Programação Orientada aos Objectos

MiEI/LCC - 2º ano 2019/20

António Nestor Ribeiro
(editado por J.C. Campos em 2015/16)

Conteúdos baseados em elementos de:

1. JAVA6 e Programação Orientada pelos Objectos
F. Mário Martins, Editora FCA, Série Tecnologias de Informação, Julho de 2009. (e posteriores revisões, eg: Java 8 - POO + Construções Funcionais)

2. **Objects First with Java – A Practical Introduction using BlueJ**, Sixth edition
David J. Barnes & Michael Kölling, Pearson, 2016.

3. Object Oriented Design with Applications
G. Booch, The Benjamim Cummings Pub. Company, USA, 1991

4.Java Program Design - Principles, Polymorphism, and Patterns
Edward Sciore, Apress Media, ISBN 978-1-4842-4142-4, 2019

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:

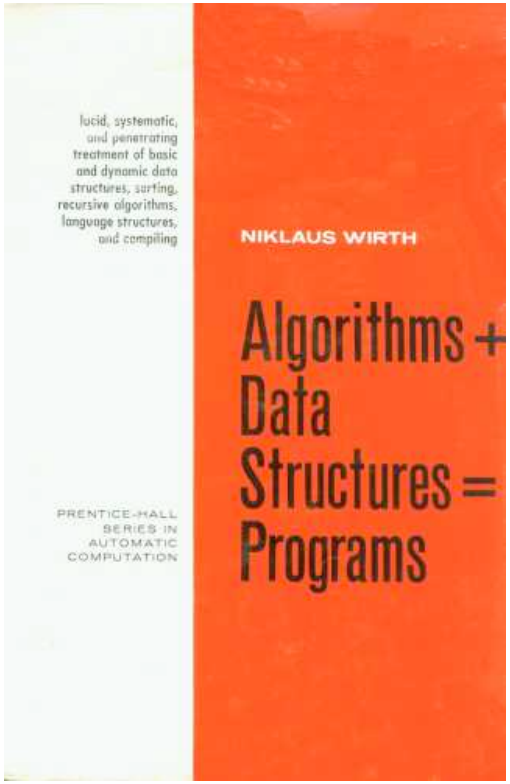
- as acções representam as entidades computacionais de 1ª classe, os algoritmos, a lógica computacional.
- os dados são entidades secundárias, as estruturas de dados que as funções e procedimentos “visitam”.

Estratégia Top-Down

Consiste num refinamento progressivo dos processos

- assente numa lógica de dividir a complexidade, uma forma de "dividir para reinar"
- na concepção de um sistema complexo é importante decompô-lo em partes mais pequenas e mais simples
- esta divisão representa uma abordagem inteligente, na medida em que se pode trabalhar cada pedaço *per* se

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 (o que é que se reutiliza? pedaços de código? funções?)

É importante reter a noção de **reutilização** de software, como mecanismo de aproveitamento de código já desenvolvido e aplicado noutros projectos.

Um exemplo: o "caso das lista ligadas":

quantas vezes é que já fizemos, em contextos diferentes, código similar para implementar uma lista ligada de "coisas"?

porque é que não se reutiliza código? Código muito orientado aos dados: uma LL de Alunos é sempre diferente de uma LL de Carros.

porquê? o que muda?

porque é que não temos uma implementação genérica?

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:

- função que dado um array de alunos os ordena por ordem crescente de nota
- função que dados dois alunos devolve o menor (alfabeticamente) deles

estes mecanismos suportam reutilização no contexto de um programa

reutilização entre programas: “copy&paste”

a reutilização está muito dependente dos tipos de dados de entrada e saída: quase sempre implica mexer nos tipos de dados

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 (em C por exemplo os .o)

o módulo como abstracção procedimental:

```
//--- ESTRUTURAS DE DADOS ----------------------------------------------------------

struct elemento {
    void *dados;
    struct elemento *proximo;
};

struct lista {
    size_t tamanho_dados;
    struct elemento *elementos;
};

//--- TYPEDEF ----------------------------------------------------------------------

typedef struct lista Lista;

//--- FUNCOES ----------------------------------------------------------------------

void inicia_sll(struct lista *,size_t);
int insere_cabeca_sll(struct lista *,void *);
int insere_ord_sll(struct lista *,void *,int (void *,void *));
int apaga_sll(struct lista *,void *,int (void *,void *));
void destroi_sll(struct lista *);
int procura_sll(struct lista ,void *,void *v,int (void *,void *));
void aplica_sll(struct lista ,void (void *));
void filtro_sll(struct lista *,void *,int (void *,void *));

//--- FIM --------------------------------------------------------------------------
```

no entanto, este modelo não garante a estanquicidade dos dados

os procedimentos de um módulo podem aceder aos dados de outros módulos

Por vezes, apesar de termos módulos estes conhecem-se e acedem aos dados uns dos outros:

- problemas vários ao nível de dependência entre os módulos (até na compilação dos mesmos)

a partilha de dados quebra as vantagens de uma possível reutilização

numa situação de utilização de uma solução destas os diversos módulos 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
- não exibem publicamente informação que permita o conhecimento da sua implementação

A estrutura de dados local passa a estar completamente escondida: **Data Hiding**

Passamos a disponibilizar serviços (a que chamaremos de API) que possibilitam que do exterior se possa obter informação acerca dos dados

Desta forma, os módulos passam assim a ser vistos como mecanismos de **abstracção de dados**

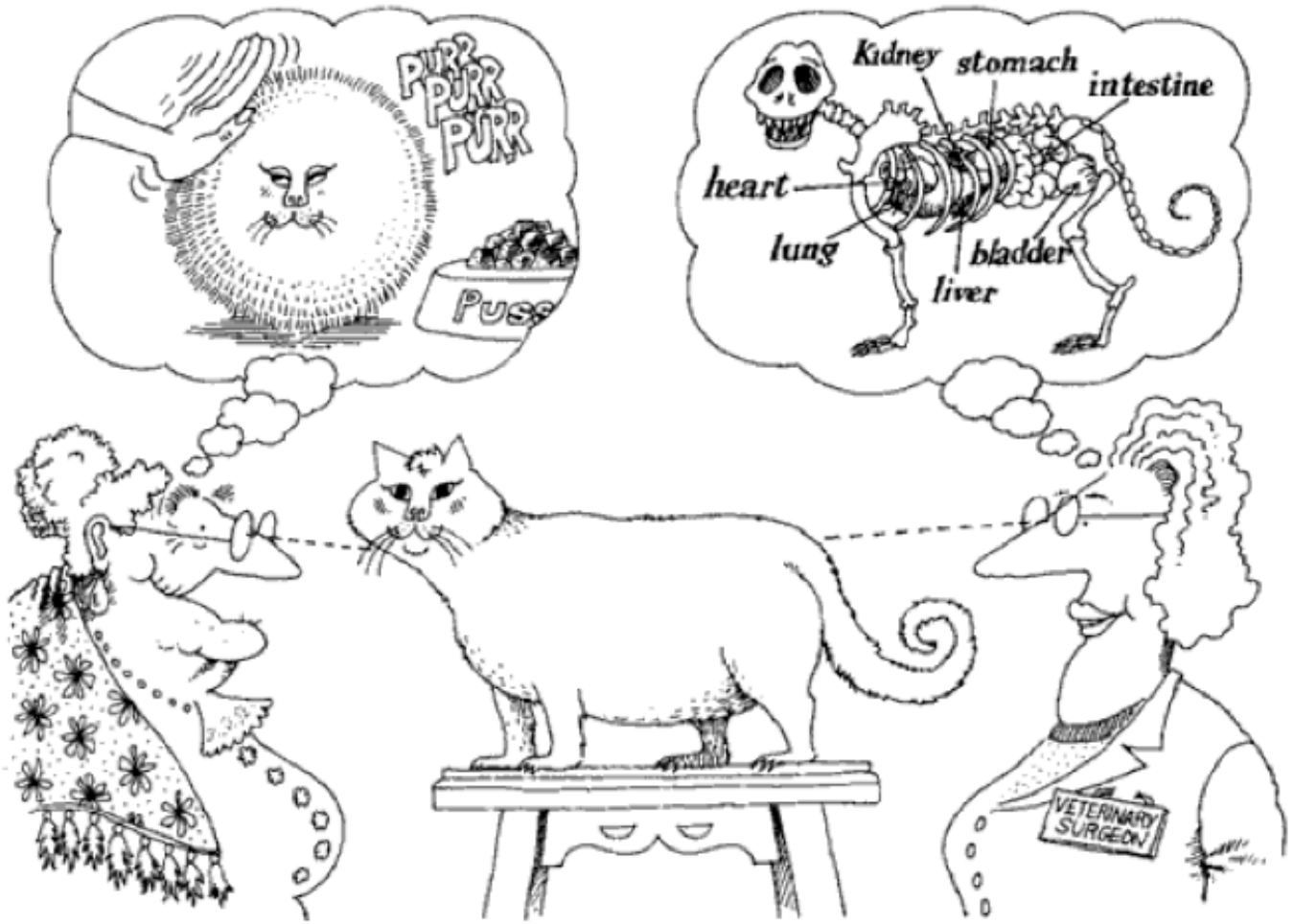
Nesta abordagem o módulo passa a assumir apenas a informação necessária para que possa ser utilizado.

```
int init (int size, int (*compare)(void *, void *));
int insert (int handle, void * data);
int search (int handle, void *data);
int remove (int handle, void *data);
int clean (int handle);
```

apenas se conhece a interface (API) e não se sabe mais nada da implementação

neste caso, o programa cliente apenas tem acesso a um *handle* que é o apontador para o início da lista

Abstracção



cada actor tem a visão que mais lhe convém (interessa)

(OOAD with Applications, Grady Booch)

se os módulos forem construídos com estas preocupações, então passamos a ter:

- capacidade de reutilização
- encapsulamento de dados
- possibilidade de termos alteração dos dados sem impacto nos programas clientes
  - ex: numa primeira fase podemos ter uma turma como sendo um array de alunos e depois (sem anúncio) mudar para uma lista ligada

Programação com TAD

Consideremos a seguinte definição de um TAD Aluno (por conveniência de escrita escrito em Java, e numa forma não completa,)

```
public class Aluno {
    String nome;
    String numero;
    String curso;
    double media;

    public Aluno(String nome, String numero, String curso, double media) {...}
    public String getNome() {...}
    public void actualizaNome(String novoNome) {...}
    public String getNumero() {...}
    public String getCurso() {...}
    public void actualizaCurso(String novoCurso) {...}

    ...
}
```

A utilização correcta deste tipo de dados é aquela que apenas utiliza a API para aceder à informação, cf:

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

  Aluno a1 = new Aluno("alberto alves", "a55255", "MiEI", 12.5);
  Aluno a2 = new Aluno("marisa pinto", "pg20255", "LMat", 15.3);

  System.out.println("Curso do Aluno número:" a1.getNumero() + " = " + a1.getCurso());
  System.out.println("Curso do Aluno número:" a2.getNumero() + " = " + a2.getCurso());

  ....
  ....
  a2.actualizaCurso("MiEngCivil");
  ...

