

# Programação Orientada aos Objectos

LEI/LCC - 2º ano 2014/15

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”!

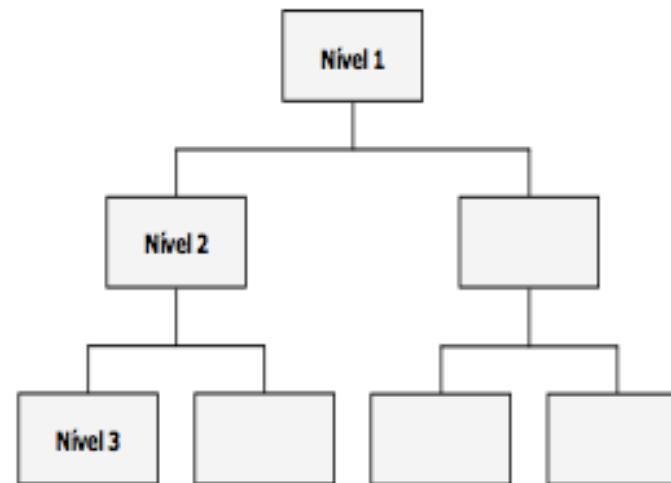
# POO na Engenharia de Software

- 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

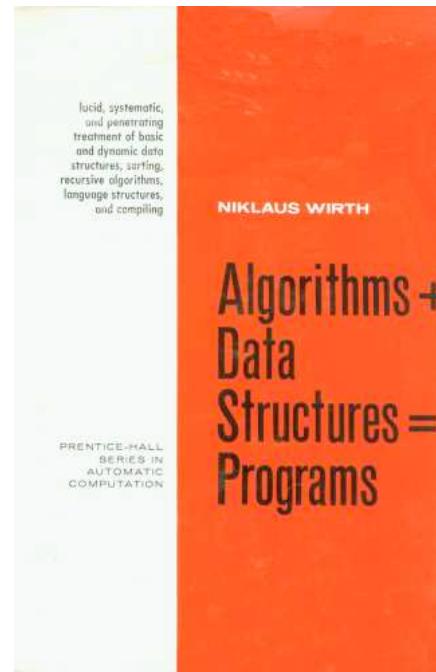
- a abordagem preconizada era do tipo “top-down”
  - estratégia para lidar com a complexidade
  - a princípio tudo é pouco definido e por refinamento vai-se encontrando mais detalhe
- neste modelo estruturado funcional e top-down:
  - as acções representam as entidades computacionais de 1ª classe
  -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



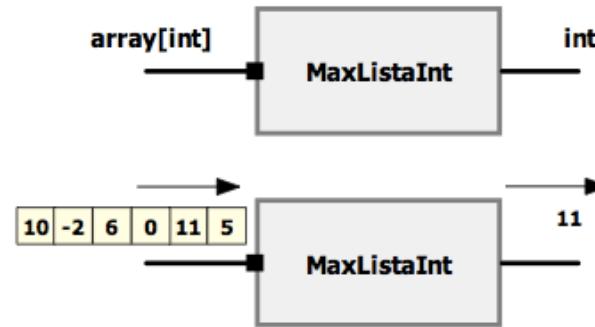
- esta abordagem não apresentava grandes riscos em projectos de pequena dimensão
- contudo em projectos de dimensão superior começou a não ser possível ignorar as vantagens da reutilização que não eram evidentes na abordagem estruturada
- É importante reter a noção de **reutilização** de software, como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

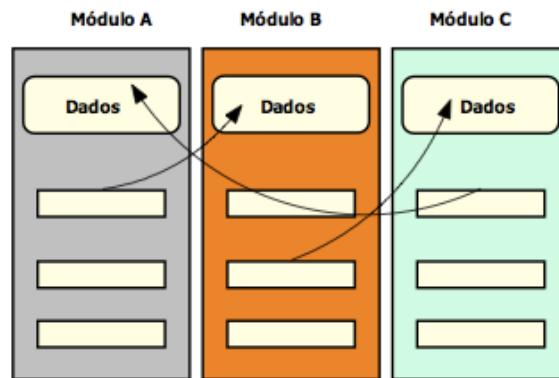
- como forma de aumentar o grão da reutilização várias linguagens criaram a noção de **módulos**
- os módulos possuem declarações de dados e declarações de funções e procedimentos invocáveis do exterior
- possuem a (grande) vantagem de poderem ser compilados de forma autónoma
  - podem assim ser associados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

- módulo como abstracção procedural:



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

- módulos interdependentes:

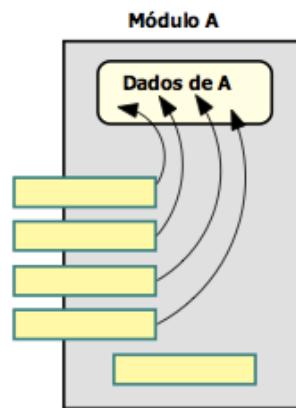


- a partilha de dados quebra as vantagens de uma possível reutilização
- num cenário mais real os diversos módulos interdependentes teriam de ser todos compilados e importados para os programas cliente

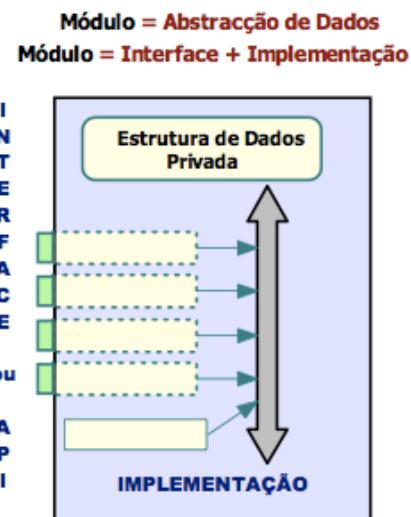
# Tipos Abstractos de Dados

- 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

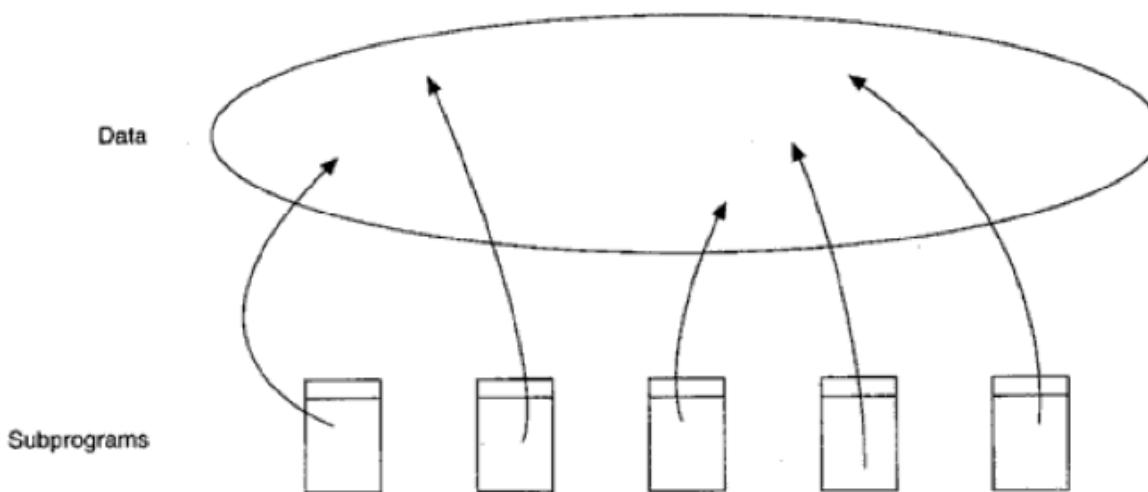
- A estrutura de dados local passa a estar completamente escondida: **Data Hiding**
- Os procedimentos e funções são serviços (API) que possibilitam que do exterior se possa obter informação acerca dos dados
- Módulos passam assim a ser vistos como mecanismos de **abstracção de dados**



