Programação Orientada aos Objectos

LEI/LCC - 2º ano 2013/14

António Nestor Ribeiro

As origens do Paradigma dos Objectos

- a maioria dos conceitos fundamentais da POO aparece nos anos 60 ligado a ambientes e linguagens de simulação
- a primeira linguagem a utilizar os conceitos da POO foi o SIMULA-67
  - era uma linguagem de modelação
  - permitia registar modelos do mundo real

- o objectivo era representar entidades do mundo real:
  - identidade (única)
  - estrutura (atributos)
  - comportamento (acções e reacções)
  - interacção (com outras entidades)

- Simula-67 introduz o conceito de "classe" como a entidade definidora e geradora de todos os "indivíduos" que obedecem a um dado padrão de:
  - estrutura
  - comportamento
- Classes são fábricas de indivíduos
  - a que mais para a frente chamaremos de "objectos"!

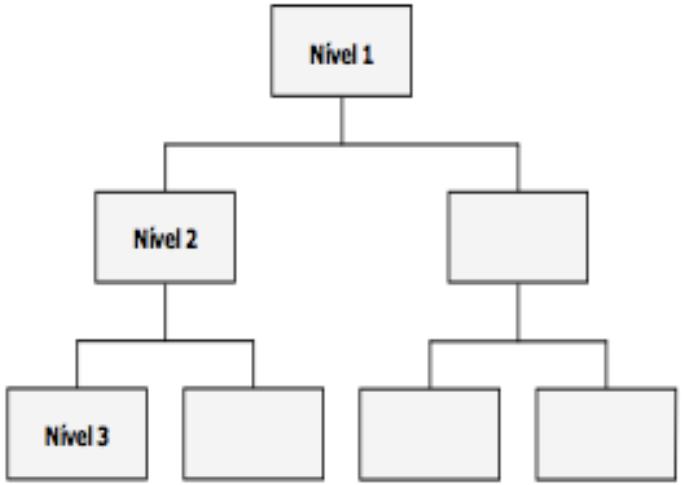
POO na Engenharia de Software

- nos anos 60 e 70 a Engenharia de Software havia adoptado uma base de trabalho que permitia ter um processo de desenvolvimento e construção de linguagens
- esses princípios de análise e programação designavam-se por estruturados e procedimentais

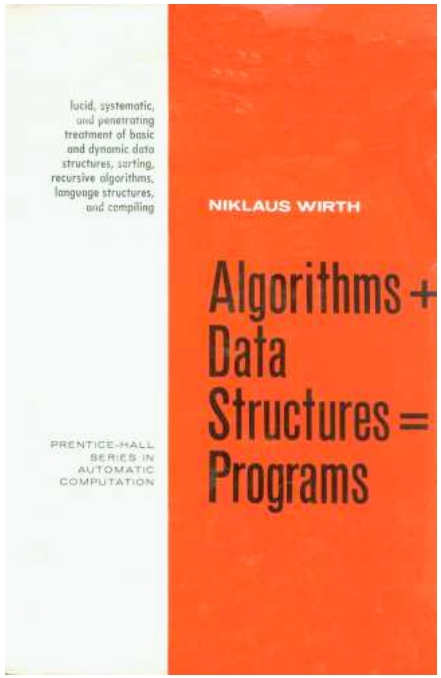
- a abordagem preconizada era do tipo "top-down"
  - estratégia para lidar com a complexidade
  - a princípio tudo é pouco definido e por refinamento vai-se encontrando mais detalhe
- neste modelo estruturado funcional e top-down:
  - as acções representam as entidades computacionais de 1ª classe
  - os dados são entidades de 2ª classe

Estratégia Top-Down

- refinamento progressivo dos processos



- Niklaus Wirth escreve nos anos 70 o corolário desta abordagem no livro "Algoritmos + Estruturas de Dados = Programas"



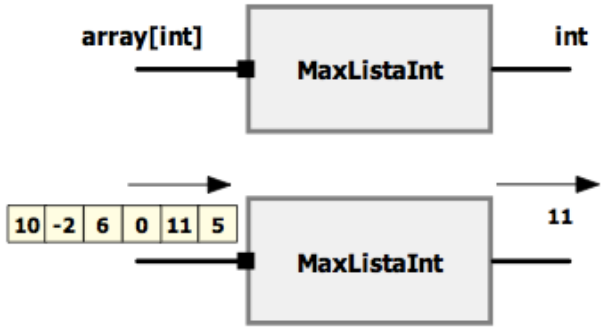
- esta abordagem não apresentava grandes riscos em projectos de pequena dimensão
- contudo em projectos de dimensão superior começou a não ser possível ignorar as vantagens da reutilização que não eram evidentes na abordagem estruturada
- É importante reter a noção de **reutilizaçao** de software, como mecanismo de aproveitamento de código já desenvolvido e aplicado noutros projectos.

- Mas, como é que isto se faz numa programação estruturada?...
  - documentação, guia de estilo de programação, etc.
  - através da utilização dos mecanismos das linguagens:
    - procedimentos
    - funções
    - rotinas

Abstracção de controlo

- utilização de procedimentos e funções como mecanismos de incremento de reutilização
- não é necessário conhecer os detalhes do componente para que este seja utilizado
- procedimentos são vistos como caixas negras (black boxes), cujo interior é desconhecido, mas cujas entradas e saídas são conhecidas

- por exemplo: ter uma função que dado um array de inteiros devolve o maior deles

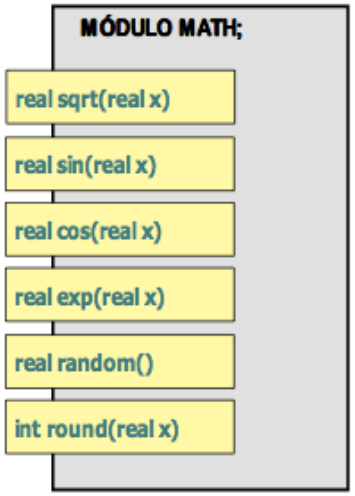


- estes mecanismos suportam uma reutilização do tipo "copy&paste"
- a reutilização está muito dependente dos tipos de dados de entrada e saída

Módulos

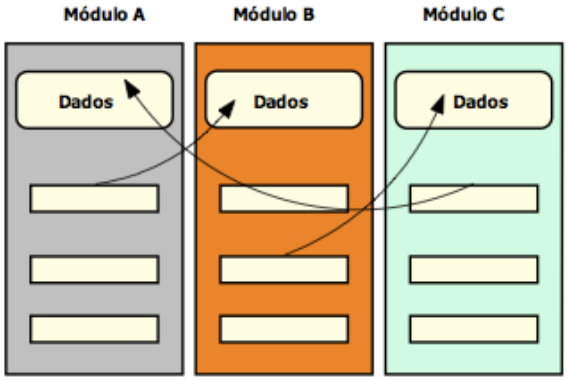
- como forma de aumentar o grão da reutilização várias linguagens criaram a noção de **módulos**
- os módulos possuem declarações de dados e declarações de funções e procedimentos invocáveis do exterior
- possuem a (grande) vantagem de poderem ser compilados de forma autónoma
  - podem assim ser associados a diferentes programas

- módulo como abstracção procedimental:



- no entanto, este modelo não garante a estanquicidade dos dados
- os procedimentos de um módulo podem aceder aos dados de outros módulos

- módulos interdependentes:

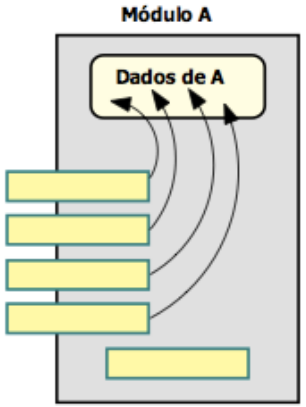


- a partilha de dados quebra as vantagens de uma possível reutilização
- num cenário mais real os diversos módulos interdependentes teriam de ser todos compilados e importados para os programas cliente

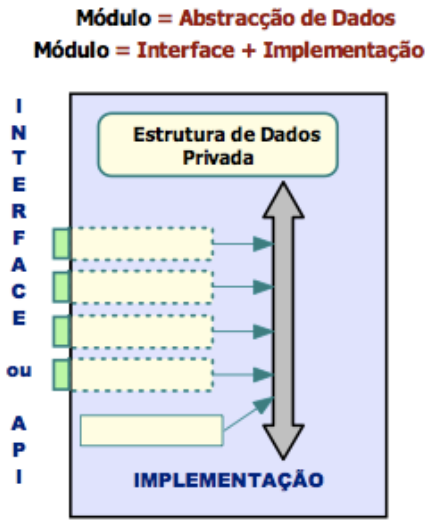
Tipos Abstractos de Dados

- os módulos para serem totalmente autónomos devem garantir que:
  - os procedimentos apenas acedem às variáveis locais ao módulo
  - não existem instruções de input/output no código dos procedimentos

- A estrutura de dados local passa a estar completamente escondida: **Data Hiding**
- Os procedimentos e funções são serviços (API) que possibilitam que do exterior se possa obter informação acerca dos dados
- Módulos passam assim a ser vistos como mecanismos de **abstracção de dados**



- se os módulos forem construídos com estas preocupações, então passamos a ter:
  - capacidade de reutilização
  - encapsulamento de dados



