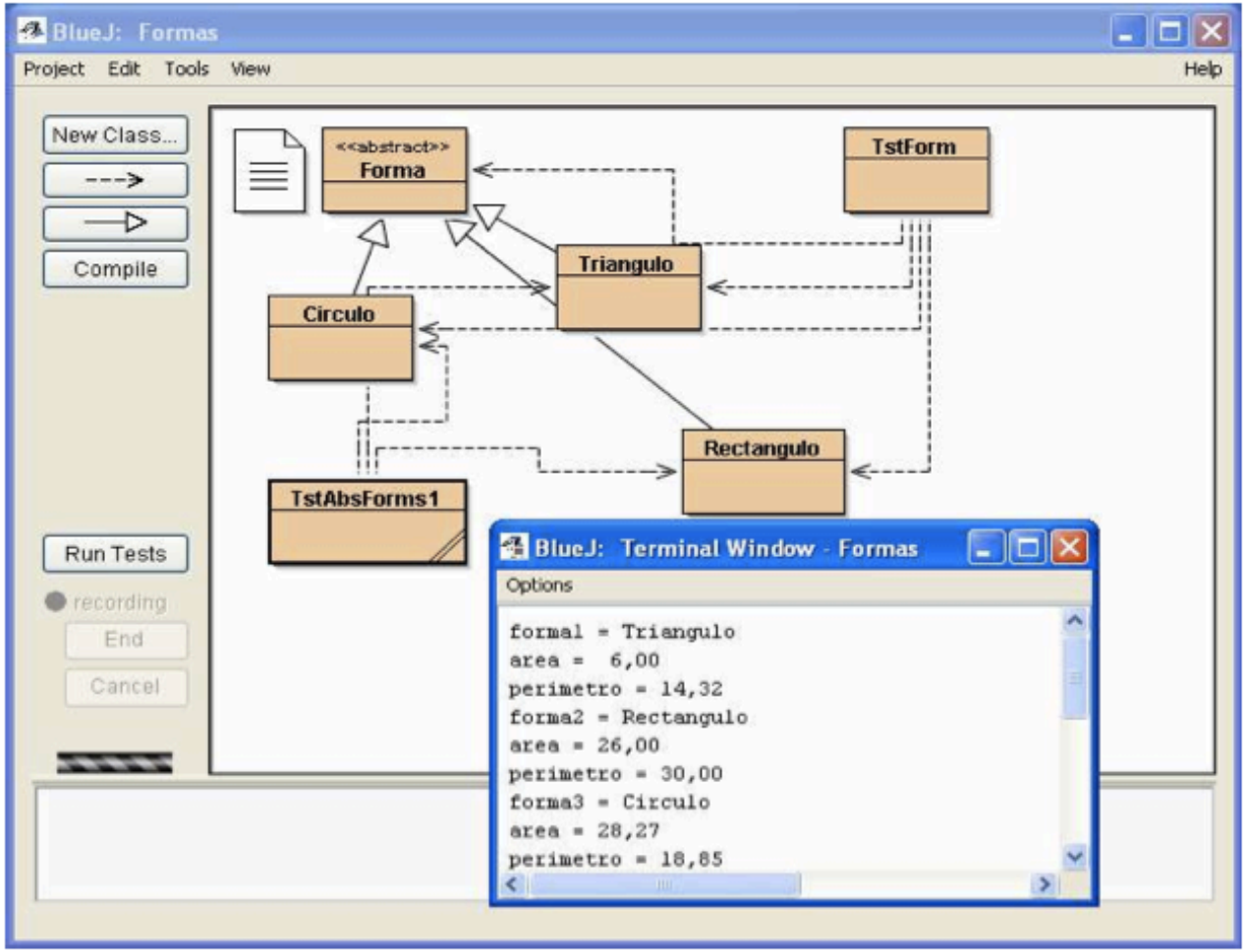
- Classe Rectangulo

```
public class Rectangulo extends Forma {
   // variáveis de instância
   private double comp, larg;
   // construtores
   public Rectangulo() {  comp = 0.0; larg = 0.0; }
   public Rectangulo(double c, double l) {  comp = c; larg = l; }
   // métodos de instância
   public double area() { return comp*larg; }
   public double perimetro() { return 2*(comp+larg); }
   public double largura() {  return larg; }
   public double comp() {  return comp; }
 }
```