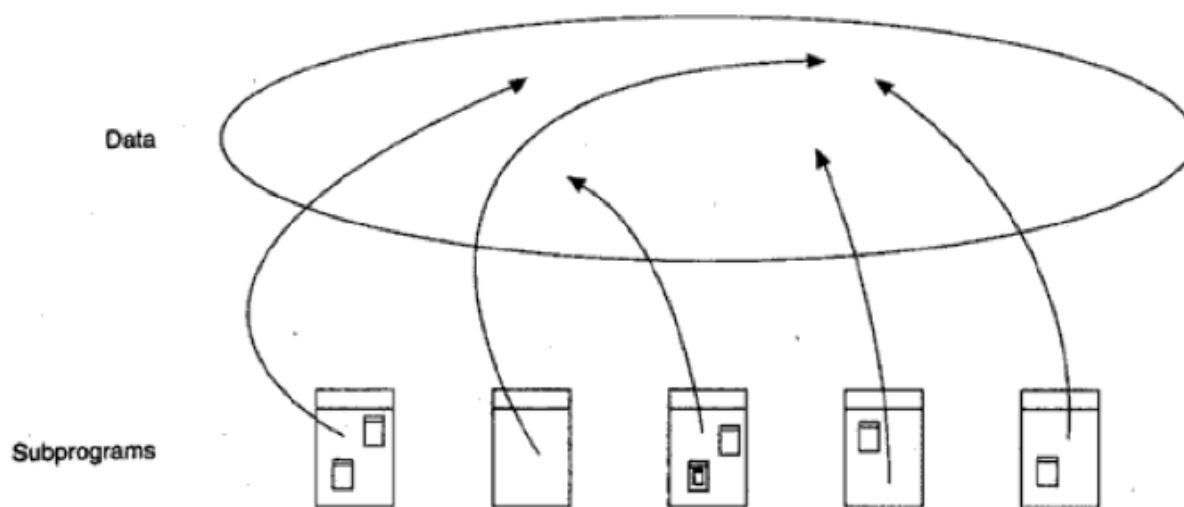
- se os módulos forem construídos com estas preocupações, então passamos a ter:
  - capacidade de reutilização
  - encapsulamento de dados