}
```

o acesso ao tipo de dados Aluno é feito apenas via a API definida.

alterações nas variáveis internas do tipo de dados não tem impacto no cliente

Uma solução incorrecta, no que concerne à utilização dos módulos como tipos abstractos de dados, seria a que acederia directamente ao estado interno. Cf:

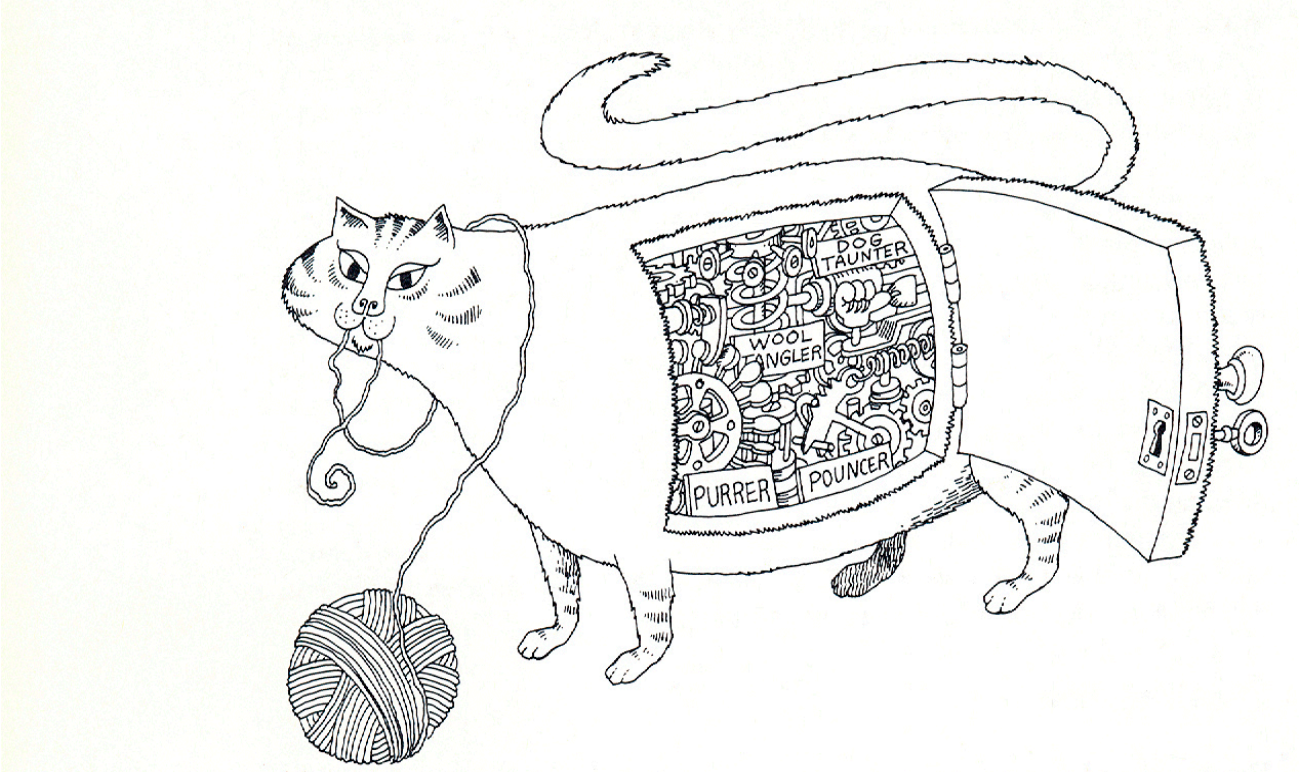
```
....
....
a2.curso = "MiEngCivil";
...
```

uma utilização destas torna a definição Aluno não reutilizável, visto que não existe a capacidade de a evoluir de forma autónoma das aplicações cliente.

- não se respeita o encapsulamento dos dados

Encapsulamento

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



Encapsulation hides the details of the implementation of an object.

(OOAD with Applications, Grady Booch)

Metodologia

criar o módulo pensando no tipo de dados que se vai representar e manipular

definir as estruturas de dados internas que se devem criar

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

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, nomeadamente para aplicações de simulação (trânsito, filas espera, etc.)

o objectivo era representar entidades do mundo real:

- identidade (única)
- estrutura (atributos)
- comportamento (acções e reacções)
- interacção (com outras entidades)

Introduz-se 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/padrões/formas/ templates de *indivíduos*

a que chamaremos de "objectos"!

Passagem para POO

Um objecto é a representação computacional de uma entidade do mundo real, com:

atributos (necessariamente) 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)

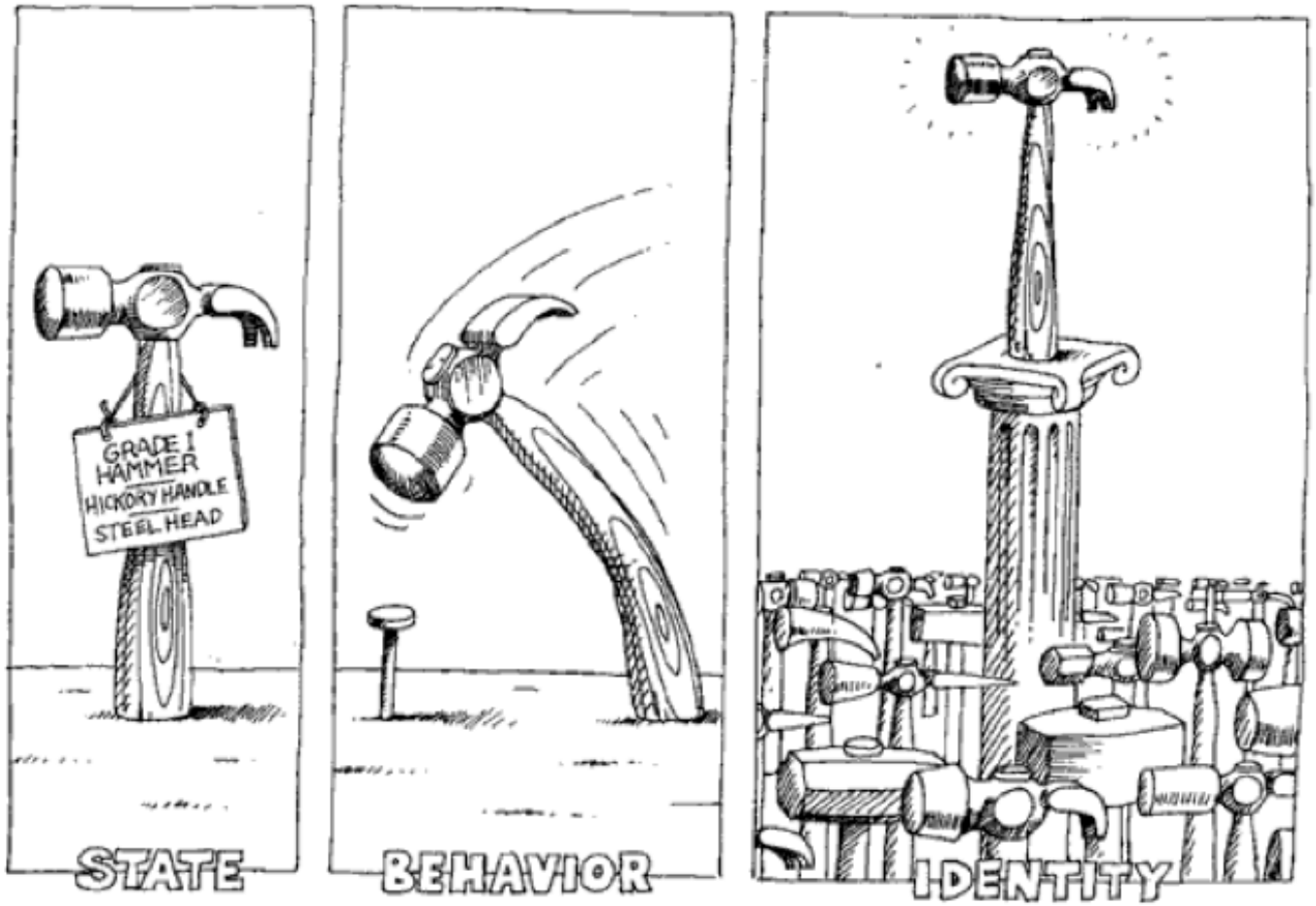
um objecto é o módulo computacional básico e único e tem como características:

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



(OOAD with Applications, Grady Booch)