Evolução das abordagens

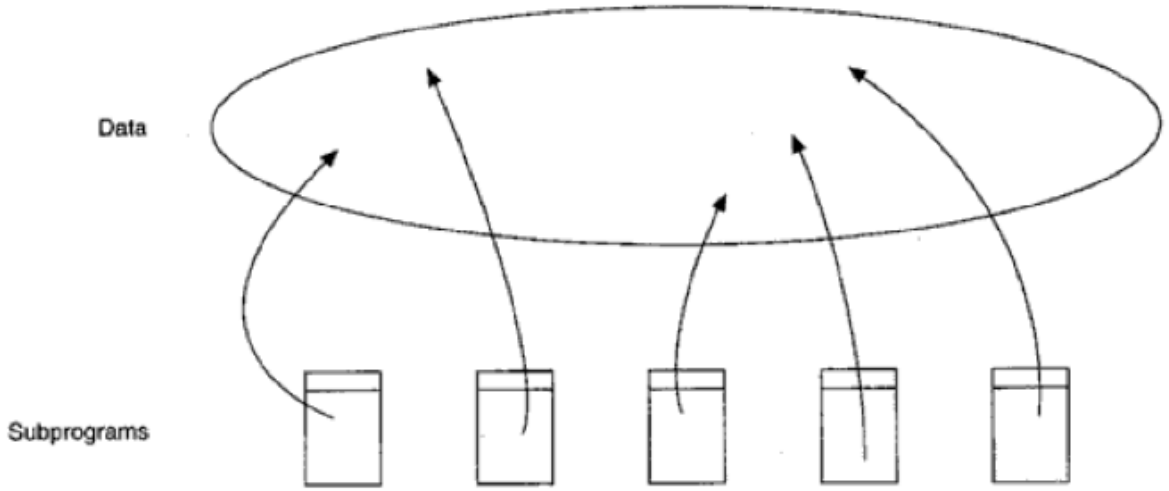


Figure 2-1
The Topology of First- and Early Second-Generation Programming Languages

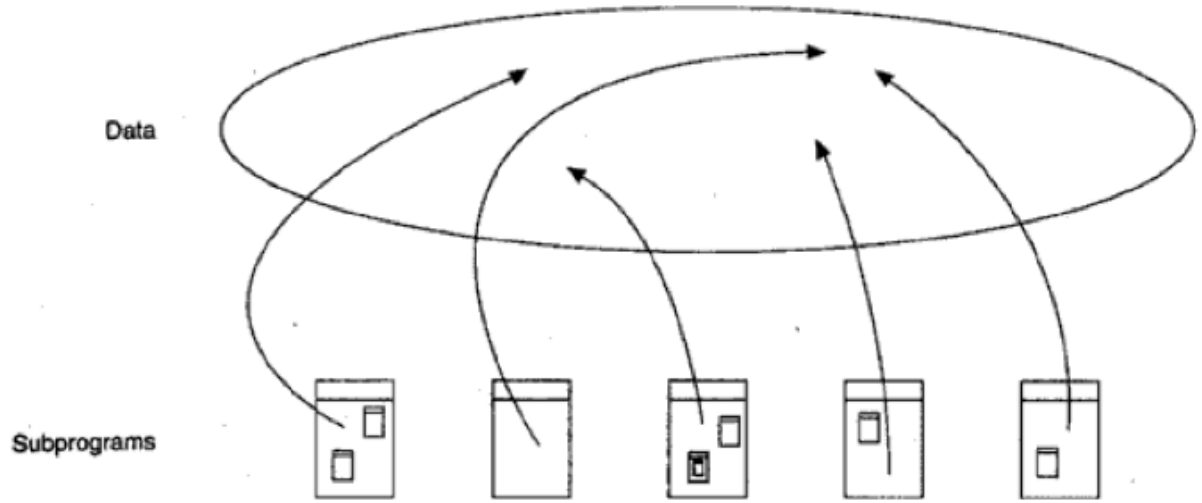


Figure 2-2
The Topology of Late Second- and Early Third-Generation Programming Languages

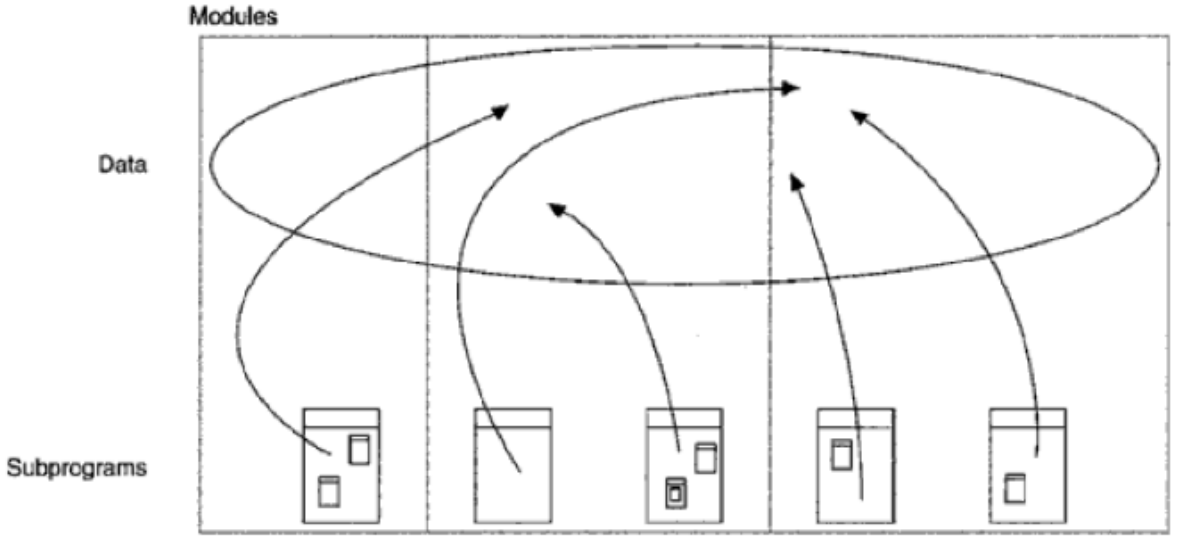


Figure 2-3
The Topology of Late Third-Generation Programming Languages

Onde queremos chegar

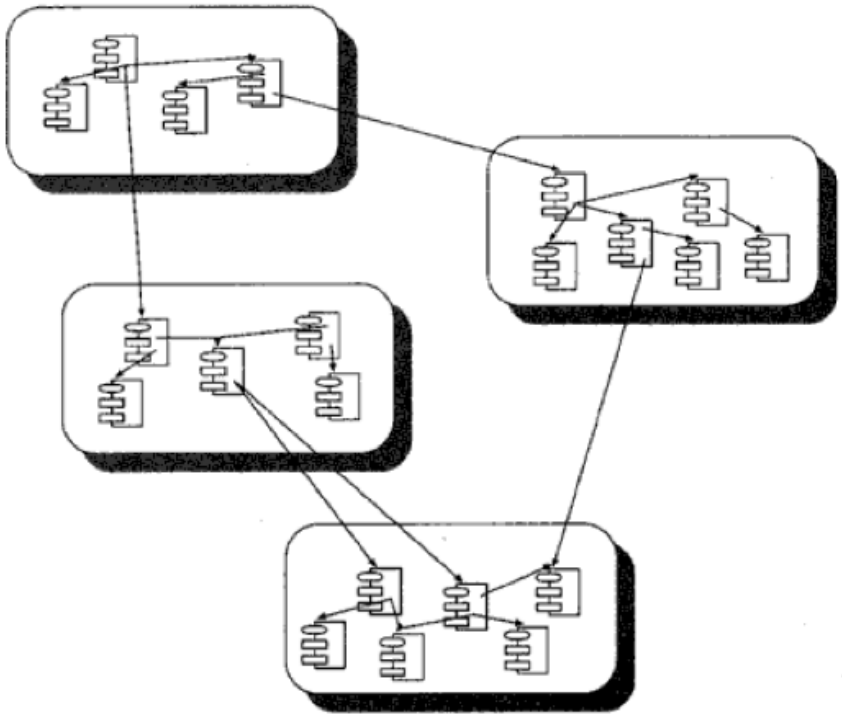
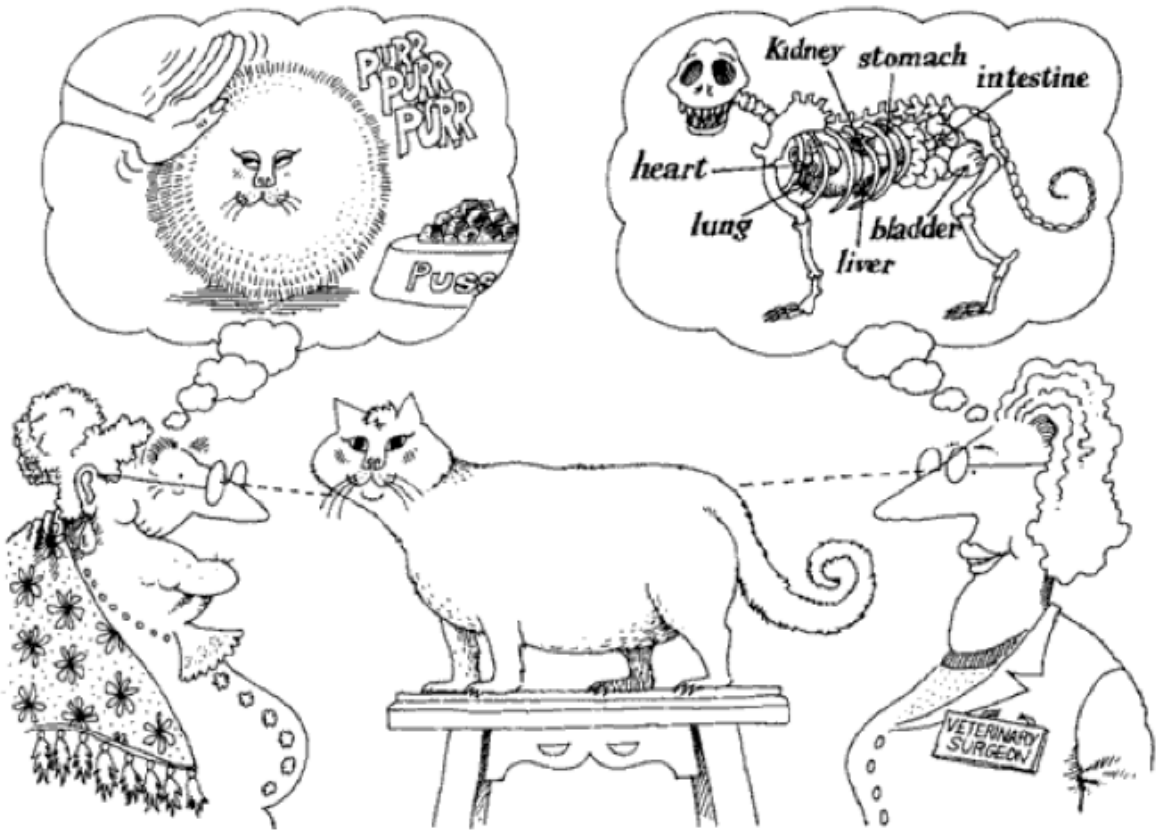


