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 por transitividade, 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 ClasseA

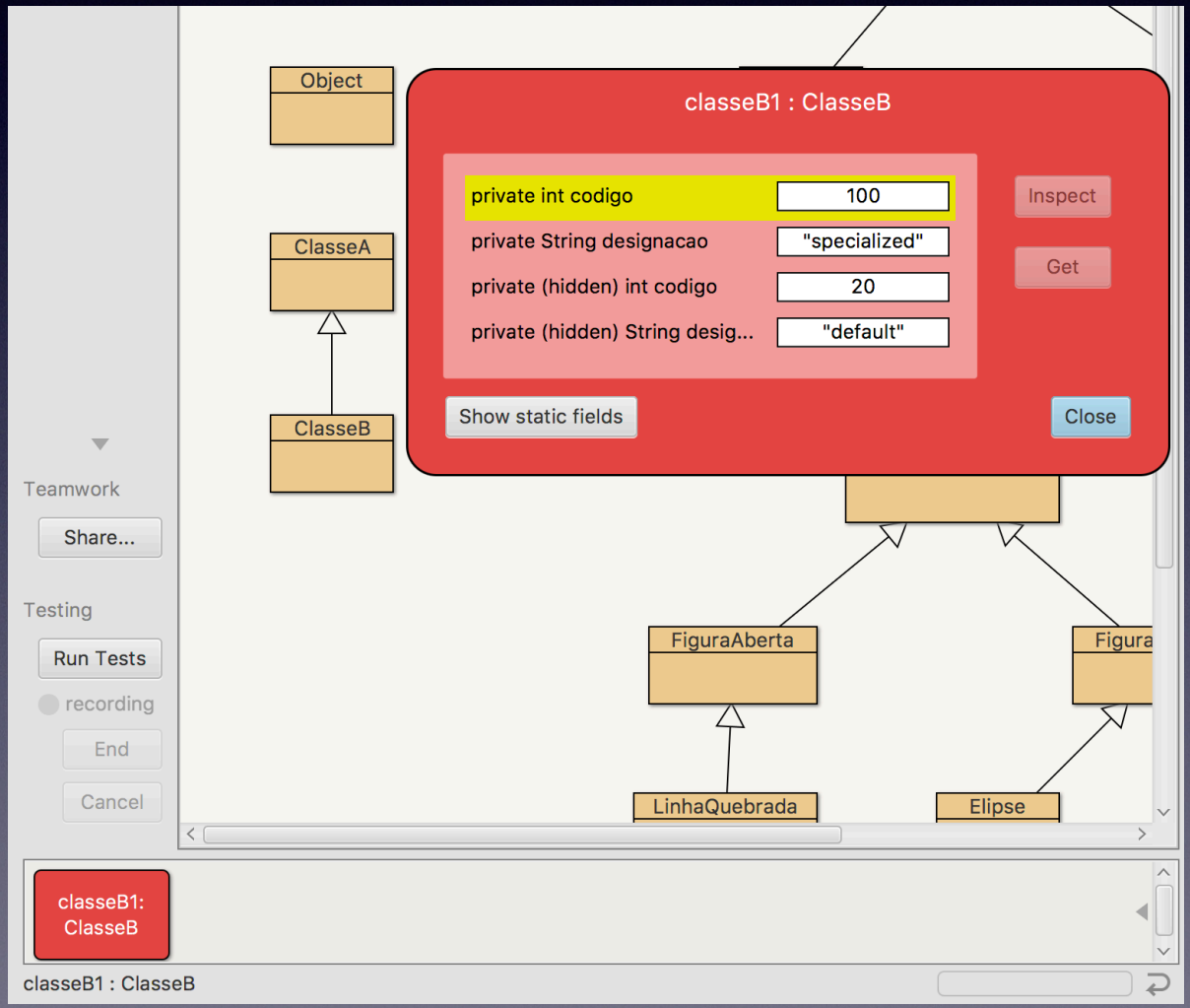
```java
public class ClasseA {
  private int codigo;
  private String designacao;

  public ClasseA() {
    this.codigo = 20;
    this.designacao = "default";
  }
  public int getCodigo() { return this.codigo;}
  public String getDesignacao() {return this.designacao;}
  public int metodo() {return this.getCodigo();}
  public int resultado() {return this.getCodigo();}
}
```