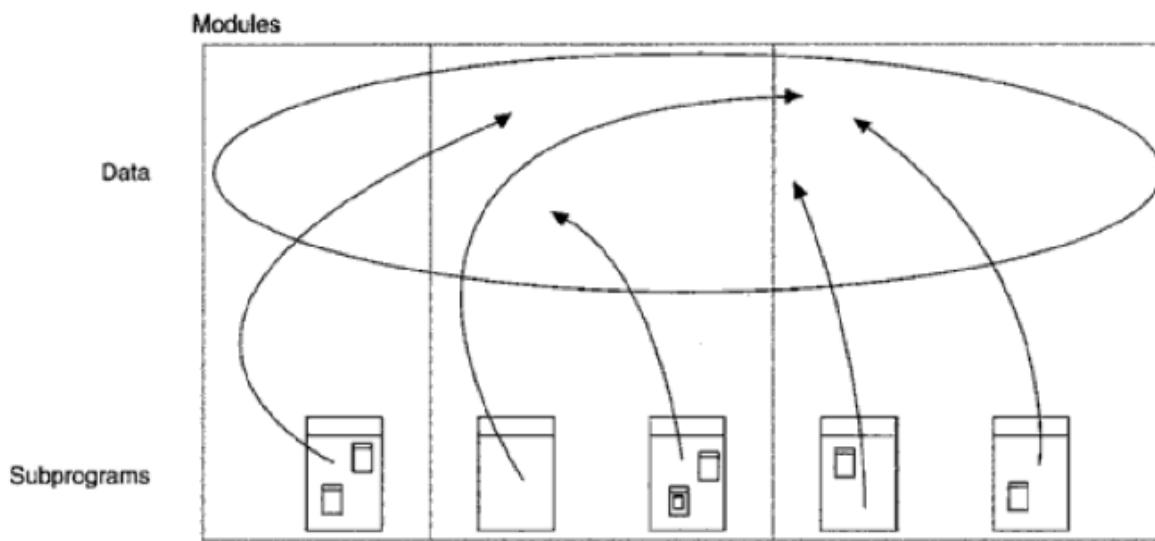
# Evolução das abordagens



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

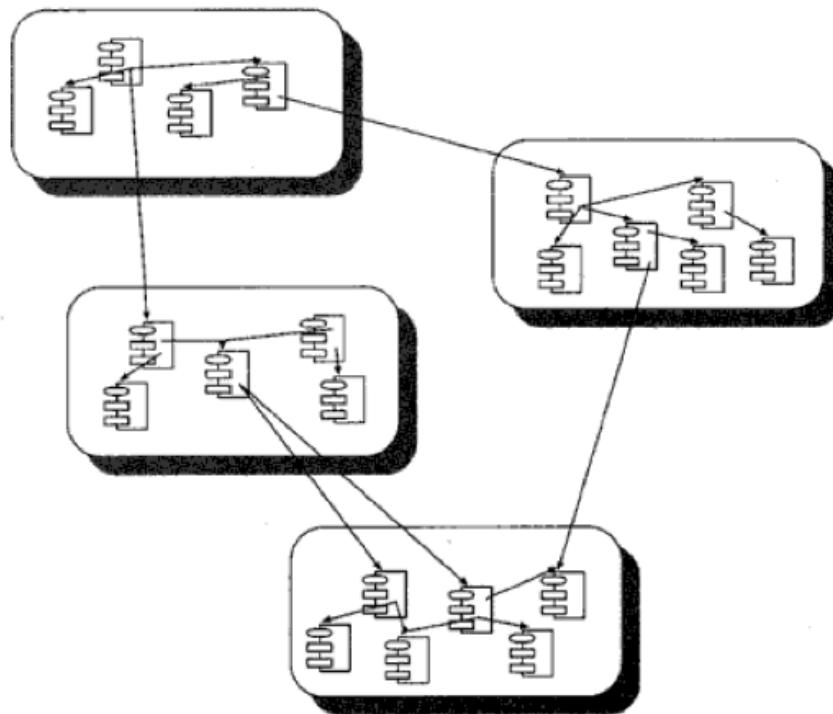


**Figure 2-2**  
**The Topology of Late Second- and Early Third-Generation Programming Languages**



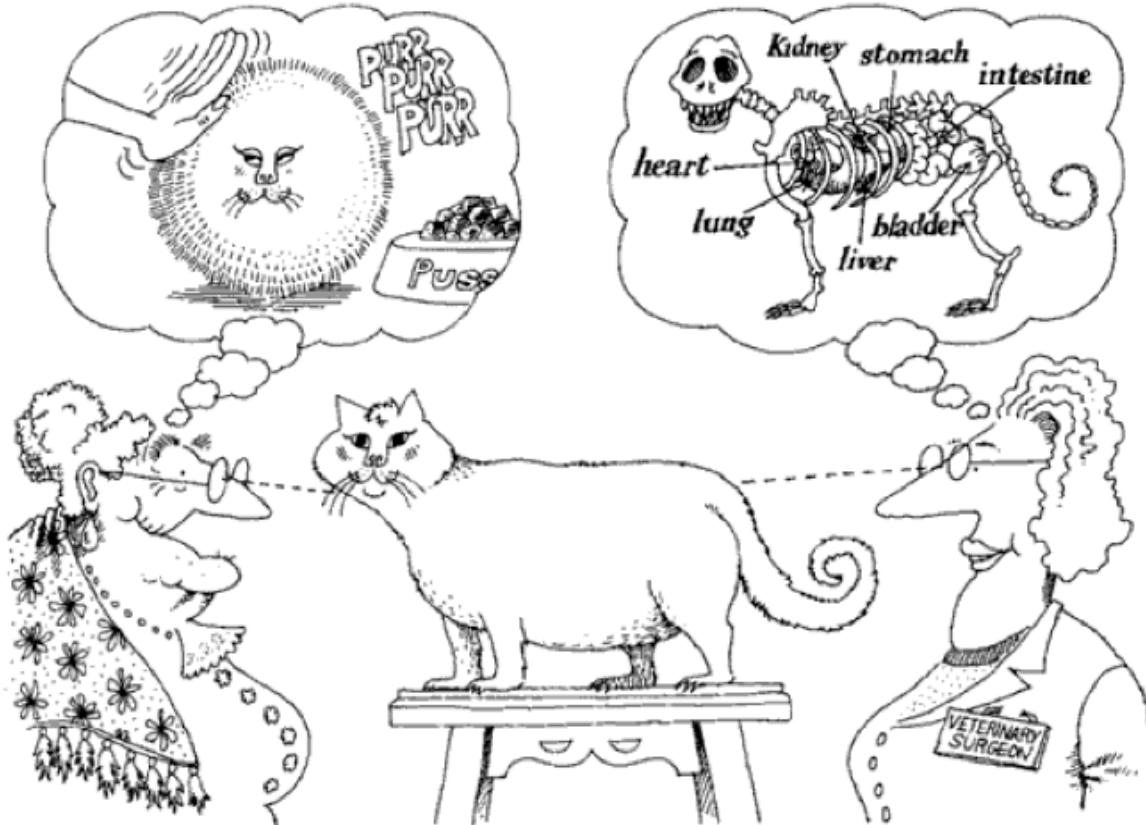
**Figure 2-3**  
The Topology of Late Third-Generation Programming Languages

# Onde queremos chegar



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

# Abstracção



# 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);

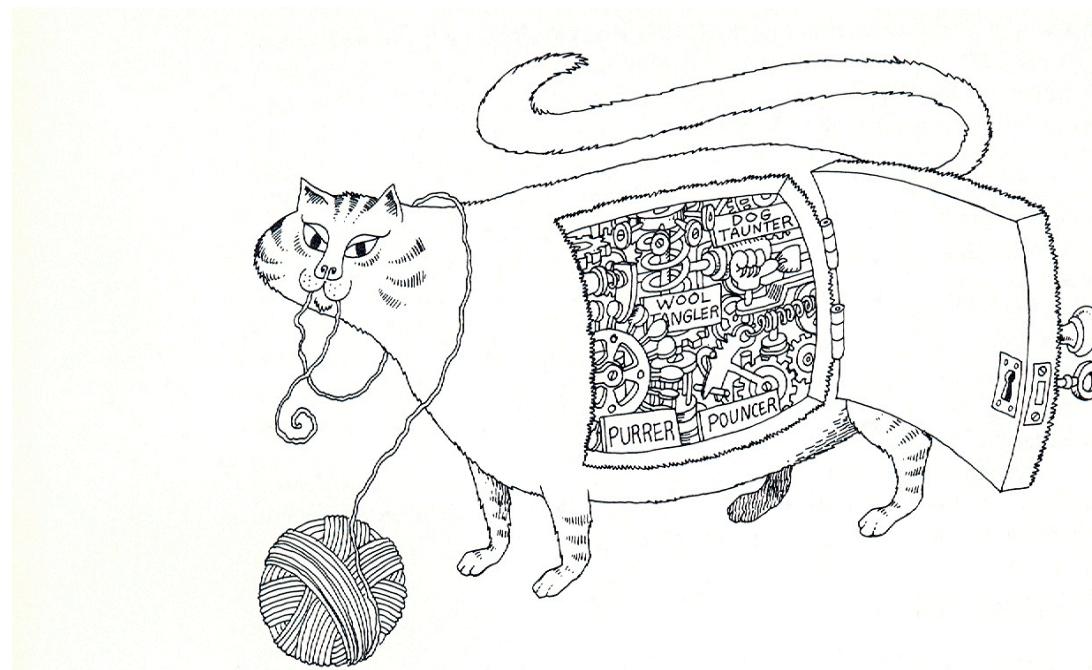
    complx2.real = complx1.real + complx2.real;
    complx2.img = complx1.img + complx2.img;

    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

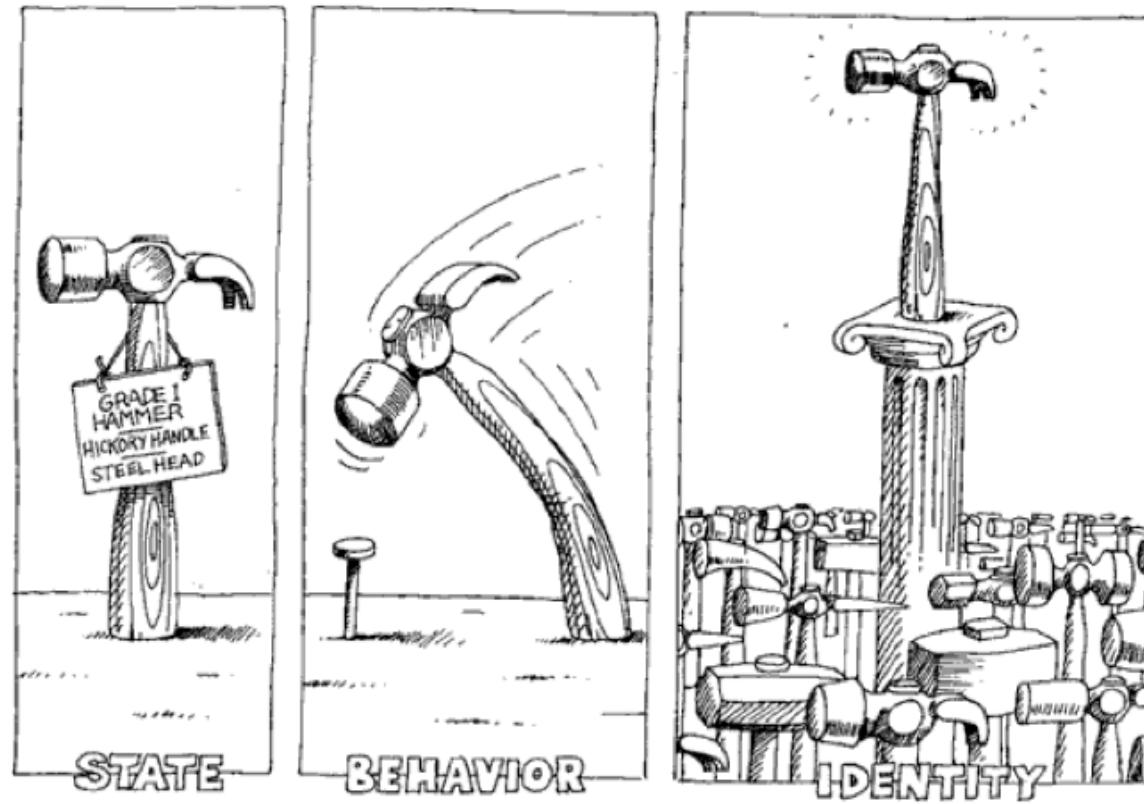
- Um objecto é a representação de uma entidade do mundo real, com:
  - atributos privados
  - operações
- **Objecto** = Dados Privados (variáveis de instância) + Operações (métodos)