Figure 2-5
The Topology of Large Applications Using Object-Based and Object-Oriented Programming Languages

Abstracção


PURR PURR PURR
Puss
Kidney
stomach
intestine
heart
lung
bladder
liver
VETERINARY SURGEON

Exemplo de um TAD

```
MODULE COMPLEXO;

TYPE
  COMPLEXO = RECORD
               real: REAL; // parte real
               img : REAL; // parte imaginária
             END;

(* --- Procedimentos e Funções ---*)

PROCEDURE criaCmplx(r: REAL; i: REAL) : COMPLEXO

PROCEDURE getReal(c: COMPLEXO): REAL;

PROCEDURE getImag(c: COMPLEXO) : REAL;

PROCEDURE mudaReal(dr: REAL; c: COMPLEXO) : COMPLEXO;

PROCEDURE iguais(c1: COMPLEXO; c2: COMPLEXO) : BOOLEAN;

PROCEDURE somaComplx(c: COMPLEXO; c1: COMPLEXO) : COMPLEXO;

END MODULE.
```

- Vejamos como é que este módulo pode ser utilizado pelos diversos programas...

- Exemplo de um programa que respeita os princípios de utilização de módulos

```
IMPORT COMPLEXO;       // PROGRAMA A

VAR complx1, complx2 : COMPLEXO;
    preal, pimg : REAL;

BEGIN
  complx1 = criaComplx(2.5, 3.6);

  preal = getReal(complx1); writeln("Real1 = ", preal);
  pimg  = getImag(complx1); writeln("Imag1 = ", pimg);

  complx2 = criaComplx(5.1, -3.4);

  complx2 = mudaReal(5.99, complx2);
  preal = getReal(complx2); writeln("Real2 = ", preal);

  complx2 = somaComplx(complx1, complx2);

  preal = getReal(complx2); writeln("Real2 = ", preal);
  pimg  = getImag(complx2); writeln("Imag2 = ", pimg);

END.
```

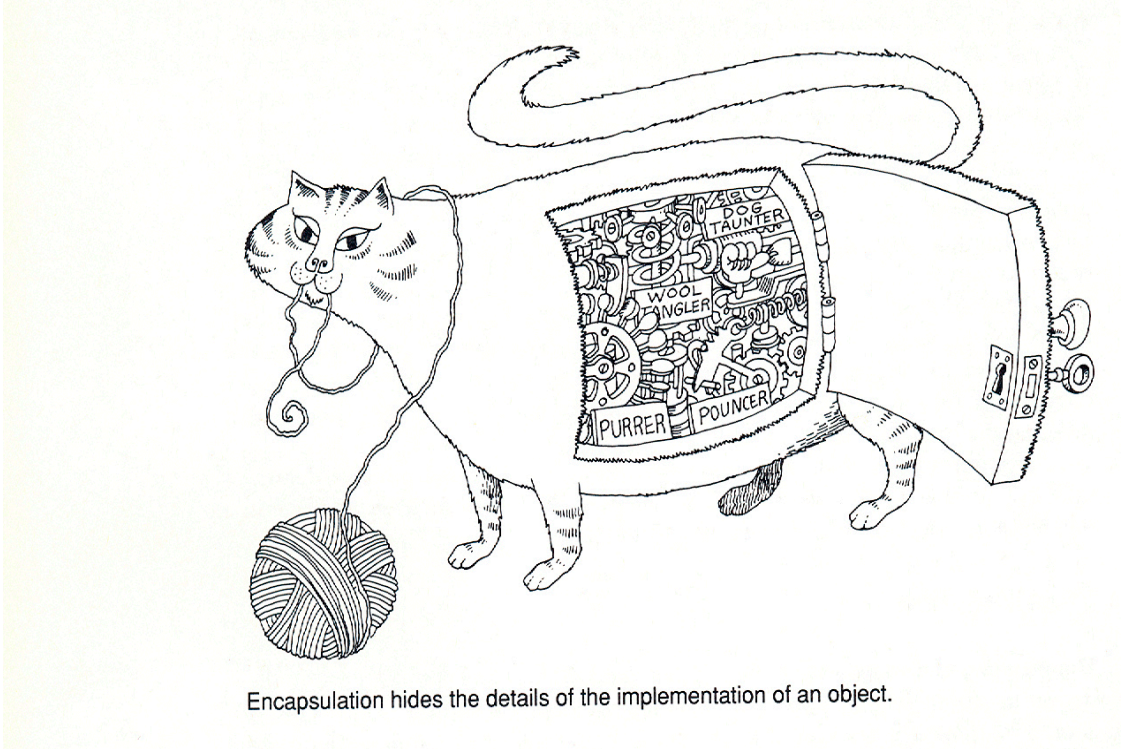
- todo o código está feito utilizando apenas a API (interface) do módulo Complexo

- é possível utilizar o mesmo módulo, mas de forma menos correcta e não respeitando o encapsulamento dos dados

```
IMPORT COMPLEXO;       // PROGRAMA B

VAR complx1, complx2 : COMPLEXO;
    preal, pimg : REAL;

BEGIN
  complx1 = criaComplx(2.5, 3.6);

  preal = complx1.real; writeln("Real1 = ", preal);
  pimg  = complx1.img; writeln("Imag1 = ", pimg);

  complx2 = criaComplx(5.1, -3.4);

  complx2.real = 5.99;
  preal = complx2.real; writeln("Real2 = ", preal);

  complx2.real = complx1.real + complx2.real;
  complx2.img = complx1.img + complx2.img;

  preal = getReal(complx2); writeln("Real2 = ", preal);
  pimg  = getImag(complx2); writeln("Imag2 = ", pimg);

END.
```

Encapsulamento

- apenas se conhece a interface e os detalhes de implementação estão escondidos



Encapsulation hides the details of the implementation of an object.

Desenvolvimento em larga escala

- desta forma estamos a favorecer as metodologias de desenvolvimento para sistemas de larga escala
- Factores decisivos:
  - data hiding
  - implementation hiding
  - abstracção de dados
  - encapsulamento
  - independência contextual

Metodologia

- criar o módulo pensando no tipo de dados que se vai representar e manipular
- definir as operações de acesso e manipulação dos dados internos
- criar operações de acesso exterior aos dados
- não ter código de I/O nas diversas operações
- na utilização dos módulos utilizar apenas a API

Passagem para POO

- Um objecto é a representação de uma entidade do mundo real, com:
  - atributos privados
  - operações

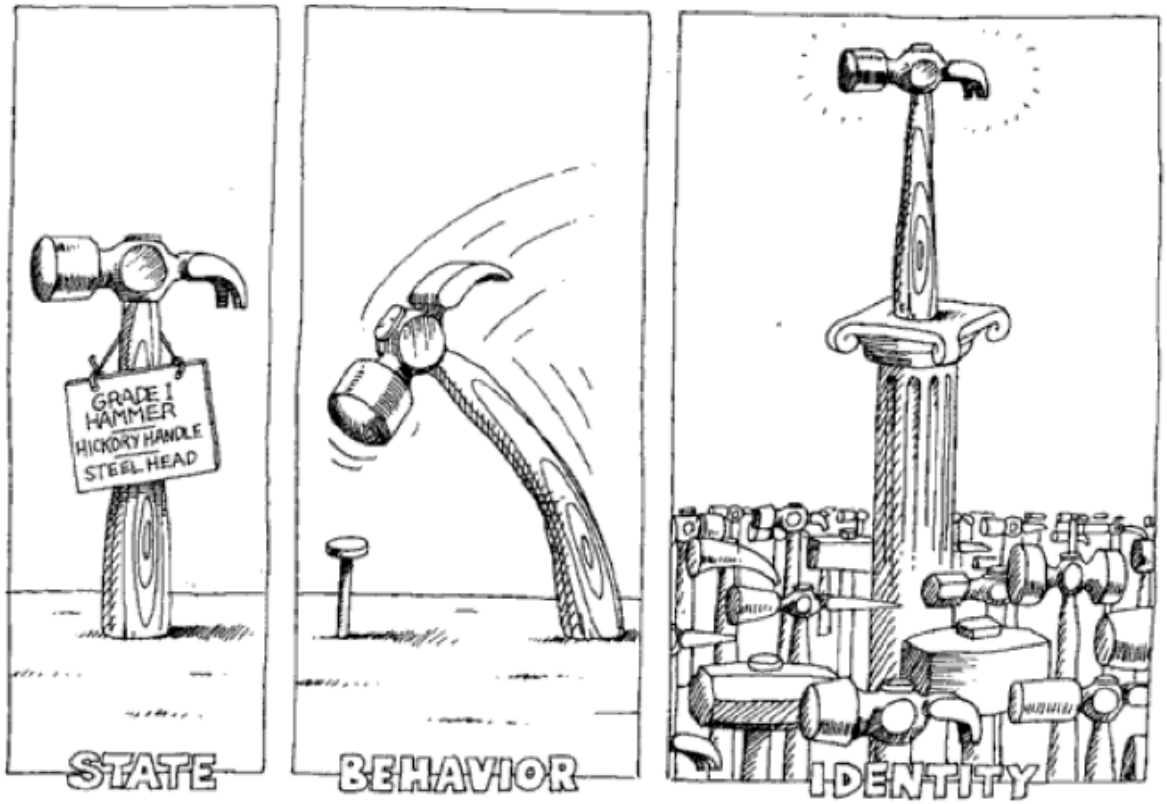
- **Objecto** = Dados Privados (variáveis de instância) + Operações (métodos)