- e uma sua subclasse, ClasseB

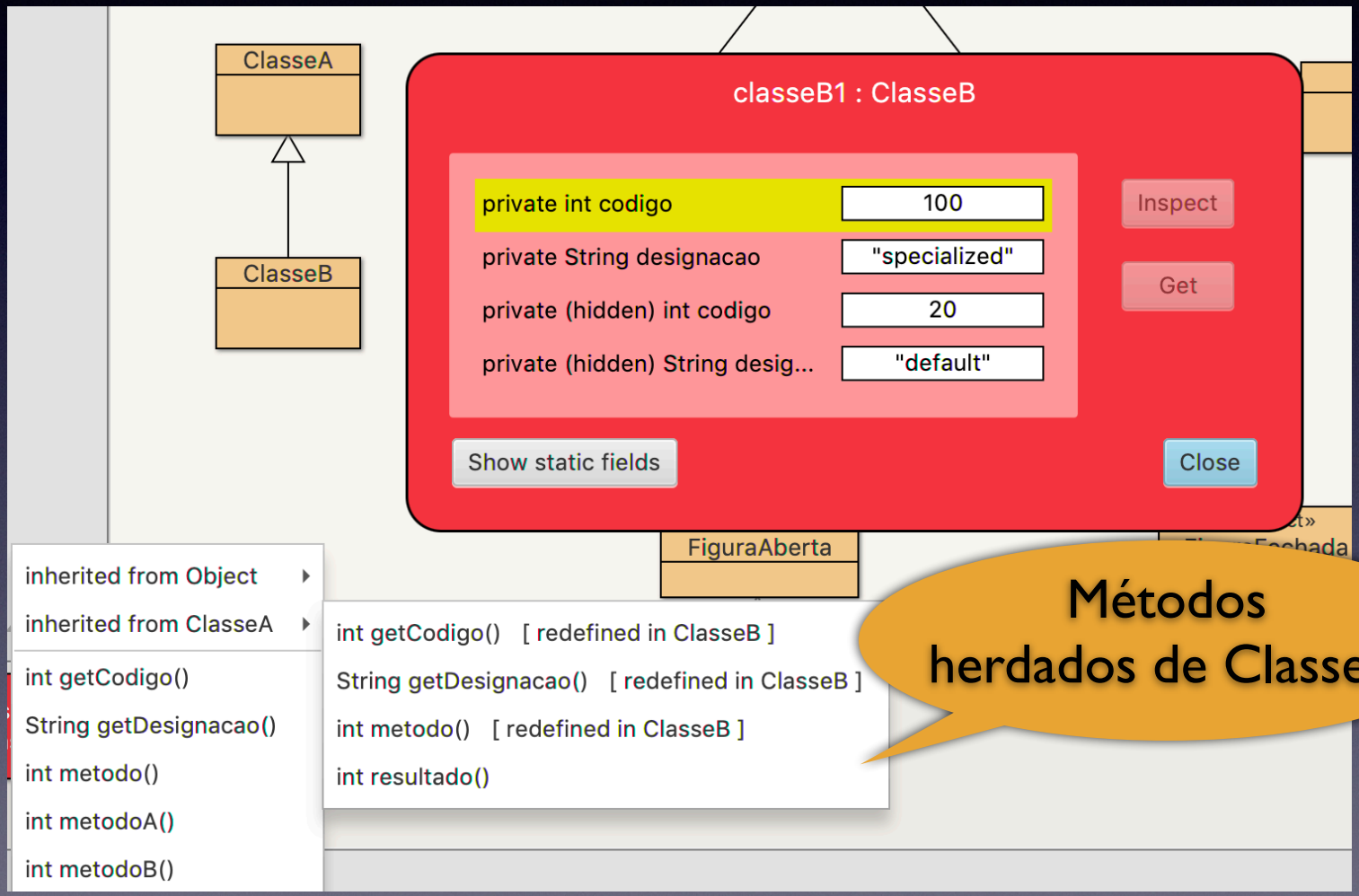
```java
public class ClasseB extends ClasseA {

  private int codigo; // esconde a v.i. de ClasseA
  private String designacao; //esconde a v.i. de ClasseA

  public ClasseB() {
    this.codigo = 100;
    this.designacao = "specialized";
  }
  public int getCodigo() {return this.codigo;}
  public String getDesignacao() {return this.designacao;}
  public int metodo() {return this.getCodigo();}
  public int metodoA() {return super.metodo();}
  public int metodoB() {return metodoA();}
}
```

- 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 BlueJ de um objecto da Classe B



- é também possível visualizar os métodos definidos na classe e os herdados da(s) superclasse(s)