# 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)

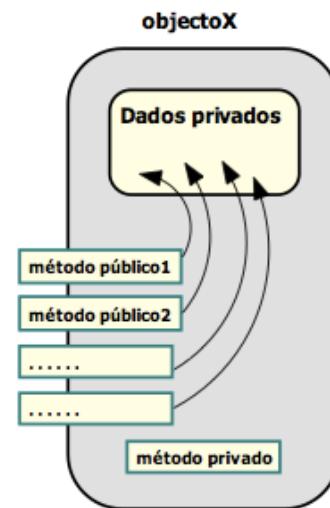
# Estado, Comportamento e Identidade



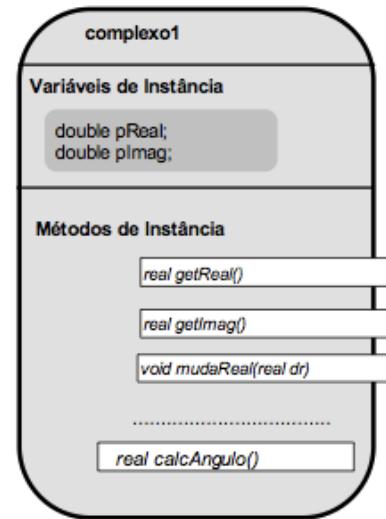
- Objecto = “black box”
  - apenas se conhecem os pontos de acesso (as operações)
  - desconhece-se a implementação interna
- Vamos chamar
  - aos dados: **variáveis de instância**
  - às operações: **métodos de instância**

# Encapsulamento

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



- Um objecto que representa um número complexo:

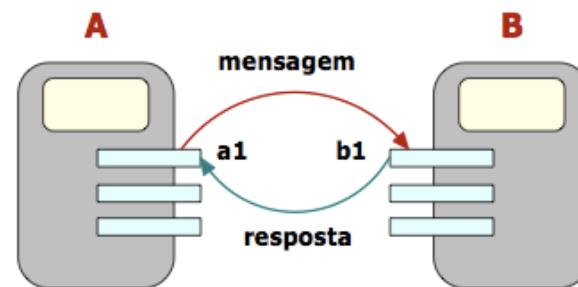


- O método `calcAngulo()` é interno e não pode ser invocado por entidades externas

- 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",9,6,2015)`, para inserir um novo evento na agenda
  - `ag.libertaEventosDia(25,4,2015)`, para remover todos os eventos de um dia
  - `String[] ev = ag.getEventos(6,2015)`, 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*

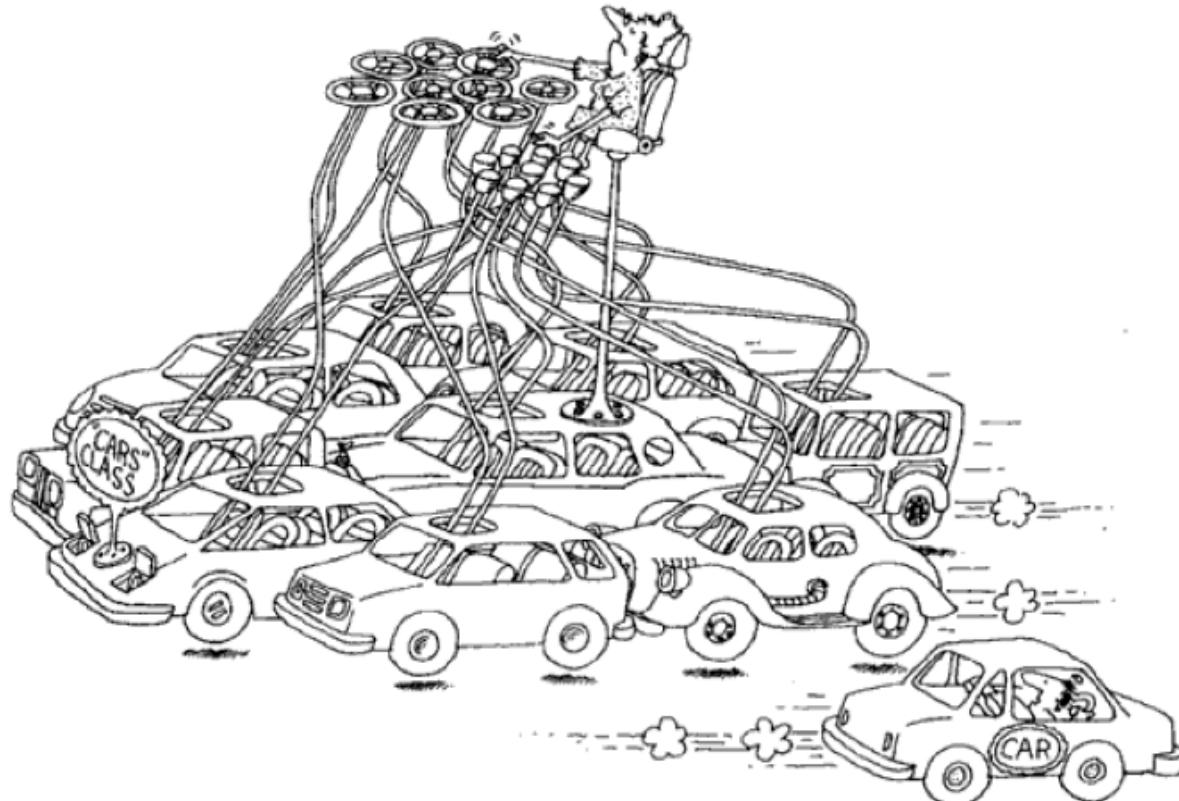
- uma classe é um *módulo* onde se especifica, quer a estrutura, quer o comportamento das instâncias, que a partir dela criamos
- uma vez que todos os objectos criados a partir de uma classe respondem à mesma interface, i.e. o mesmo conjunto de mensagens, são exteriormente utilizáveis de igual forma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



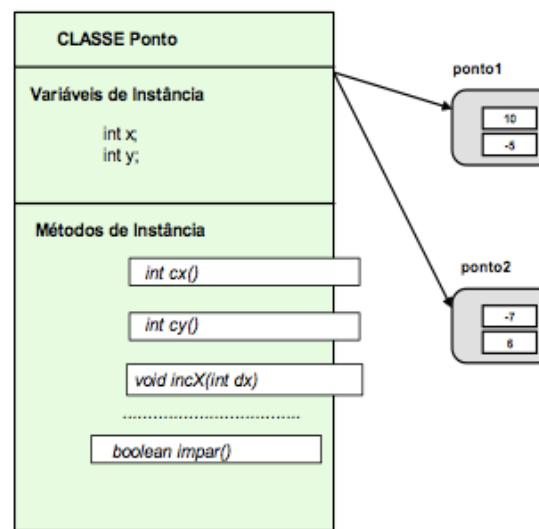
# ...e ainda sobre classes

- “*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:

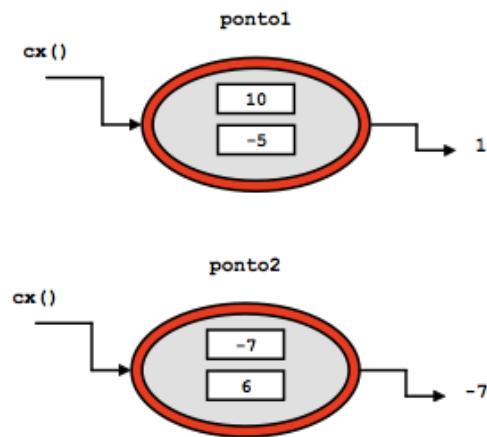


- *ponto1* e *ponto2* são instâncias diferentes, mas possuem o mesmo comportamento
- não faz sentido replicar o comportamento por todos os objectos