Definição de Objecto

- a noção de objecto é uma das definições essenciais do paradigma
- assenta nos seguintes princípios:
  - independência do contexto (reutilização)
  - abstracção de dados (abstração)
  - encapsulamento (abstração e privacidade)
  - modularidade (composição)

- um objecto é o módulo computacional básico e único e tem como características:
  - **identidade** única
  - um conjunto de atributos privados (o **estado** interno)
  - um conjunto de operações que acedem ao estado interno e que constituem o **comportamento**. Algumas das operações são públicas e visíveis do exterior (a API)

Estado, Comportamento e Identidade

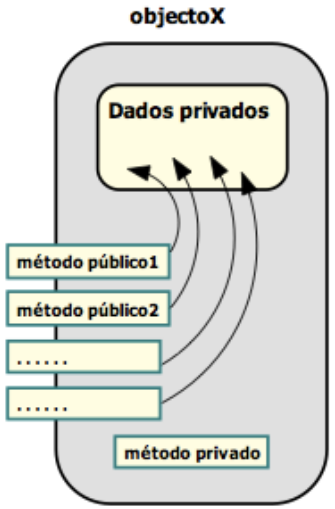


- Objecto = “black box”
  - apenas se conhecem os pontos de acesso (as operações)
  - desconhece-se a implementação interna

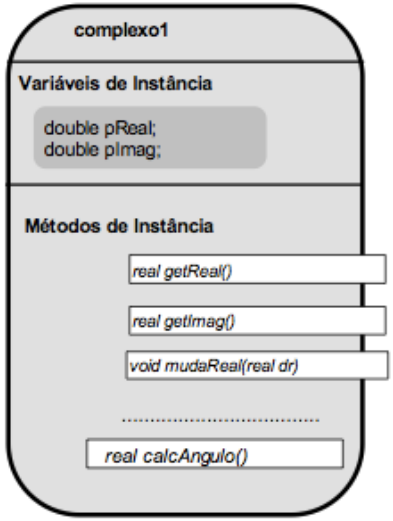
- Vamos chamar
  - aos dados: **variáveis de instância**
  - às operações: **métodos de instância**

Encapsulamento

- Um objecto deve ser visto como uma "cápsula", assegurando a protecção dos dados internos



- Um objecto que representa um número complexo:

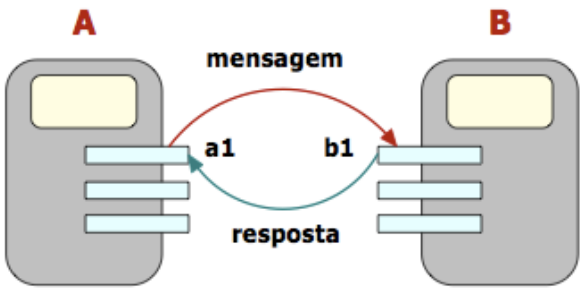


- O método `calcAngulo()` é interno e não pode ser invocado por entidades externas

- Um objecto é:
  - uma unidade computacional fechada e autónoma
  - capaz de realizar operações sobre os seus atributos internos
  - capaz de devolver respostas para o exterior, sempre que estas lhe sejam solicitadas
  - capaz de garantir uma gestão autónoma do seu espaço de dados interno

Mensagens

- a interacção entre objectos faz-se através do envio de mensagens



Novo alfabeto

- o facto de termos agora um alfabeto de mensagens a que cada objecto responde, altera as frases válidas em POO
  - **objecto.m()**
  - **objecto.m(arg1,...,argn)**
  - **r = objecto.m()**
  - **r = objecto.m(arg1,...,argn)**

- propositadamente, estão fora deste alfabeto as frases:
  - **r = o.var**
  - **o.var = x**
  - em que se acede, de forma directa e não protegida, ao campo **var** da estrutura interna do objecto **o**

- Se **ag** for um objecto que represente uma agenda, então poderemos fazer:
  - ag.insereEvento("TestePOO",11,6,2014), para inserir um novo evento na agenda
  - ag.libertaEventosDia(25,4,2014), para remover todos os eventos de um dia
  - String[] ev = ag.getEventos(6,2014), para obter a descrição de todos os eventos do mês de Junho

...noção de Objecto

- *"An object has state, behavior, and identity; the structure and behavior of similar objects are defined in their common class; the terms instance and object are interchangeable."*, Grady Booch, 1993

Definição de Classe

- numa linguagem por objectos, tudo são objectos, logo uma classe é um objecto "especial"
- uma classe é um objecto que serve de padrão (molde, forma) para a criação de objectos similares (uma vez que possuem a mesma estrutura e comportamento)
  - aos objectos criados a partir de uma classe chamam-se *instâncias*

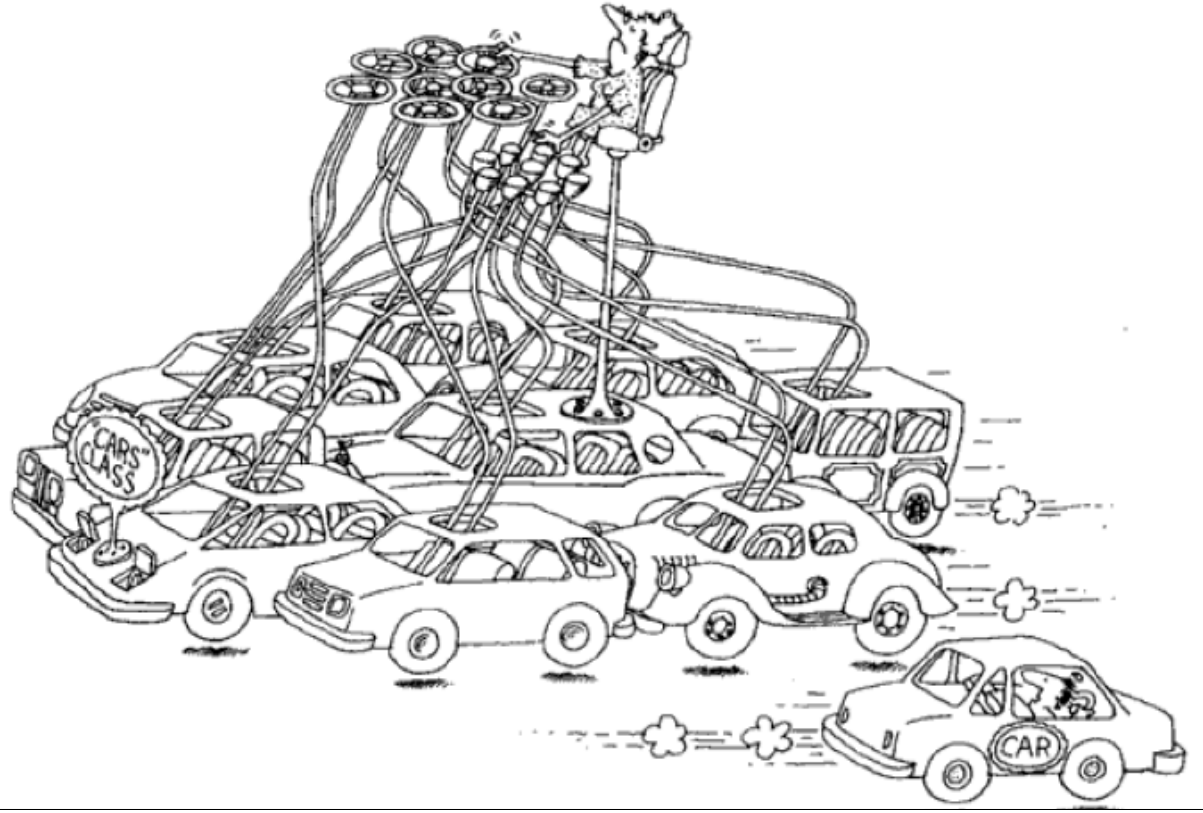
- uma classe é um *módulo* onde se especifica quer a estrutura quer o comportamento das instâncias que a partir dela criamos
- uma vez que todos os objectos criados a partir de uma classe respondem à mesma interface, i.e. o mesmo conjunto de mensagens, são exteriormente utilizáveis de igual forma
  - a classe pode ser vista como um *tipo de dados*

- *“The concepts of a class and an object are tightly interwoven, for we cannot talk about an object without regard for its class. However, there are important differences between these two terms. Whereas an object is a concrete entity that exists in time and space, a class represents only an abstraction, the “essence" of an object, as it were.*

  Thus, we may speak of the class ***Mammal****, which represents the characteristics common to all mammals. To identify a particular mammal in this class, we must speak of "this mammal” or "that mammal.",* Grady Booch, 1993

Uma classe e as suas instâncias

- *A class is a set of objects that share a common structure and a common behavior*, Grady Booch, 1993