Objecto = “black box” (fechado, opaco)

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

Dados Privados
(v. instância)

método 1

API pública

método 2

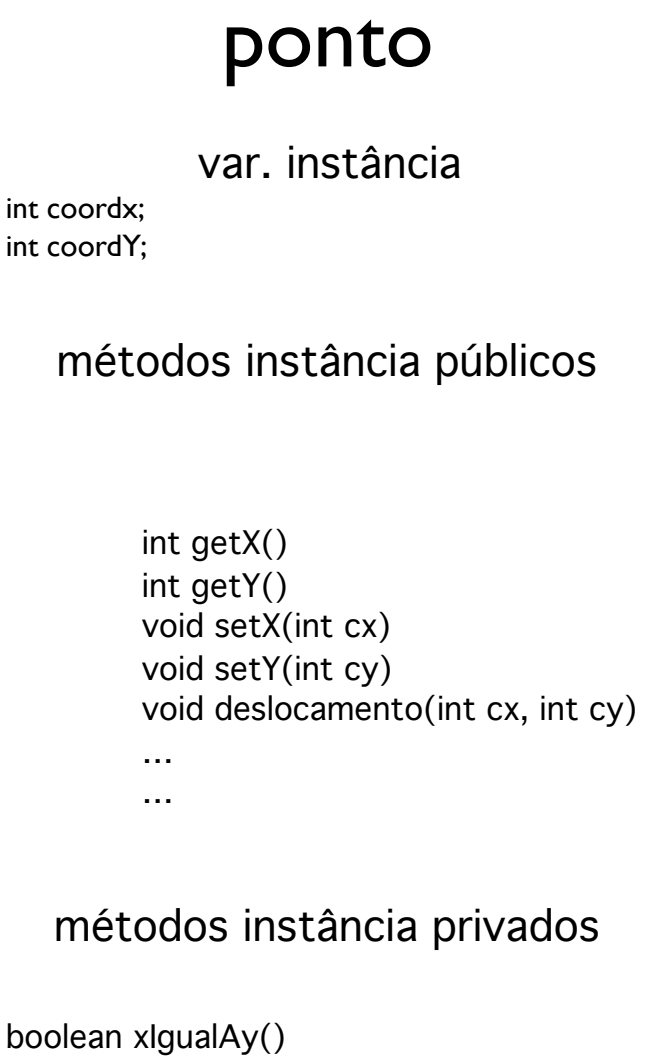
método 3

método 4

método privado

Um objecto que representa um ponto (com coordenadas no espaço 2D)

o método `xIgualAy()` é privado e não pode ser invocado por entidades externas ao objecto

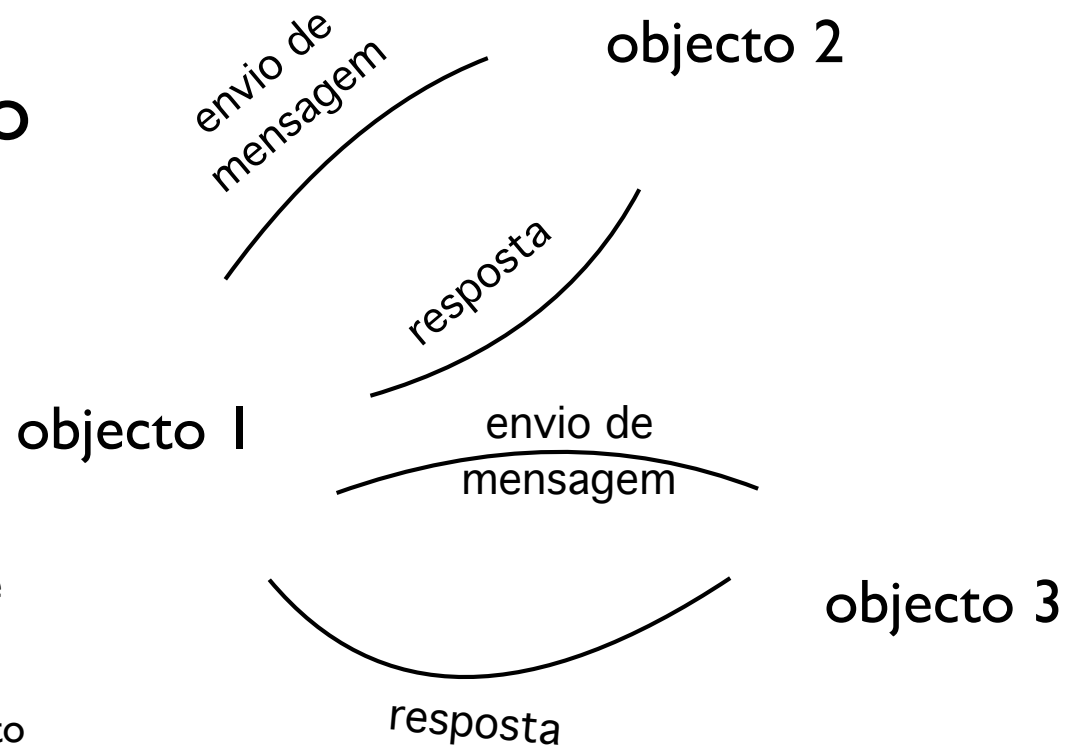


Um objecto é:

- 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



dependendo da invocação de método especificada, o envio da mensagem origina uma resposta por parte do objecto receptor.

Novo alfabeto

o facto de termos agora um alfabeto de mensagens a que cada objecto responde, condiciona 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 = objecto.var

objecto.var = x

em que se acede, de forma directa e não protegida (não encapsulada), ao campo **var** da estrutura interna do **objecto**

Se **ag** for um objecto que represente uma agenda, então poderemos fazer:

ag.insereEvento("TestePOO",21,5,2020), para inserir um novo evento na agenda

ag.libertaEventosDia(25,4,2020), para remover todos os eventos de um dia

String[] ev = ag.getEventos(6,2020), para obter as descrições de todos os eventos do mês de Junho

…definição 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, template) 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 *tipo abstracto de dados* onde se especifica, quer a estrutura quer o comportamento das instâncias, que são criadas a partir dela

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

...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

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

Exemplo: Classe Ponto

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, que pertençam à lógica de negócio expectável dos pontos

declaração da estrutura:

```
/**
 * Classe que implementa um Ponto num plano2D.
 * As coordenadas do Ponto são inteiras.
 *
 * @author anr (MaterialPOO)
 * @version 20180212
 */
public class Ponto {

  //variáveis de instância
  private int x;
  private int y;
```

- as variáveis de instância são privadas!
- respeita-se o princípio do encapsulamento

declaração do comportamento:

- métodos de construção de instâncias
- métodos de acesso/manipulação das variáveis de instância

Construtores - métodos que são invocados quando se cria uma instância

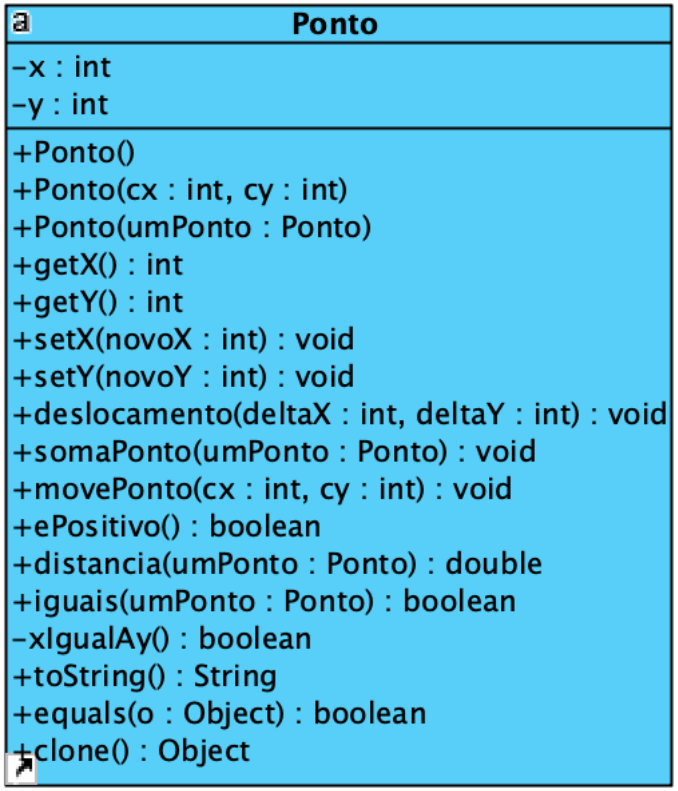
não são métodos de instância, são métodos da classe!

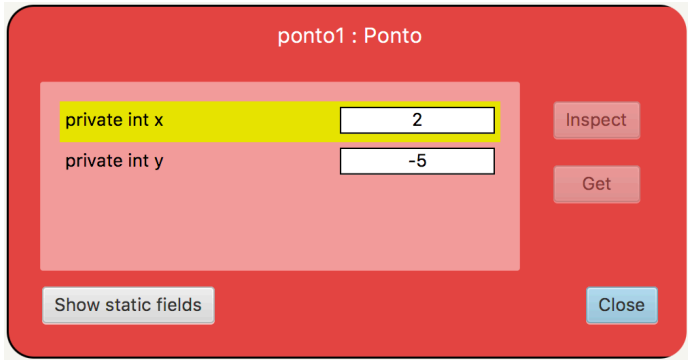
```java
/**
 * Construtor por omissão de Ponto.
 */
public Ponto() {
  this.x = 0;
  this.y = 0;
}
```

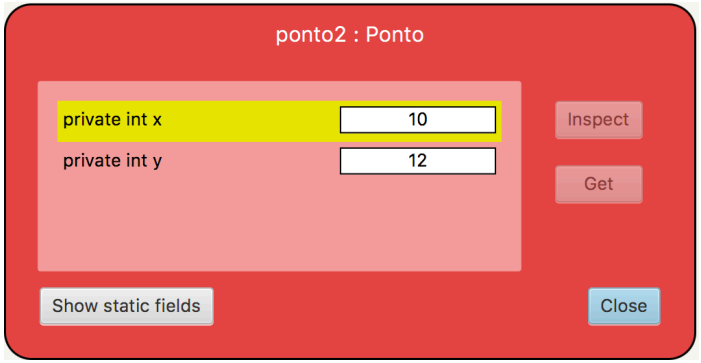
```java
/**
 * Construtor parametrizado de Ponto.
 * Aceita como parâmetros os valores para cada coordenada.
 */
public Ponto(int cx, int cy) {
  this.x = cx;
  this.y = cy;
}
```