# 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;  
  
    // Construtor de instâncias
```

- 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 referenciar 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
<code>public</code>	Qualquer classe
<code>protected</code>	Própria classe, qualquer classe do mesmo <i>package</i> e qualquer subclasse
<code>private</code>	Própria classe
<code>nenhum</code>	Própria classe e classes dentro do mesmo <i>package</i>

- 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

```
/*
 * 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

## ● declaração das v.i.

---

```
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();
}
```

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

public int getCapacidade() {
    return this.capacidade;
}

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

/**
 * 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
  - `Istalunos[i] == a`, não é eficaz porque compara os apontadores
  - `(Istalunos[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.ocupaçao && !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  $x.equals(y) == \text{true}$ ,  $y.equals(z) == \text{true}$ , então  $x.equals(z) == \text{true}$
- 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.ocupaçao && !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 acúcar 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("Número:" + 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() {
    StringBuilder sb= new StringBuilder();

    sb.append("Número: ");
    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 noção 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 Círculo

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

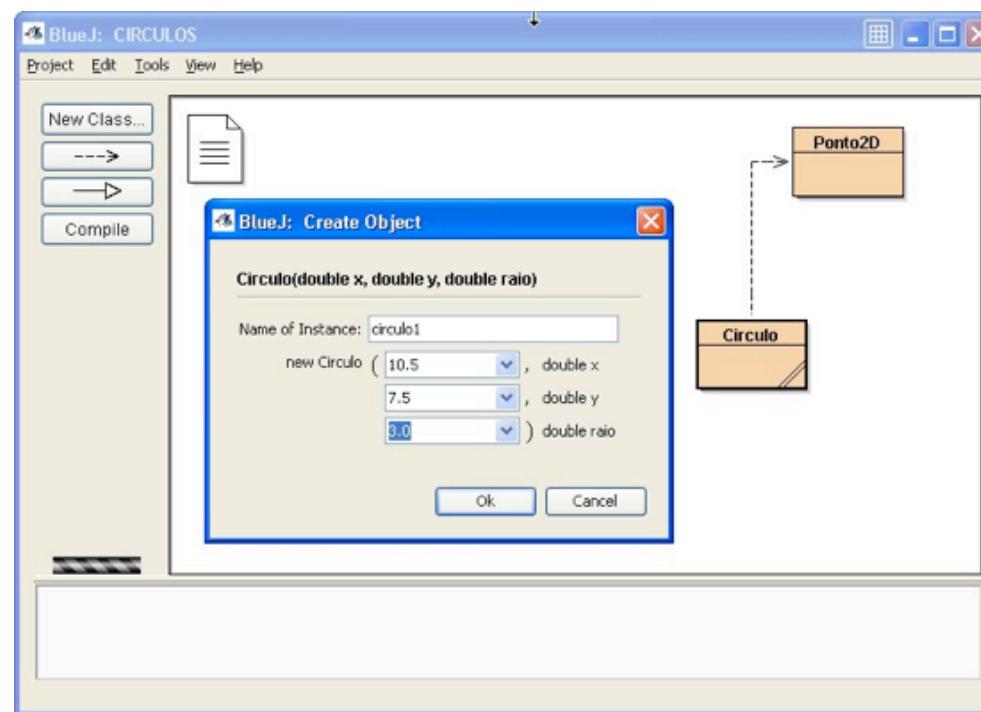
    private double raio;      // o raio do círculo
    private Ponto2D centro;   // ponto que define o centro do círculo

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

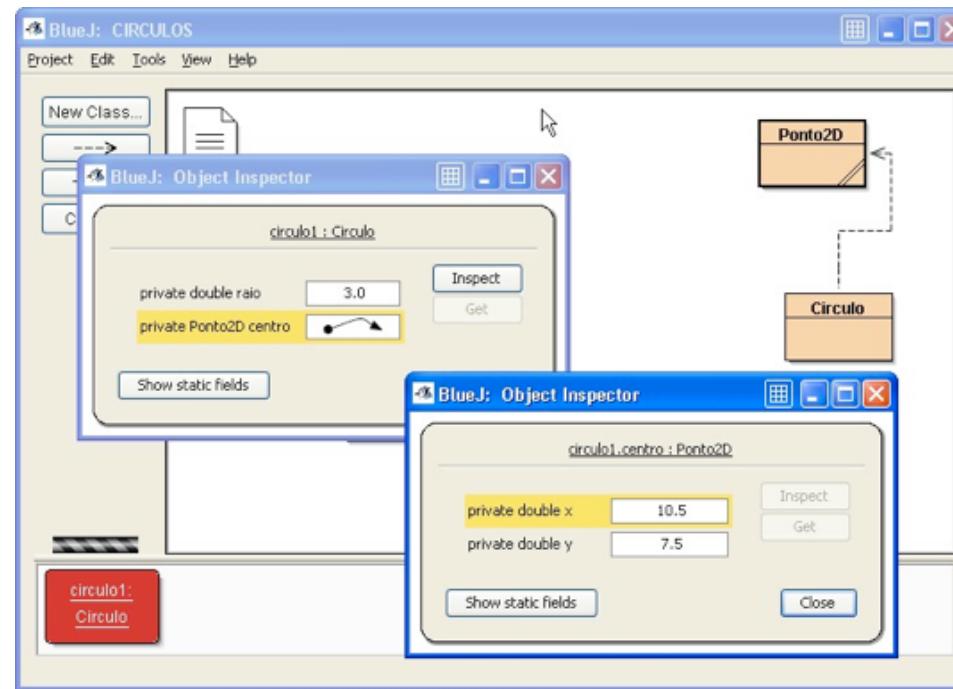
- e seja o método, daCentro() definido como:

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

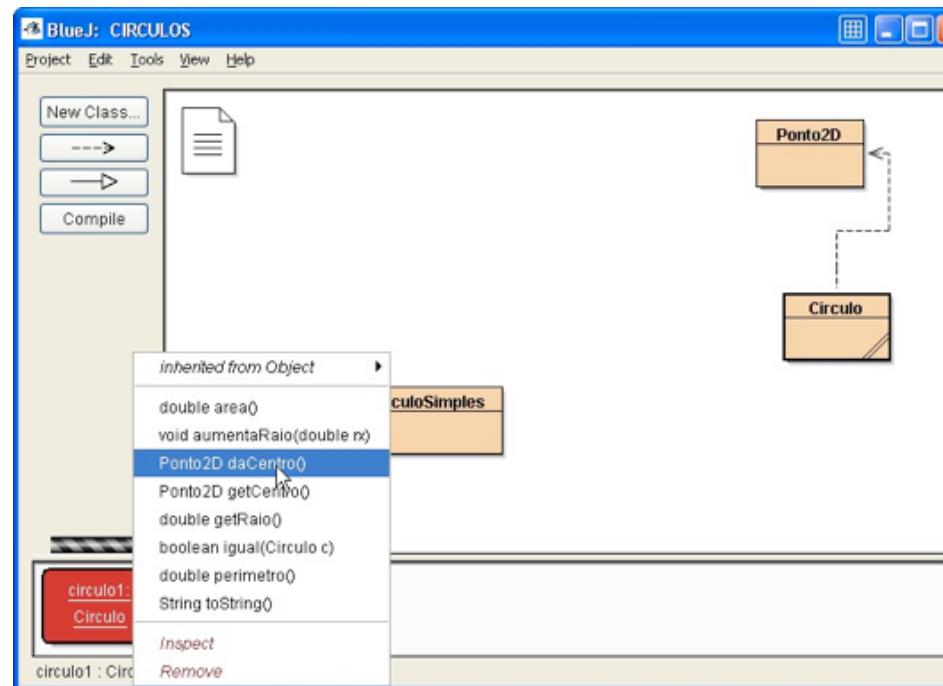
# Criação de um círculo



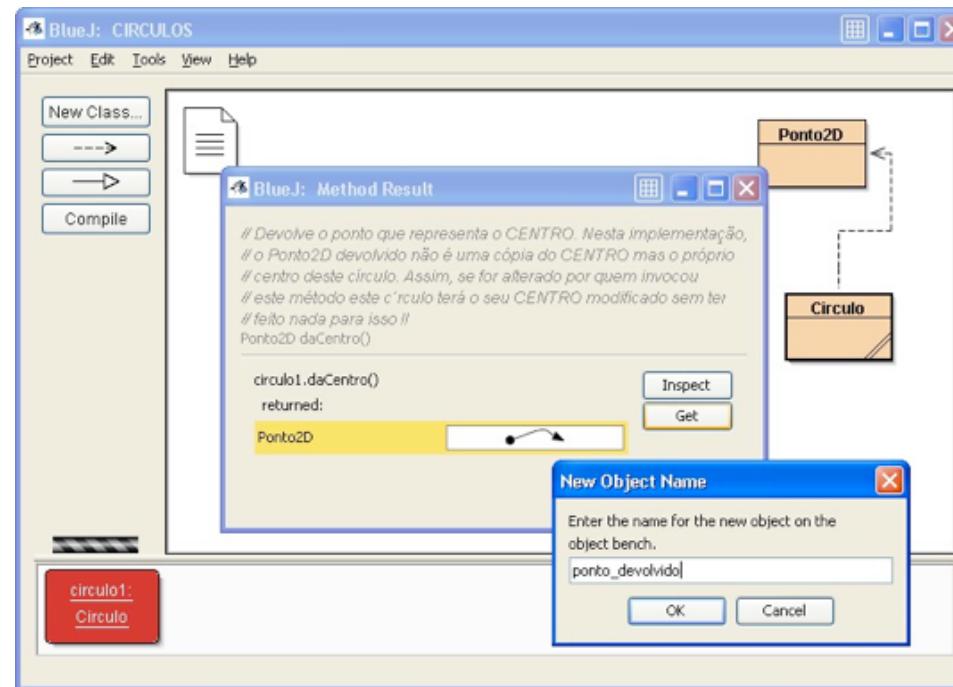
# Inspect do objeto



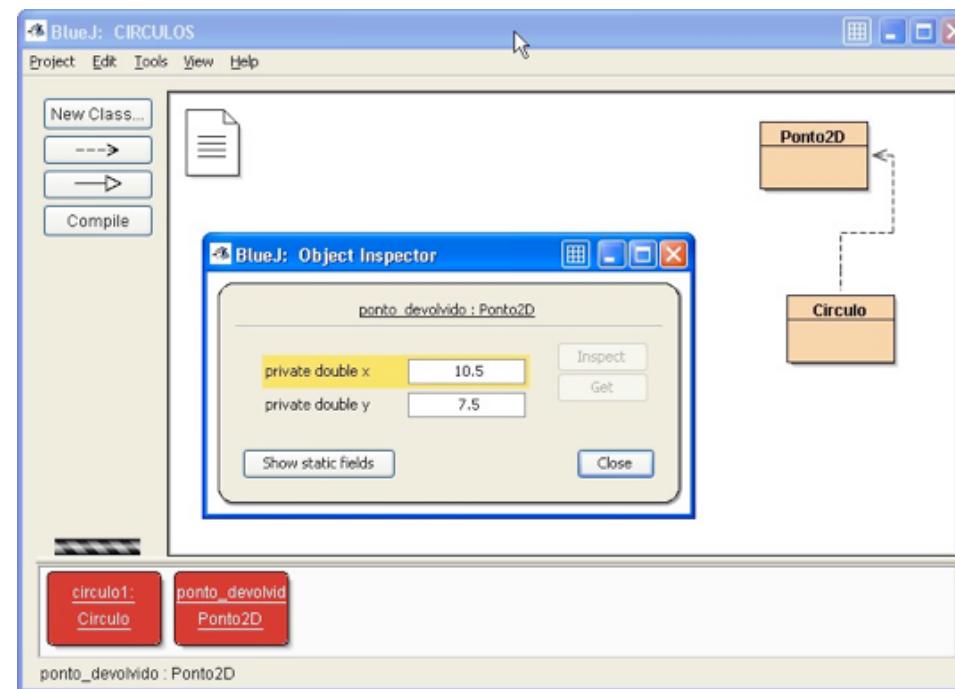
# Invocação de daCentro()