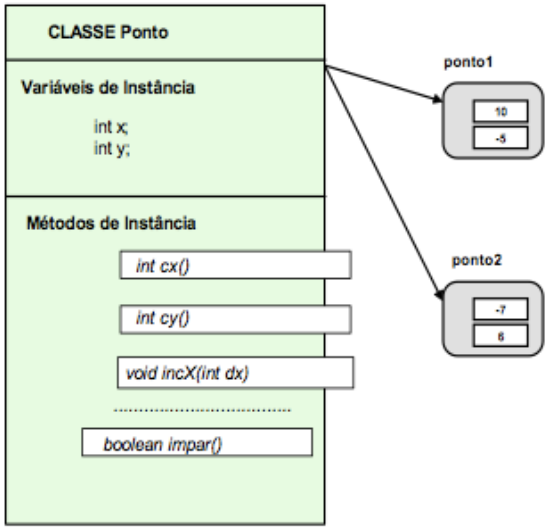
...e ainda sobre classes

- *"What isn't a class? An object is not a class, although, curiously, [...], a class may be an object. Objects that share no common structure and behavior cannot be grouped in a class because, by definition, they are unrelated except by their general nature as objects."*, Grady Booch, 1993

Classe Ponto 2D

- um ponto no espaço 2D inteiro (X,Y) possui como estado interno as duas coordenadas
- um conjunto de métodos que permitem que os objectos tenham comportamento
  - métodos de acesso às coordenadas
  - métodos de alteração das coordenadas
  - outros métodos

- a classe Ponto2D e duas instâncias:

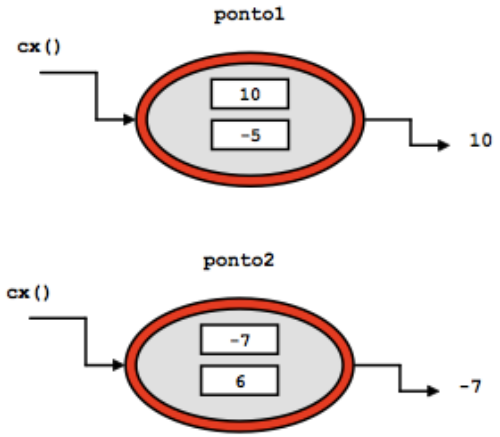


- *ponto1* e *ponto2* são instâncias diferentes, mas possuem o mesmo comportamento
- não faz sentido replicar o comportamento por todos os objectos

Modelo de execução dos métodos

- quando uma instância de uma classe recebe uma dada mensagem, solicita à sua classe a execução do método correspondente
  - os valores a utilizar na execução são os do estado interno do objecto receptor da mensagem

- o envio da mensagem *cx()* a cada um dos pontos 2D, origina a seguinte execução:



- importa referir que existe uma distinção entre mensagem e o resultado de tal envio
  - o resultado é activação do método correspondente

Construção de uma classe

- para a definição do objecto classe é necessário
  - identificar as variáveis de instância
  - identificar as diferentes operações que constituem o comportamento dos objectos instância

- declaração da estrutura:

```
/**
 * Write a description of class Ponto here.
 *
 * @author anr
 * @version 2010/11
 */
import static java.lang.Math.abs;

public class Ponto2D {

  // Variáveis de Instância
   private double x, y;

   // Construtores vazios
```

  - as variáveis de instância são privadas!
  - respeita-se o princípio do encapsulamento
- declaração do comportamento:
  - métodos de construção de instâncias
  - métodos de acesso às v. instância

- Construtores - métodos que são invocados quando se cria uma instância
  - não são métodos de instância, são métodos da classe!

```
// Construtores usuais
public Ponto2D(double x, double y) { this.x = x; this.y = y; }
public Ponto2D(){ this(0.0, 0.0); }  // usa o outro construtor
public Ponto2D(Ponto2D p) { x = p.getX(); y = p.getY(); }
```

- Utilização na criação de instâncias:
  - Ponto2D p = new Ponto2D(2.0,3.0);
  - Ponto2D r = new Ponto2D();

- métodos de acesso e alteração do estado interno
  - *getters* e *setters*
  - por convenção tem como nome getX() e setX(). Ex: getNotaAluno()

```
// Métodos de Instância
public double getX() { return this.x; }
public double getY() { return this.y; }

public void setX(double x) {this.x = x;}
public void setY(double y) {this.y = y;}
```

- outros métodos - decorrentes do domínio da entidade, isto é, o que representa e para que serve!

```
/** incremento das coordenadas */
public void incCoord(double dx, double dy) {
   this.x += dx; this.y += dy;
}
/** decremento das coordenadas */
public void decCoord(double dx, double dy) {
   this.x -= dx; this.y -= dy;
}

 /** soma as coordenadas do ponto parâmetro ao ponto receptor */
public void somaPonto(Ponto2D p) {
   this.x += p.getX(); this.y += p.getY();
}
/** soma os valores parâmetro e devolve um novo ponto */
public Ponto2D somaPonto(double dx, double dy) {
  return new Ponto2D(this.x += dx, this.y+= dy);
}
/* determina se um ponto é simétrico (dista do eixo dos XX o
    mesmo que do eixo dos YY */
public boolean simetrico() {
  return abs(this.x) == abs(this.y);
}

/** verifica se ambas as coordenadas são positivas */
public boolean coordPos() {
  return this.x > 0 && this.y > 0;
}
```

Utilização de uma classe

- uma classe teste, com um método main(), onde se criam instâncias e se enviam métodos
- um programa em POO é o resultado do envio de mensagens entre os objectos, de acordo com o alfabeto definido anteriormente

```
public class TestePonto {
  // Classe de teste da Classe Ponto.
  public static void main(String args[]) {
    // Criação de Instâncias
    Ponto pt1, pt2, pt3;
    pt1 = new Ponto();
    pt2 = new Ponto(2, -1);
    pt3 = new Ponto(0, 12);

    // Utilização das Instâncias
    int cx1, cx2, cx3;     // variáveis auxiliares
    int cy1, cy2, cy3;     // variáveis auxiliares
    cx1 = pt1.getx();
    cx2 = pt2.getx();

    // saída de resultados para verificação
    System.out.println("cx1 = " + cx1);
    System.out.println("cx2 = " + cx2);

    // alterações às instâncias e novos resultados

    pt1.incCoord(4,4); pt2.incCoord(12, -3);
    cx1 = pt1.getx(); cx2 = pt2.getx();
    cx3 = cx1 + cx2;
    System.out.println("cx1 + cx2 = " + cx3);

    pt3.decCoord(10, 20); pt2.decCoord(5, -10);
    cy1 = pt2.gety(); cy2 = pt3.gety();
    cy3 = cy1 + cy2;
    System.out.println("cy1 + cy2 = " + cy3);
  }
}
```

sobre classes e instâncias

- *"Classes and object are separate yet intimately related concepts. Specifically, every object is the instance of some class, and every class has zero or more instances. For practically all applications, classes are static; therefore, their existence, semantics, and relationships are fixed prior to the execution of a program. Similarly, the class of most objects is static, meaning that once an object is created, its class is fixed. In sharp contrast, however, objects are typically created and destroyed at a furious rate during the lifetime of an application."*

Definição do objecto classe

- Estado
  - identificação das variáveis de instância
- Comportamento
  - construtores/destrutores
  - getters e setters
  - outros métodos de instância, decorrentes do que representam

A referência this

- é usual precisarmos de referenciarmos o objecto que recebe a mensagem
- mas no contexto da escrita do código da classe ainda não sabemos como é que se vai chamar o objecto
- sempre que precisamos de ter acesso a uma variável do objecto podemos usar a referência *this*

- uma utilização muito normal é quando queremos desambiguar e identificar as variáveis de instância

- Por exemplo, em

```
// Métodos de Instância
public double getX() { return this.x; }
public double getY() { return this.y; }

public void setX(double x) {this.x = x;}
public void setY(double y) {this.y = y;}
```

- a utilização de *this* permite desambiguar a qual das variáveis nos estamos a referir

- a referência *this* pode ser utilizada para identificar um método da classe

```
// modificador - decrementa Coordenada X
void decCoordX(int deltaX) {
   this.decCoord(deltaX, 0);     // invoca decCoord() local
}
```

  - no caso de não termos escrito apenas decCoord(deltaX,0)

  - o compilador teria acrescentado automaticamente a referência *this*

Regras de acesso a variáveis e métodos

- a declaração das variáveis de instância pode ser precedida de informação sobre o nível de visibilidade

Modificador	Acessível a partir do código de
`public`	Qualquer classe
`protected`	Própria classe, qualquer classe do mesmo *package* e qualquer subclasse
`private`	Própria classe
nenhum	Própria classe e classes dentro do mesmo *package*

- para garantir o total encapsulamento do objecto as v.i. devem ser declaradas como **private**

- ao ter encapsulamento total é necessário garantir que existem métodos que permitem o acesso e modificação das v.i.

- os métodos que se pretendem que sejam visíveis do exterior devem ser declarados como **public**

A classe Aluno

- declaração das v.i.

```
/**
 * Classe Aluno.
 * Classe que modela de forma muito simples a
 * informação e comportamento relevante de um aluno.
 *
 * @author Antonio Nestor Ribeiro
 * @version 2006/03/20 (modificada Março 2009)
 */
public class Aluno {
    // instance variables
    private int numero;
    private int nota;
    private String nome;
```

- construtores: vazio, parametrizado e de cópia

```java
/**
 * Constructores para a classe Aluno
 */
public Aluno() {
  this.numero = 0;
  this.nota = 0;
  this.nome = "NA";
}

public Aluno(int numero, int nota, String nome) {
    this.numero = numero;
    this.nota = nota;
    this.nome = nome;
}


public Aluno(Aluno umAluno) {
    this.numero = umAluno.getNumero();
    this.nota = umAluno.getNota();
    this.nome = umAluno.getNome();
}
```

- métodos *getters* e *setters*

```
public int getNumero() {

    return this.numero;
}

public int getNota() {

    return this.nota;
}

public String getNome() {

    return this.nome;
}


public void setNota(int novaNota) {
    this.nota = novaNota;
}

private void setNumero(int numero) {
    this.numero = numero;
}

public void setNome(String nome) {
    this.nome = nome;
}
```

A classe Turma

- criação de um objecto que permita guardar instâncias de Aluno
- como estrutura de dados vamos utilizar um array de objectos do tipo Aluno
  - Aluno alunos[]
- A utilização de Aluno na definição de Turma corresponde à utilização de **composição** na definição de objectos mais complexos