- o que acontece quando enviamos à instância classeB1 (imagem anterior) a mensagem resultado**()**?
  - **resultado()** 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.getCodigo()`

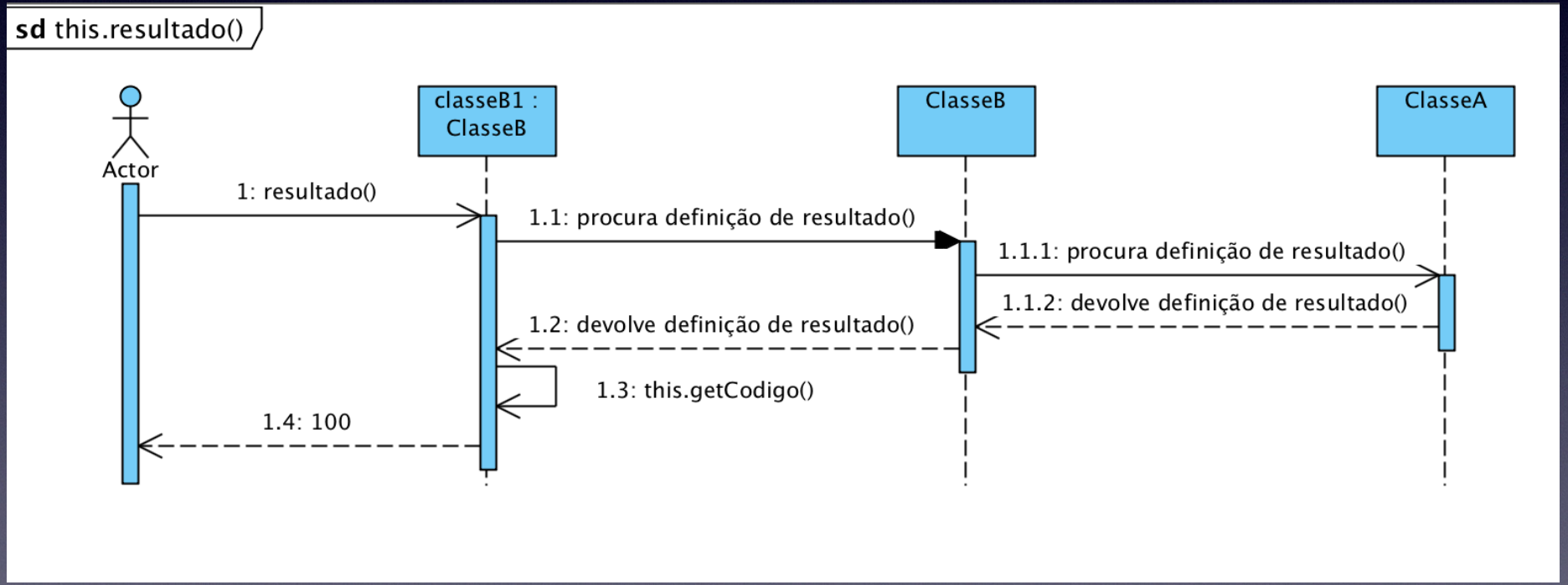
- em ClasseA o valor de codigo é 20, enquanto que em ClasseB o valor é 100.
  - qual é o contexto de execução de **this.getCodigo()**?
  - a que instância é que o **this** se refere?
- Vejamos o algoritmo de procura e execução de métodos...

- qual o resultado de invocar `resultado()` numa instância de `ClasseB`?

```java
public class ClasseA {
  private int codigo;
  private String designacao;

  public ClasseA() {
    this.codigo = 20;
    this.designacao = "default";
  }
  public int getCodigo() { return this.codigo;}
  public String getDesignacao() {return this.designacao;}
  public int metodo() {return this.getCodigo();}
  public int resultado() {return this.getCodigo();}
}
```

```java
public class ClasseB extends ClasseA {

  private int codigo; // esconde a v.i. de ClasseA
  private String designacao; //esconde a v.i. de ClasseA

  public ClasseB() {
    this.codigo = 100;
    this.designacao = "specialized";
  }
  public int getCodigo() {return this.codigo;}
  public String getDesignacao() {return this.designacao;}
  public int metodo() {return this.getCodigo();}
  public int metodoA() {return super.metodo();}
  public int metodoB() {return metodoA();}
}
```

- algoritmo de execução da invocação de resultado() no objecto classeB1:



- na execução do código, a referência a ***this*** corresponde sempre ao objecto que recebeu a mensagem
  - neste caso, a instância classeB1
- sendo assim, o método **getCodigo()** é o método de ClasseB, que é a classe do receptor da mensagem
  - logo, independentemente do contexto "subir e descer", o **`this`** refere sempre o receptor da mensagem!