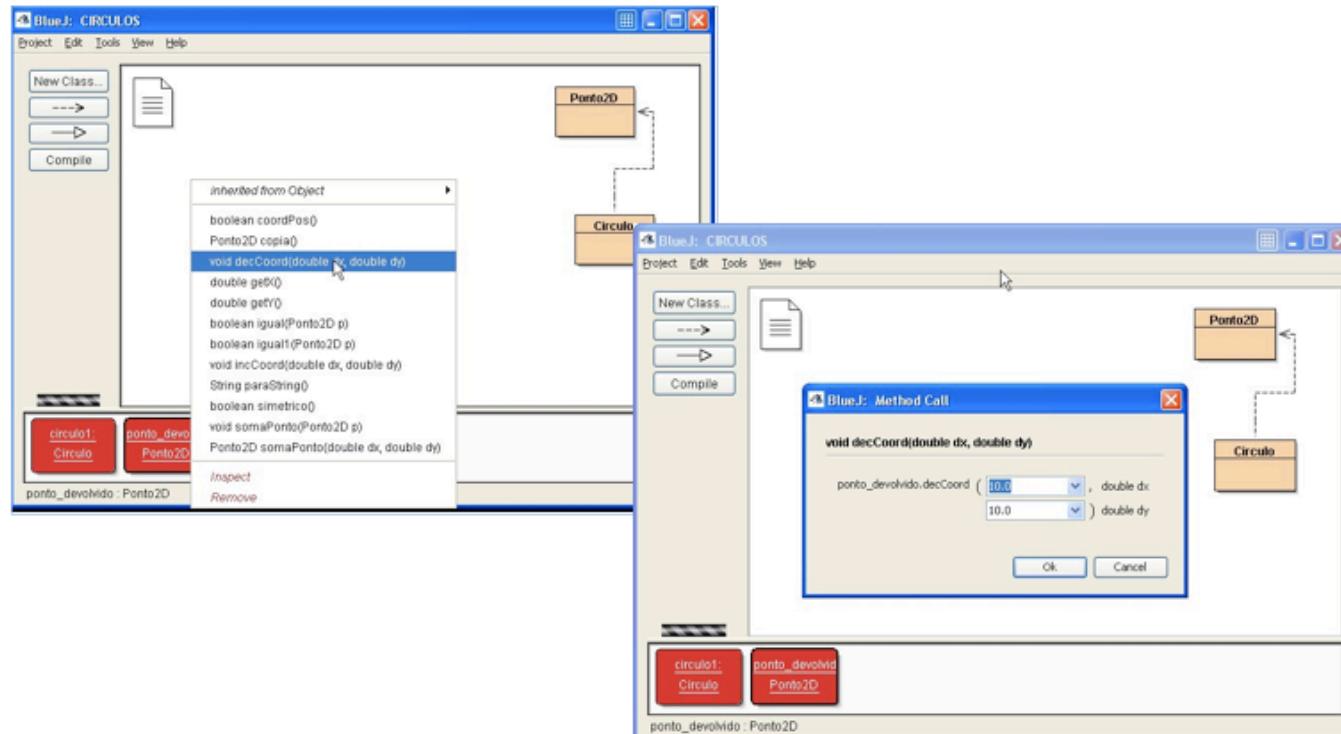
# Ficar com o objecto devolvido



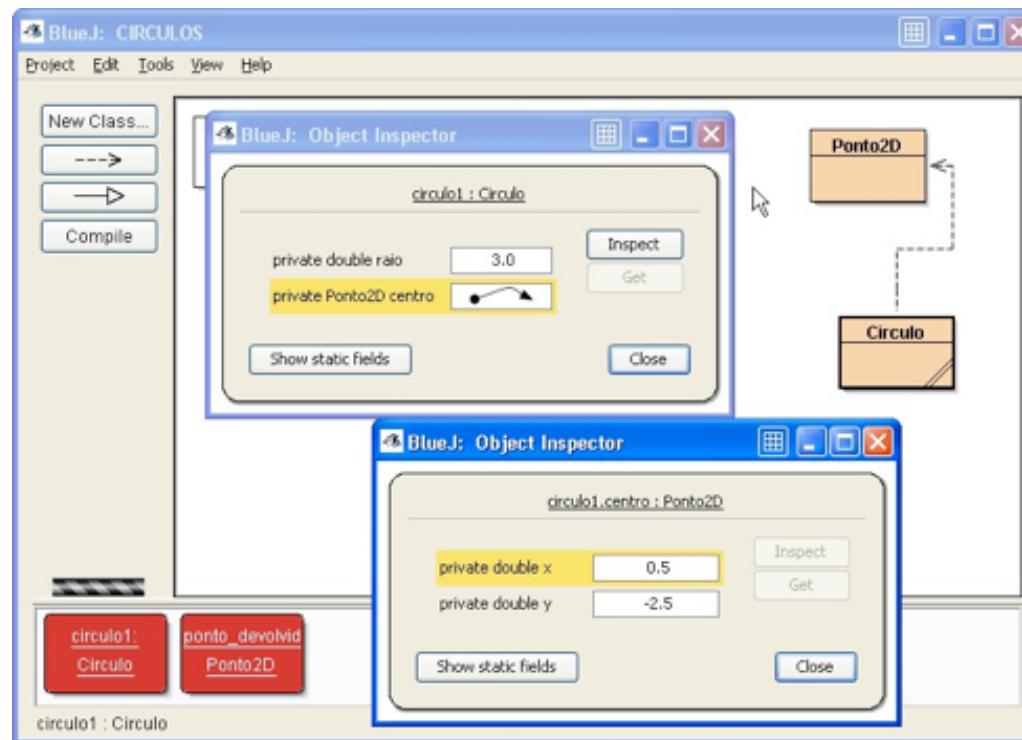
# Inspect do resultado



# 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:

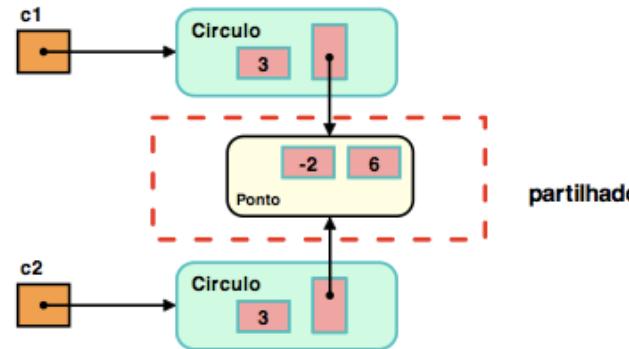
- implementar o clone como sendo uma invocação do construtor de cópia

```
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 instâncias de turma.
 */

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

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

## ● `toString`

```
/*
 * Método toString por questões de compatibilização com as restantes
 * classes do Java.
 *
 * Como o toString é estrutural e a classe Aluno tem esse método
 * implementado o resultado é o esperado.
 */
public String toString() {
    StringBuilder sb = new StringBuilder();

    sb.append("Designação: "); sb.append(this.designacao+"\n");
    sb.append("Capacidade: "); sb.append(this.capacidade+"\n");
    sb.append("Alunos: "+"\n"); sb.append(this.lstalunos.toString());

    return sb.toString();
}
```

## ● `clone`

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

# Ainda sobre classes

- “*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.*
- a definição de classe que temos vindo a utilizar está incompleta
  - quer as instâncias, quer as classes são **objectos**

- as classes não deixam de ser objectos
  - objectos que guardam o que é comum a todas as instâncias
  - **apenas um** objecto-classe por classe
- se objectos possuem estado e comportamento, então podemos extrapolar e dizer que a classe também tem:
  - variáveis e métodos de classe

- os métodos de classe são activados a partir de mensagens que são enviados para o objecto classe.
  - exemplo: Ponto2D.metodo()
- se uma classe possui variáveis de classe o acesso a essas variáveis deverá ser feito através dos métodos de classe
  - métodos de instância => v. instância
  - métodos de classe => v. classe

- o que é que se pode guardar como variável de classe?
- valores que sejam comuns a todas os objectos instância
- não faz sentido colocar estes valores em todos os objectos (repetição)

- Imagine-se que na classe Conta Bancária se pretende:
  - a) saber quantas contas foram criadas
  - b) saber qual é o somatório dos saldos das contas existentes
- se para b) podemos ter outras soluções (quais?), como é que poderemos satisfazer o requisito expresso em a)?

- uma variável que guarde o número de contas não é certamente uma variável de instância
  - actualização do contador em todas as instâncias?
  - redundância?
- teriam de se ter implementados mecanismos de comunicação entre todas as instâncias!!

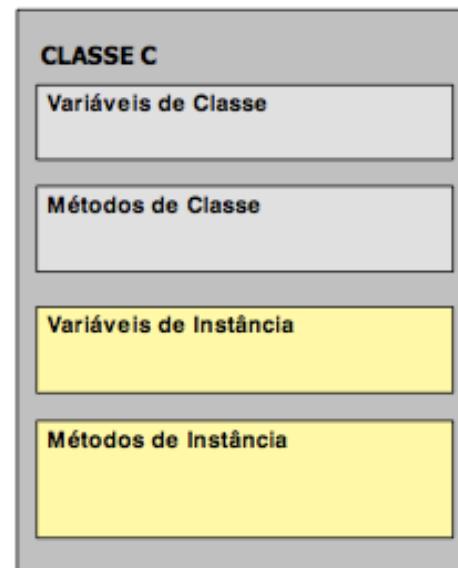
- as variáveis de classe servem para guardar *informação global* a todas as instâncias
- podem também ser utilizadas para guardar constantes que são utilizadas pelos diversos objectos instância
  - exemplo: Math.PI

- os métodos de classe fazem o acesso às variáveis de classe
- aos métodos de classe aplicam-se as mesmas regras de visibilidade que se aplicam aos métodos de instância
- os métodos de classe são sempre acessíveis às instâncias, mas métodos de classe **não** tem acesso aos métodos de instância

- como se declaram métodos e variáveis de classe?
  - utilizando o prefixo **static**
- a definição de classe passa a ter:
  - declaração de variáveis de classe
  - declaração de métodos de classe
  - declaração de variáveis de instância
  - declaração de métodos de instância

# Estrutura de uma classe

- estrutura tipo de uma classe



# Classe PMMB

- seja uma classe PMMB (porta moedas multibanco)
- queremos acrescentar informação sobre o número total de PMMB's criados e sobre o valor total de saldo existente em todos os cartões

- duas variáveis de classe e respectivos métodos de acesso