- declação das v.i.

```java
import java.util.*;

/**
 * Primeira implementação de uma turma de alunos.
 * Assume que a turma é mantida num array.
 *
 * @author António Nestor Ribeiro
 * @version 2006/03/20
 * @version 2009/03/30
 */
public class Turma
{
    private String designacao;
    private Aluno[] lstalunos;
    private int capacidade;

    //variaveis internas para controlo do numero de alunos
    private int ocupacao;


    //se não for especificado o tamanho da turma usa-se esta constante
    private static final int capacidade_inicial = 20;
```

- construtores

```
/**
 * Constructor for objects of class Turma
 */
public Turma()
{
    this.designacao = new String();
    this.lstalunos = new Aluno[capacidade_inicial];
    this.capacidade = capacidade_inicial;
    this.ocupacao = 0;
}

public Turma(String designacao, int tamanho) {
     this.designacao = designacao;
     this.lstalunos = new Aluno[tamanho];
     this.capacidade = tamanho;
     this.ocupacao = 0;
}


public Turma(Turma outraTurma) {
    this.designacao = outraTurma.getDesignacao();
    this.capacidade = outraTurma.getCapacidade();
    this.ocupacao = outraTurma.getOcupacao();
    this.lstalunos = outraTurma.getLstAlunos();
}
```

```java
public String getDesignacao() {
    return this.designacao;
}

public int getCapacidade() {

    return this.capacidade;
}

public int getOcupacao() {
    return this.ocupacao;
}

/**
 * Método privado (auxiliar)
 *
 * Possível "buraco negro"!!! Como resolver?
 */
private Aluno[] getLstAlunos() {
    return this.lstalunos.clone(); //!!!
}
```

- o método getLstAlunos é auxiliar e privado

- inserir um novo Aluno

```
/**
 * Este método assume que se verifique previamente se
 * ainda existe espaço para mais um aluno na turma.
 *
 * Em futuras versões desta classe poderemos fazer internamente a
 * gestão das situações de erro. Neste momento assume-se que a
 * pré-condição é verdadeira.
 *
 * Este método deverá ser reescrito em futuras implementações
 * para evitar potenciais quebras de encapsulamento - já feito ao
 * agregar uma CÓPIA do aluno passado como parâmetro.
 *
 */
public void insereAluno(Aluno umAluno) {


    this.lstalunos[this.ocupacao] = new Aluno(umAluno); //encapsulamento garantido
    this.ocupacao++;
}
```

- utiliza-se o construtor de cópia de Aluno.
  - porquê?!

- Como implementar os métodos
  - public boolean existeAluno(Aluno a)
  - public void removeAluno(Aluno a)
- como é que determinamos se o objecto está efectivamente dentro do array de alunos?

- A solução
  - lstalunos[i] == a, não é eficaz porque compara os apontadores
  - (lstalunos[i]).getNumero() == a.getNumero(), é assumir demasiado sobre a forma como se comparam alunos
- Quem é a melhor entidade para determinar como é que se comparam objectos do tipo Aluno?

- através da disponibilização de um método, na classe Aluno, que permita comparar instâncias de alunos
  - é importante que esse método seja universal, isto é, que tenha sempre a mesma assinatura
  - é importante que todos os objectos respondam a este método
- **public boolean equals(Object o)**

- para a implementação inicial, na classe Aluno, vamos considerar que o parâmetro é do tipo Aluno

```
/**
 * 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(Aluno umAluno) {
     if (umAluno != null)
       return(this.nome.equals(umAluno.getNome()) && this.nota == umAluno.getNota()
              && this.numero == umAluno.getNumero());
     else
       return false;

 }
```

- dessa forma o método existeAluno(Aluno a) da classe Turma, assume a seguinte forma:

```
/**
 * De acordo com o funcionamento tipo destes métodos,
 * vai-se percorrer o array e enviar o método equals a cada objecto
 */

public boolean existeAluno(Aluno umAluno) {
    boolean resultado = false;

    if (umAluno != null) {

      for(int i=0; i< this.ocupacao && !resultado; i++)
        resultado = this.lstalunos[i].equals(umAluno);

      return resultado;
    }
    else
      return false;
}
```

- Em resumo:
  - método de igualdade é determinante para que sejam possível ter colecções de objectos
  - o método de igualdade não pode codificado a não ser pela classe
  - existem um conjunto de regras básicas que todos os métodos de igualdade devem respeitar

O método equals

- a assinatura é:
  - **public boolean equals(Object o)**
- é importante referir, antes de explicar em detalhe o método, que:
  - não se comparam objectos de classes diferentes, logo o método deve fazer cast para o tipo de dados que estamos a trabalhar

O método equals

- a relação de equivalência que o método implementa é:
- é **reflexiva**, ou seja x.equals(x) == true, para qualquer valor de x que não seja nulo
- é **simétrica**, se para valores não nulos de x e y, x.equals(y) == true, então y.equals(x) == true

- é **transitiva**, em que para x,y e z, não nulos, se x.equals(y) == true, y.equals(z) == true, então x.equals(z) == true

- é **consistente**, dado que para x e y não nulos, sucessivas invocações do método equals (x.equals(y) ou y.equals(x)) dá sempre o mesmo resultado

- para valores nulos, a comparação com x, não nulo, dá como resultado false.

- quando os objectos envolvidos sejam o mesmo, o resultado é true, ie, x.equals(y) == true, se x == y
  - dois objectos são iguais se forem o mesmo, ie, se tiverem o mesmo apontador
- caso não se implemente o método equals, temos uma implementação, por omissão, com o seguinte código:

```
public boolean equals(Object object) {
        return this == object;
}
```

- esqueleto típico de um método equals

```
public boolean equals(Object o) {

    if (this == o)
      return true;

    if((o == null) || (this.getClass() != o.getClass()))
      return false;

    <CLASSE> m = (<CLASSE>) o;
    return ( <condições de igualdade> );

}
```

- o método equals da classe Aluno

```
/**
 * 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
     else {
       Aluno a =  (Aluno) umAluno;
       return(this.nome.equals(a.getNome()) && this.nota == a.getNota()
              && this.numero == a.getNumero());
     }

 }
```

- como é que será o método equals da classe Turma?

- quais as consequências de não ter o método equals implementado??
  - consideremos que Aluno não tem equals
  - o que acontece neste método de Turma?

```
/**
 * De acordo com o funcionamento tipo destes métodos,
 * vai-se percorrer o array e enviar o método equals a cada objecto
 */

public boolean existeAluno(Aluno umAluno) {
    boolean resultado = false;

    if (umAluno != null) {

      for(int i=0; i< this.ocupacao && !resultado; i++)
        resultado = this.lstalunos[i].equals(umAluno);

      return resultado;
    }
    else
      return false;
}
```

- existem contudo outros métodos que tem na plataforma Java uma filosofia semelhante ao método de igualdade
  - **toString**, que deve transformar a representação interna de um objecto numa String
  - **clone**, que tem como objectivo devolver uma cópia do objecto a quem é enviado

O método toString

- a informação deve ser concisa (sem *acuçar de ecran*), mas ilustrativa
- todas as classes devem implementar este método
- caso não seja implementado a resposta será:

getClass().getName() + '@' + Integer.toHexString(hashCode())

O método toString

- implementação normal de toString na classe Aluno

```
/**
 * Implementação do método toString
 * comum na maioria das classes Java
 *
 * @return     uma string com a informação textual do objecto aluno
 */
public String toString() {

    return("Numero:" + this.numero + "Nome:"+ this.nome + "Nota:" + this.nota);

}
```

- o operador "+" é a concatenação de Strings, sempre que o resultado seja uma String

- o mesmo método, de forma mais eficiente, na medida em que as concatenações de Strings são muito pesadas
  - Strings são objectos imutáveis, logo não crescem, o que as torna muito ineficientes

```
/**
 * Implementação do método toString
 * comum na maioria das classes Java
 *
 * @return     uma string com a informação textual do objecto aluno
 */
public String toString() {
    StringBuffer sb= new StringBuffer();

    sb.append("Numero: ");
    sb.append(this.numero+"\n");
    sb.append("Nome: ");
    sb.append(this.nome+"\n");
    sb.append("Nota: ");
    sb.append(this.nota+"\n");

    return sb.toString();


}
```

O método clone

- este método tem como objectivo a criação de uma cópia do objecto a quem é enviado
  - a nocão de cópia depende muito da classe que faz a implementação
  - a noção geral é que x.clone() != x
  - sendo que,

    x.clone().getClass() == x.getClass()

O método clone

- regra geral, e de acordo com a visão em POO, a expressão seguinte deve prevalecer

  x.clone().equals(x),

  - embora isso dependa muito da forma como ambos os métodos estão implementados

- a implementação de clone é relativamente simples

O método clone

- na metodologia de POO já temos um método que faz cópia de objectos
  - o construtor de cópia de cada classe
- Dessa forma podemos dizer que apenas temos de invocar esse construtor e passar-lhe como referência o objecto que recebe a mensagem - neste caso o *this*

O método clone

- implementação do método clone da classe Aluno

```
/**
* Implementação do método de clonagem de um Aluno
*
* @param  umAluno   aluno que é comparado com o receptor
* @return  objecto do tipo Aluno
* ** * */

public Aluno clone() {
  return new Aluno(this);
}
```

- optamos por devolver um objecto do mesmo tipo de dados e não Object como é a definição padrão do Java.