- E qual o resultado da invocação em classeB1 (instância de ClasseB) dos seguintes métodos?

```
public int metodo() {return this.getCodigo();}
public int metodoA() {return super.metodo();}
public int metodoB() {return metodoA();}
```

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 determinante é saber se uma classe é compatível com as suas subclasses!
- é importante reter o princípio da substituição de Liskov(*) que diz que...

(*) "Family Values: a behavioral notion of subtyping", Barbara Liskov & Jeanette Wing

- “se uma variável é declarada como sendo de uma dada classe (tipo), é admissível 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

```
ClasseA a1, a2;

a1 = new ClasseA();
a2 = new ClasseB();
```

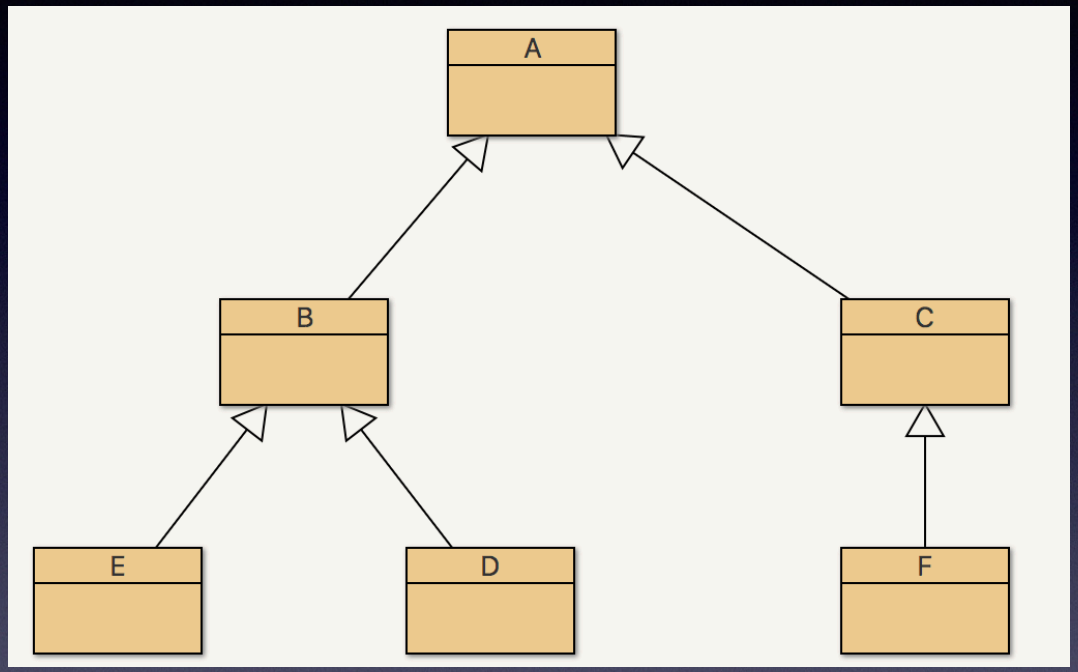
- ambas as declarações estão correctas, tendo em atenção a declaração de variável e a atribuição de valor
  - ClasseB é uma subclasse de ClasseA, pelo que está correcto
- mas o que acontece quando se executa `a2.m()`?

- o compilador tem de verificar se m() existe em ClasseA ou numa sua superclasse (teria sido herdado)
  - se existir é como se estivesse declarado em ClasseB
  - 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. (cf algoritmo procura apresentado)

- 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 novamente considerada a hierarquia:



- … as implementações das várias classes:

```java
public class A {
    private int x;

    public A() {
        this.x = 0;
    }
    public int sampleMethod(int y) {
        return this.x + y;
    }
}
```

```java
public class B extends A {
    private int x;

    public B() {
        this.x = 10;
    }
    public int sampleMethod(int y) {
        return this.x + 2* y;
    }
}
```

```java
public class C extends A {
    private int x;

    public C() {
        this.x = 20;
    }
    public int sampleMethod(int y) {
        return this.x + 2*y;
    }
}
```

```java
public class E extends B {
    private int x;

    public E() {
        this.x = 100;
    }
    public int sampleMethod(int y) {
        return this.x + 10*y;
    }
}
```

```java
public class D extends B {
    private int x;

    public D() {
        this.x = 100;
    }
    public int sampleMethod(int y) {
        return this.x + 20*y;
    }
}
```

```java
public class F extends C {
    private int x;

    public F() {
        this.x = 200;
    }
    public int sampleMethod(int y) {
        return this.x + 3*y;
    }
}
```

- do ponto de vista dos tipos de dados especificados e da relação entre eles, podemos estabelecer as seguintes relações:
  - um B é um A, um C é um A
  - um E é um B, um D é um B
  - um F é um C
  - ou seja, um D pode ser visto como um B ou um A. Um F pode ser visto como um A, etc...