```java
/**
 * Construtor de cópia de Ponto.
 * Aceita como parâmetro outro Ponto e utiliza os métodos
 * de acesso aos valores das variáveis de instância.
 */
public Ponto(Ponto umPonto) {
  this.x = umPonto.getX();
  this.y = umPonto.getY();
}
```

a classe Ponto e duas instâncias (ponto1 e ponto2):







ponto1 e *ponto2* são instâncias diferentes, mas possuem o mesmo comportamento

não faz sentido replicar o comportamento por todos os objectos

cada instância apenas tem de ter a representação dos valores das suas variáveis de instância

a informação do comportamento, os métodos, está guardada no objecto classe

métodos de acesso e alteração do estado interno

getters e *setters*

por convenção, tem como nome getX() e setX().

getX(), nas definições de Ponto

getNota(), nas definições de Aluno

getReal(), nas definições de NúmeroComplexo

etc.

```java
/**
 * Devolve o valor da coordenada em x.
 *
 * @return valor da coordenada x.
 */
public int getX() {
  return this.x;
}
```

```java
/**
 * Devolve o valor da coordenada em y.
 *
 * @return valor da coordenada y.
 */
public int getY() {
  return this.y;
}
```

```java
/**
 * Actualiza o valor da coordenada em x.
 *
 * @param novoX novo valor da coordenada em X
 */
public void setX(int novoX) {
  this.x = novoX;
}
```

```java
/**
 * Actualiza o valor da coordenada em y.
 *
 * @param novoY novo valor da coordenada em Y
 */
public void setY(int novoY) {
  this.y = novoY;
}
```

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

```java
/**
 * Método que desloca um ponto somando um delta às coordenadas
 * em x e y.
 *
 * @param deltaX valor de deslocamento do x
 * @param deltaY valor de deslocamento do y
 */
public void deslocamento(int deltaX, int deltaY) {
  this.x += deltaX;
  this.y += deltaY;
}
```

```java
/**
 * Método que soma as componentes do Ponto passado como parâmetro.
 * @param umPonto ponto que é somado ao ponto receptor da mensagem.
 */
public void somaPonto(Ponto umPonto) {
  this.x += umPonto.getX();
  this.y += umPonto.getY();
}
```

```java
/**
 * Método que move o Ponto para novas coordenadas.
 * @param novoX novo valor de x.
 * @param novoY novo valor de y.
 */
public void movePonto(int cx, int cy) {
  this.x = cx;  // ou setX(cx)
  this.y = cy;  // ou this.setY(cy)
}
```

```java
/**
 * Método que determina se o ponto está no quadrante positivo de x e y
 * @return booleano que é verdadeiro se x>0 e y>0
 */
public boolean ePositivo() {
  return (this.x > 0 && this.y > 0);
}

/**
 * Método que determina a distância de um Ponto a outro.
 * @param umPonto ponto ao qual se quer determinar a distância
 * @return double com o valor da distância
 */
public double distancia(Ponto umPonto) {

  return Math.sqrt(Math.pow(this.x - umPonto.getX(), 2) +
                   Math.pow(this.y - umPonto.getY(), 2));
}

/**
 * Método que devolve a representação em String do Ponto.
 * @return String com as coordenadas x e y
 */
public String toString() {
  return "Cx = " + this.x + " Cy = " + this.y;
}
```