Clone vs Encapsulamento

- a utilização de clone() permite que seja possível preservarmos o encapsulamento dos objectos, desde que:
  - seja feita uma cópia dos objectos à entrada dos métodos
  - seja devolvida uma cópia dos objectos e não o apontador para os mesmos

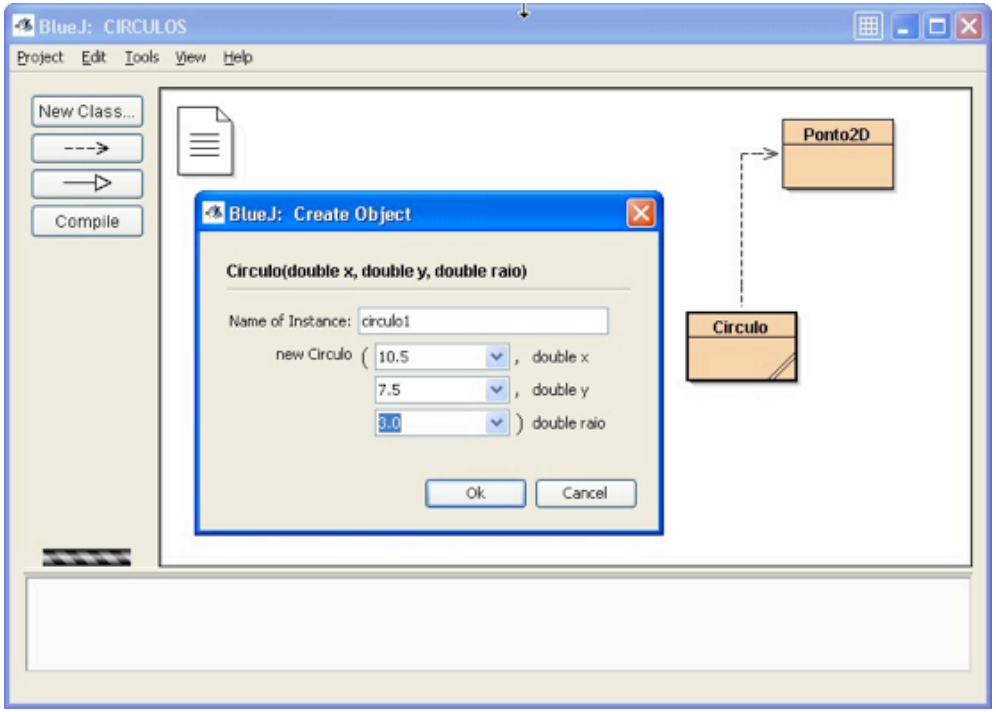
- Considere-se a seguinte definição da Classe Circulo

```
import static java.lang.Math.PI;
public class Circulo {

   private double raio;     // o raio do círculo
   private Ponto2D centro;  // ponto que define o centro do círculo

   // Construtores de circulos
   ..........
```

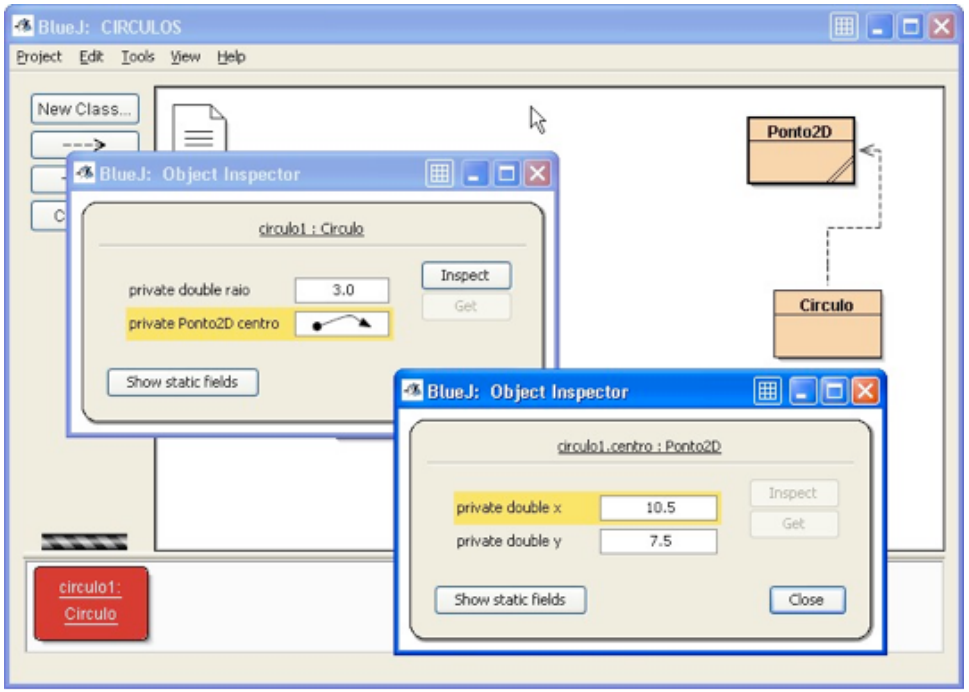
- e seja o método, daCentro() definido como:

```
public Ponto2D daCentro() {return this.centro;}
```