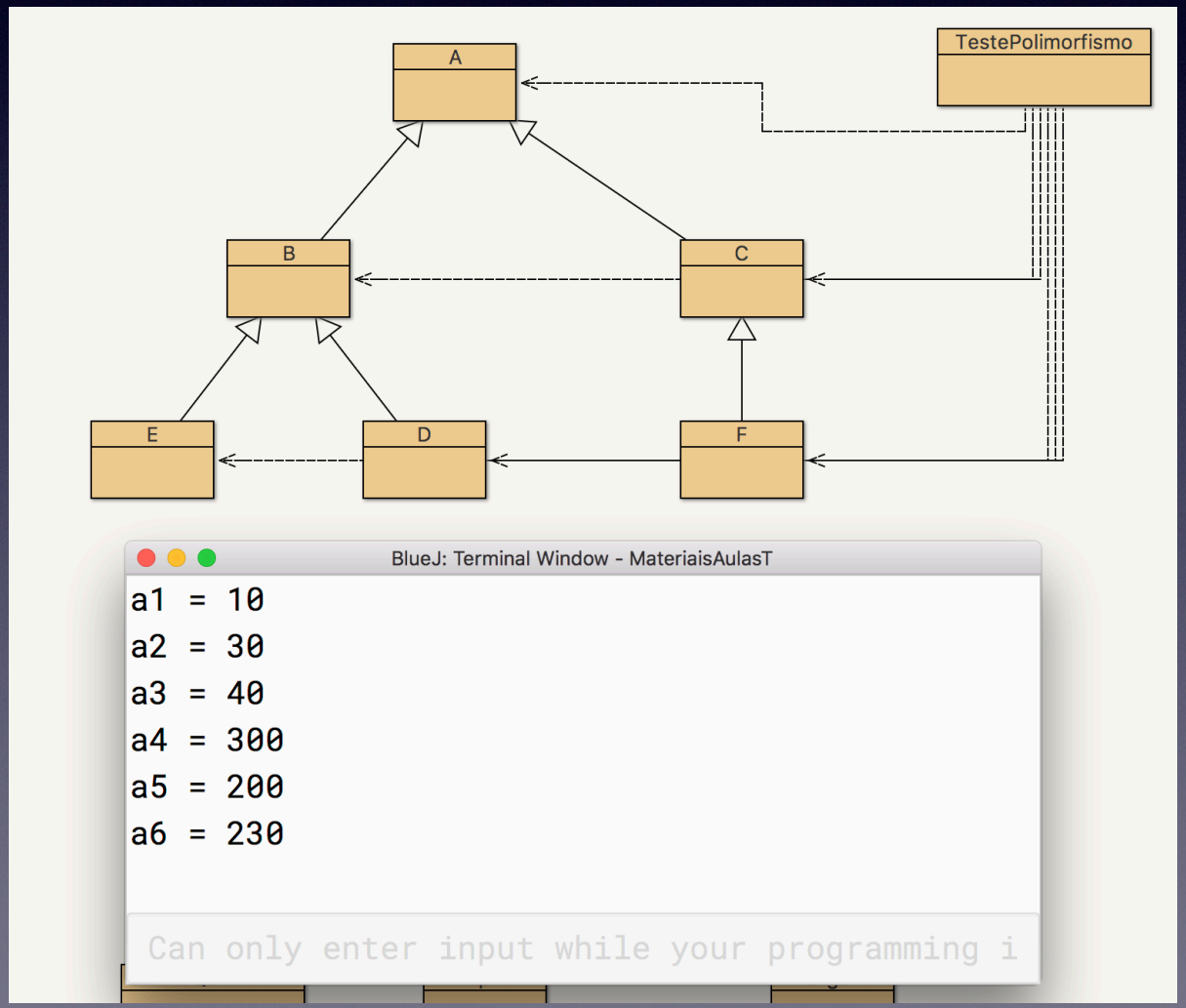
- considere-se o seguinte programa teste:

```
public static void main(String[] args) {

  A a1,a2,a3,a4,a5,a6;
  a1 = new A();
  a2 = new B();
  a3 = new C();
  a4 = new D();
  a5 = new E();
  a6 = new F();

  System.out.println("a1 = " + a1.sampleMethod(10));
  System.out.println("a2 = " + a2.sampleMethod(10));
  System.out.println("a3 = " + a3.sampleMethod(10));
  System.out.println("a4 = " + a4.sampleMethod(10));
  System.out.println("a5 = " + a5.sampleMethod(10));
  System.out.println("a6 = " + a6.sampleMethod(10));
}
```

- qual é o resultado?