- Classe Triangulo:

```
public class Triangulo extends Forma {
   /* altura tirada a meio da base */
   // variáveis de instância
   private double base, altura;
   // construtores
   public Triangulo() { base = 0.0; altura = 0.0; }
   public Triangulo(double b, double a) {
      base = b; altura = a;
   }
   // métodos de instância
   public double area() { return base*altura/2; }
   public double perimetro() {
      return base + (2*this.hipotenusa()); }
   public double base() { return base; }
   public double altura() { return altura; }
   public double hipotenusa() {
      return sqrt(pow(base/2, 2.0) + pow(altura, 2.0)) ;
   }
}
```

- execução do envio dos métodos a diferentes objectos - respostas diferentes consoante o receptor: **polimorfismo**!!



Herança vs Composição

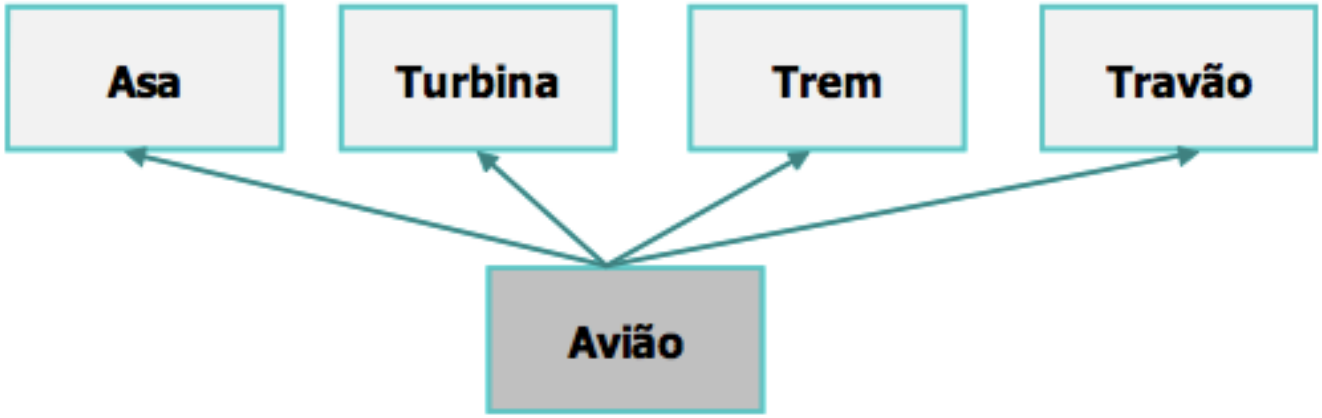
- Herança e composição são duas formas de relacionamento entre classes
  - são no entanto abordagens muito distintas e constitui um erro muito comum achar que podem ser utilizadas da mesma forma
- existe uma tendência para se confundir herança com composição

- quando uma classe é criada por composição de outras, isso implica que as instâncias das classes agregadas fazem parte da definição do contentor
  - é uma relação do tipo "parte de" (part-of)
  - qualquer instância da classe vai ser constituída por instâncias das classes agregadas
    - Exemplo: Círculo tem um ponto central (Ponto2D)

- do ponto de vista do ciclo de vida a relação é fácil de estabelecer:
  - quando a instância contentor desaparece, as instâncias agregadas também desaparecem
  - o seu tempo de vida está iminentemente ligado ao tempo de vida da instância de que fazem parte!

- esta é uma forma (e está aqui a confusão) de criar entidades mais complexas a partir de entidades mais simples:
  - Turma é composta por instâncias de Aluno
  - Automóvel é composto por Pneu, Motor, Chassis, ...
  - Empresa é composta por instâncias de Empregado

- Por vezes em situação de herança múltipla parece tornar-se apelativa uma solução como:



- embora o que se pretende ter é composição. Na solução apresentada o avião apenas *tem (**é**!)* **uma** asa, **uma** turbina, **um** trem de aterragem e **um** travão.

- no caso de termos herança simples (a que temos em Java) a solução de ter um Avião como subclasse de Asa é (ainda mais) claramente errada.
  - é errado dizer que Avião *is-a* Asa
  - é correcto dizer que Asa *part-of* Avião

- quando uma classe (apesar de poder ter instâncias de outras classes no seu estado interno) for uma especialização de outra, então a relação é de **herança**

- quando não ocorrer esta noção de especialização, então a relação deverá ser de **composição**