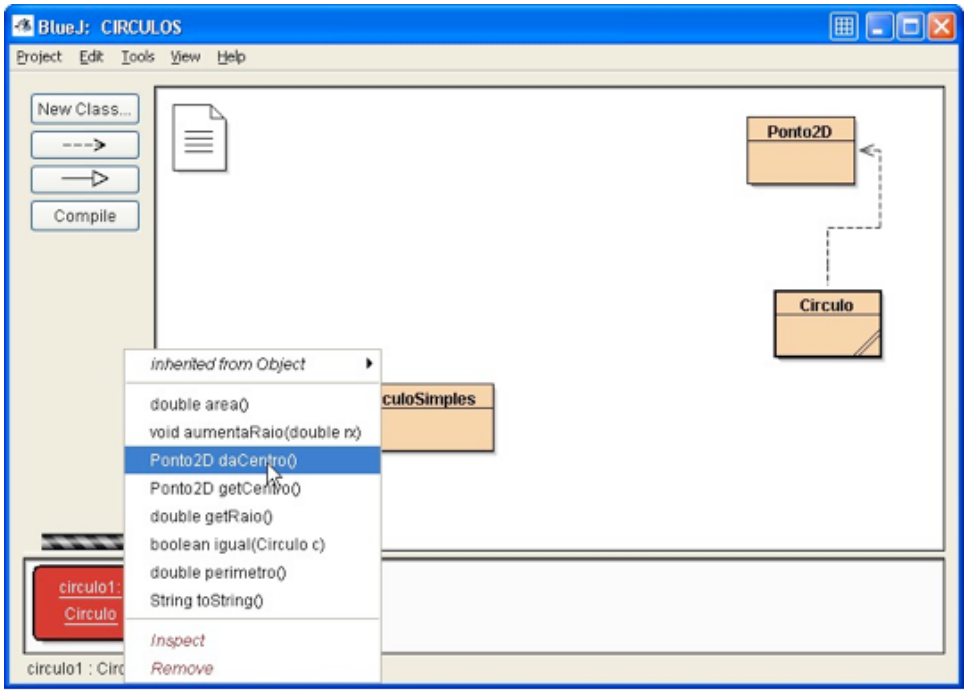
Criação de um círculo



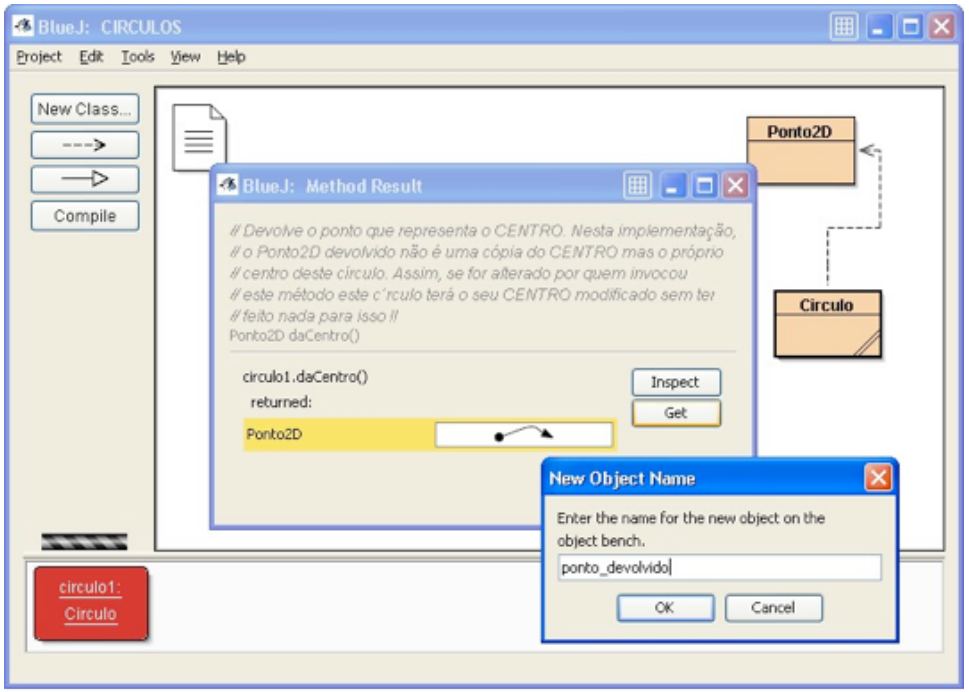
Inspect do objecto



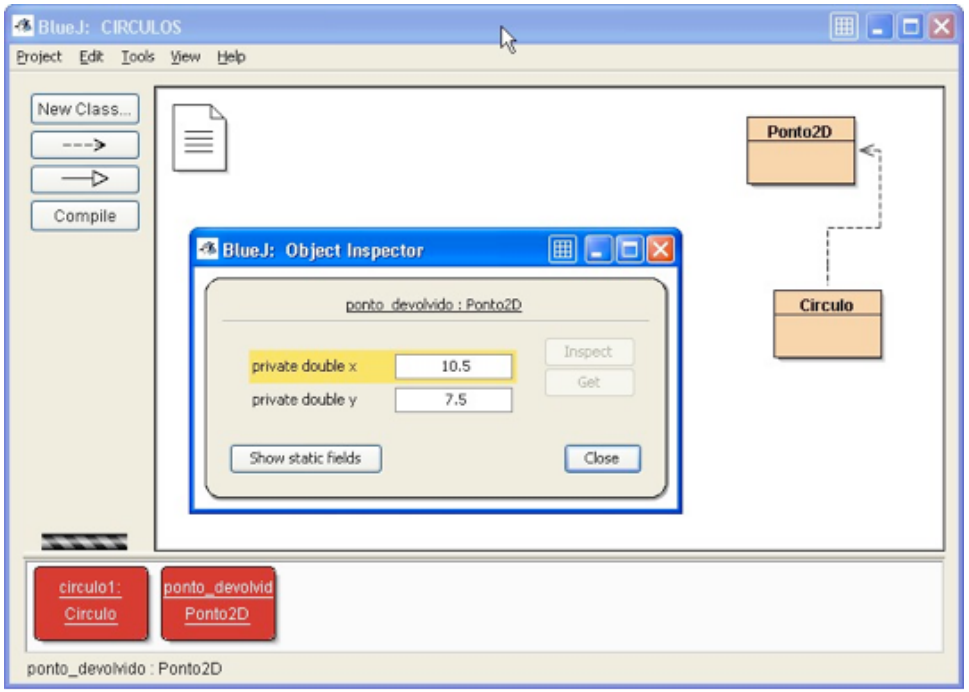
Invocação de daCentro()



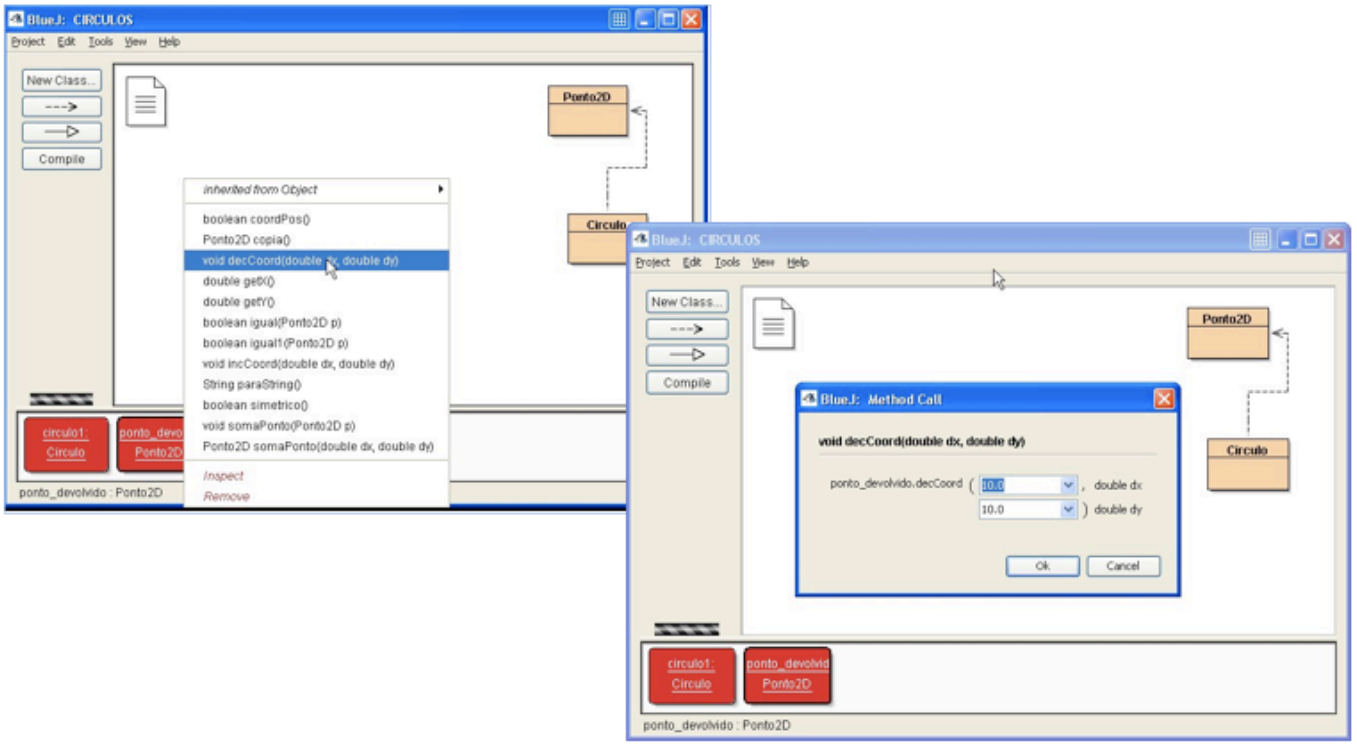
Ficar com o objecto devolvido



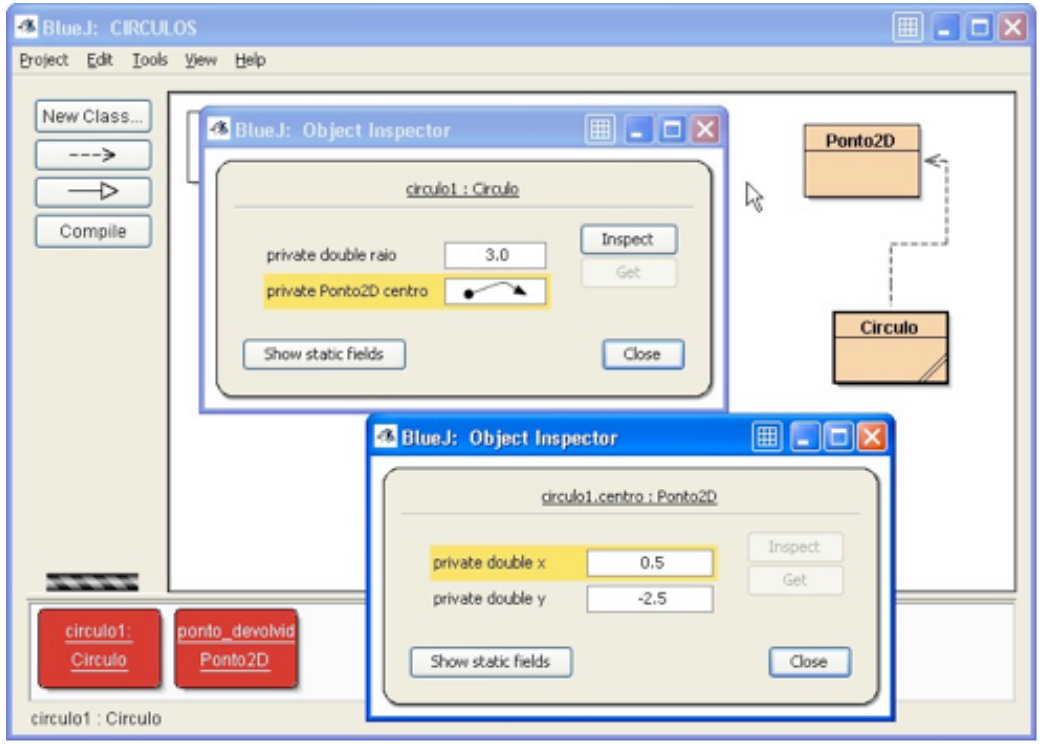
Inspect do resultado



Alterar o ponto que obtivemos



O círculo foi alterado!



A clonagem de objectos

- Duas abordagens:
  - *shallow* clone: cópia parcial que deixa endereços partilhados
  - *deep* clone: cópia em que nenhum objecto partilha endereços com outro

- A sugestão é utilizar sempre *deep* clone, na medida em que podemos controlar todo o processo de acesso aos dados

- **REGRA**: clone do todo = "soma" do clone das partes

  - tipos simples e objectos imutáveis (String, Integer, Float, etc.) não precisam (não devem!) ser clonados.

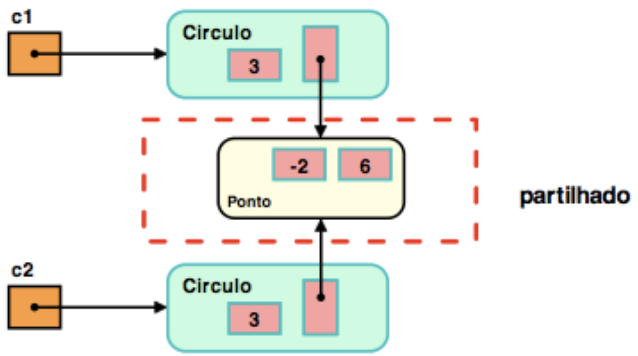
- A saber:
  - implementar o clone como sendo uma invocação do construtor de cópia

```
public Ponto2D getCentro() {
    return centro.clone(); // cria um novo Ponto2D, cópia do centro !!
}
```

  - o método clone() existente nas classes Java é sempre *shallow*, e devolve sempre um Object, logo é necessário fazer cast
  - os clones que vamos fazer nas nossas classes, devolvem sempre um tipo de dados da classe

- Exemplificação de um *shallow clone*

```
c2 = c1.clone1();
```



- existem conteúdos partilhados

...completar a classe Turma

- equals

```
/**
 * Método equals.
 * Utiliza o método privado getLstAlunos para efectuar a comparação entre
 * duas instância de turma.
 */

public boolean equals(Turma umaTurma) {
    if (umaTurma != null)
      return (this.designacao.equals(umaTurma.getDesignacao())
              && this.capacidade == umaTurma.getCapacidade()
              && this.ocupacao == umaTurma.getOcupacao()
              && Arrays.equals(this.lstalunos,umaTurma.getLstAlunos()));
    else
      return false;
}
```

  - nesta versão recorreu-se ao método equals da classe Arrays