- importa 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

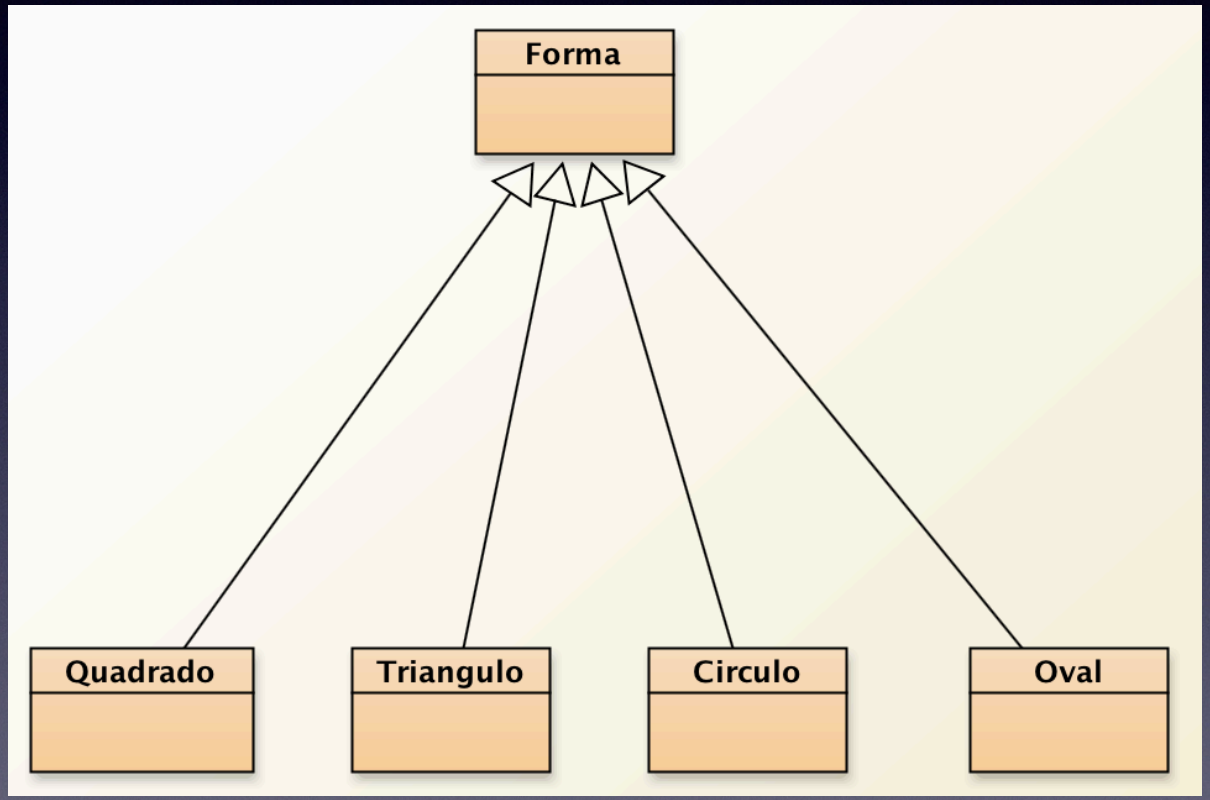
- como o interpretador executa o algoritmo de procura dinâmica de métodos, executando sampleMethod() em cada uma das classes, então o resultado é:



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

Hierarquia das Formas Geométricas



- todas as formas respondem a area() e a perimetro()

- sendo assim é possível tratar de forma igual as diversas instâncias de Forma

```
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;
}
```

- Apesar de termos muitas vantagens em tratar objectos diferentes da mesma forma, por vezes existe a necessidade de saber qual é a natureza de determinado objecto:
  - determinar qual é a classe de um objecto em tempo de execução
  - usando `instanceof` ou `getClass().getSimpleName()`