```java
/**
 * Método que determina se dois pontos são iguais.
 * @return booleano que é verdadeiro se os valores das duas
 * coordenadas forem iguais
 */
public boolean iguais(Ponto umPonto) {
  return (this.x == umPonto.getX() && this.y == umPonto.getY());
}
```

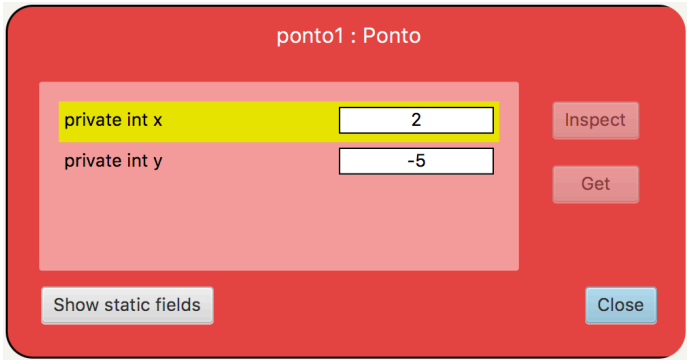
```java
/**
 * Método que determina se o módulo das duas coordenadas é o mesmo.
 * @return true, se as coordenadas em x e y
 * forem iguais em valor absoluto.
 */
private boolean xIgualAy() {
  return (Math.abs(this.x) == Math.abs(this.y));
   }
```

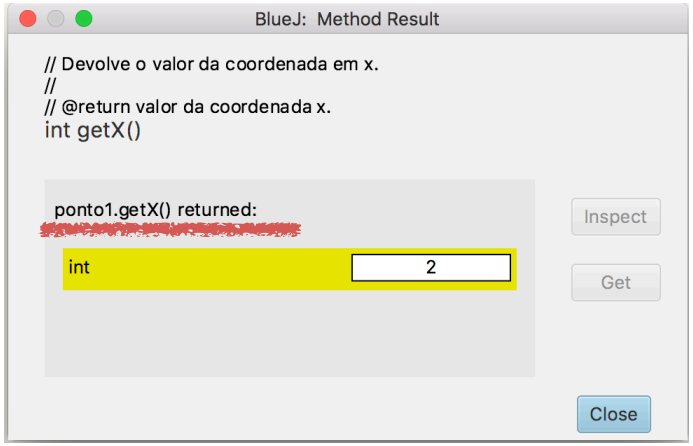
Modelo de execução dos métodos

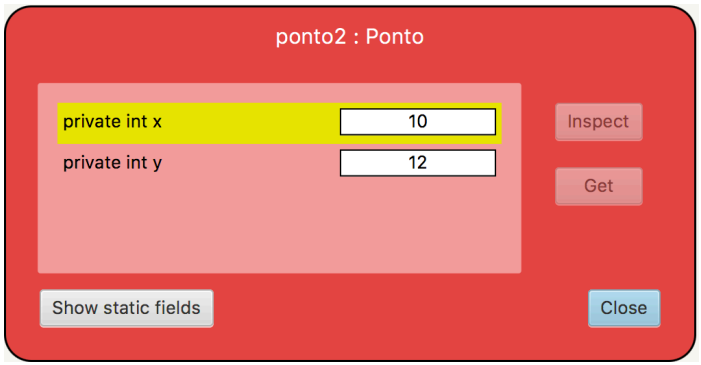
quando uma instância de uma classe recebe uma dada mensagem, solicita à sua classe a execução do método correspondente

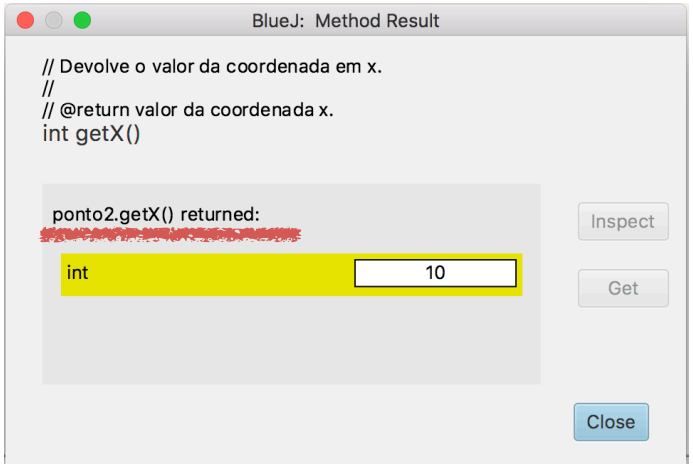
- os valores a utilizar na execução são os do estado interno do objecto receptor da mensagem

o envio da mensagem `getX()` a cada um dos pontos, 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
- o método é executado no contexto do objecto receptor, utilizando os valores do estado interno do objecto.
  - daí, o mesmo método ter resultados diferentes consoante o objecto receptor

Utilização de uma classe: regra geral

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

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

    Ponto p1, p2, p3, p4;
    int c1, c2, c3, c4;

    p1 = new Ponto(1,1);
    p2 = new Ponto();
    p3 = new Ponto(4,-5);
    p4 = new Ponto(10,15);

    //imprimir a representação interna dos pontos
    System.out.println("P1 = " + p1.toString());
    System.out.println("P2 = " + p2.toString());
    System.out.println("P3 = " + p3.toString());
    System.out.println("P4 = " + p4.toString());

    //aceder às coordenadas dos pontos através dos métodos de acesso
    c1 = p2.getX(); c2 = p2.getY();
    System.out.println("c1 = " + c1 + " c2 = " + c2);

    //alterar os pontos
    p2.deslocamento(3,5);
    p1.somaPonto(p3);
    p3.movePonto(5,5);
    p4.setY(-1);
    System.out.println("P1 = " + p1.toString());
    System.out.println("P2 = " + p2.toString());
    System.out.println("P3 = " + p3.toString());
    System.out.println("P4 = " + p4.toString());
    System.out.println("p1 == p2? : " + p1.iguais(p4));
```