```
public class PMMB {  
    // Variáveis de Classe  
    public static int numPMMB = 0;  
    public static double saldoTotal = 0.0;  
  
    // Métodos de Classe  
    public static int getNumPMMB() {  
        return numPMMB;  
    }  
    public static double getSaldoTotal() {  
        return saldoTotal;  
    }  
    public static void incNumPMMB() {  
        numPMMB++;  
    }  
    public static void actSaldoTotal(double valor) {  
        saldoTotal += valor;  
    }  
}
```

```
// Variáveis de Instância  
private String codigo;  
private String titular;  
private double saldo;  
private int numMovs; // total de movimentos
```

- onde é que se actualiza a informação do número de cartões existentes?

- no construtor de PMMB

```
public PMMB() {  
    codigo = ""; titular = "";  
    saldo = 0.0; numMovs = 0;  
    this.actSaldoTotal(0);  
    this.incNumPMMB();  
}
```

- no construtor, *this* representa a classe, sempre que o método invocado seja de classe.

```
// Métodos de Instância
. . .
public void carregaPM(double valor) {
    saldo = saldo + valor;
    numMovs++; actSaldoTotal(valor);
}
// pré-condição
public boolean prePaga(double valor) {
    return saldo >= valor ;
}

public void pagamento(double valor) {
    saldo = saldo - valor;
    numMovs++; actSaldoTotal(-valor);
}
```

- para não confundir quem lê, as invocações anteriores poderiam ter sido feitas na forma:
  - PMMB.actSaldoValorTotal(valor)

- os métodos de classe anteriormente definidos devem ser utilizados como:

```
int pMoedas = PMMB.getNumPMMB();
double saldos = PMMB.getSaldoTotal();

PMMB.incNumPMMB();      //
PMMB.actSaldoTotal(valor);
```

- para que se consiga perceber o código é necessário que se siga a convenção que diz que as classes começam por letra maiúscula

# Estruturas de Dados

- Como já vimos atrás, podemos criar conceitos mais complexos através do mecanismo de composição/agregação de objectos de classes mais simples:
  - a Turma a partir de Aluno
  - o Stand a partir de Veículo
  - etc.
- Mas para isso precisamos de ter colecções de objectos...

- Até ao momento apenas temos disponível a utilização de arrays

```
Aluno[] alunos = new Aluno[30];
Veiculo[] carros = new Veiculo[10];

for (int i=0; i<alunos.length && !encontrado; i++)
    if (alunos[i].getNota() == 20)
        encontrado = true;

for(Veiculo v: carros)
    System.out.println(v.toString());
```

- Esta é uma solução simples e testada, com a inconveniente de o tamanho da estrutura de dados ser estaticamente definido.
- Será que conseguimos ter uma estrutura de dados, baseada em arrays, que pudesse crescer de forma transparente para o utilizador?
  - Sim, bastando para tal criarmos uma classe com esse comportamento

- Seja essa classe chamada de `GrowingArray` e, por comodidade, vamos utilizar instâncias de `Circulo`
- Que operações necessitamos:
  - adicionar um circulo (no fim e numa posição)
  - remover um círculo
  - ver se um circulo existe
  - dar a posição de um circulo na estrutura
  - dar número de elementos existentes

- Documentação com os métodos necessários:

### Constructor Summary

[GrowingArray\(\)](#)

[GrowingArray\(int capacidade\)](#)

### Method Summary

void	<a href="#">add(Circulo c)</a> Adiciona o elemento passado como parâmetro ao fim do array
void	<a href="#">add(int indice, Circulo c)</a> Método que adiciona um elemento numa determinada posição, forçando a que os elementos à direita no array façam shift.
boolean	<a href="#">contains(Circulo c)</a> Método que determina se um elemento está no array.
Circulo	<a href="#">get(int indice)</a> Devolve o elemento que está na posição indicada.
int	<a href="#">indexOf(Circulo c)</a> Método que determina o índice do array onde está localizada a primeira ocorrência de um objecto.
boolean	<a href="#">isEmpty()</a> Método que determina se o array contém elementos, ou se está vazio.
boolean	<a href="#">remove(Circulo c)</a> Remove a primeira ocorrência do elemento que é passado como parâmetro.
Circulo	<a href="#">remove(int indice)</a> Remove do array o elemento que está na posição indicada no parâmetro.
void	<a href="#">set(int indice, Circulo c)</a> Método que actualiza o valor de uma determinada posição do array.
int	<a href="#">size()</a> Método que determina o tamanho do array de elementos.

- Declarações iniciais:

```
public class GrowingArray {  
  
    private Circulo[] elementos;  
    private int size;  
  
    /**  
     * variável que determina o tamanho inicial do array,  
     * se for utilizado o construtor vazio.  
     */  
    private static final int capacidade_inicial = 20;  
  
    public GrowingArray(int capacidade) {  
        this.elementos = new Circulo[capacidade];  
        this.size = 0;  
    }  
  
    public GrowingArray() {  
        this(capacidade_inicial);  
    }  
}
```

- get de um elemento da estrutura de dados

```
/**  
 * Devolve o elemento que está na posição indicada.  
 *  
 * @param indice posição do elemento a devolver  
 * @return o objecto que está na posição indicada no parâmetro  
 * (deveremos ter atenção às situações em que a posição não existe)  
 */  
  
public Circulo get(int indice) {  
    if (indice <= this.size)  
        return this.elementos[indice];  
    else  
        return null; // ATENÇÃO!  
}
```

- set de uma posição da estrutura

```
/**  
 * Método que actualiza o valor de uma determinada posição do array.  
 *  
 * @param indice a posição que se pretende actualizar  
 * @param c o circulo que se pretende colocar na estrutura de dados  
 *  
 */  
public void set(int indice, Circulo c) {  
    if (indice <= this.size) //não se permitem "espaços vazios"  
        this.elementos[indice] = c;  
}
```

- adicionar um elemento à estrutura de dados

```
/**  
 * Adiciona o elemento passado como parâmetro ao fim do array  
 *  
 * @param c circulo que é adicionado ao array  
 *  
 */  
  
public void add(Circulo c) {  
    aumentaCapacidade(this.size + 1);  
    this.elementos[this.size++] = c;  
}
```

- método auxiliar que aumenta espaço

```
/**  
 * Método auxiliar que verifica se o array alocado tem capacidade  
 * para guardar mais elementos.  
 * Por cada nova inserção, verifica se estamos a mais de metade  
 * do espaço  
 * alocado e, caso se verifique, aloca mais 1.5 de capacidade.  
 */  
  
private void aumentaCapacidade(int capacidade) {  
    if (capacidade > 0.5 * this.elementos.length) {  
        int nova_capacidade = (int)(this.elementos.length * 1.5);  
        this.elementos = Arrays.copyOf(this.elementos, nova_capacidade);  
    }  
}
```

```
/**  
 * Método que adiciona um elemento numa determinada posição,  
 * forçando a  
 * que os elementos à direita no array façam shift.  
 * Tal como no método de set não são permitidos espaços.  
 *  
 * @param indice índice onde se insere o elemento  
 * @param c circulo que será inserido no array  
 *  
 */  
  
public void add(int indice, Circulo c) {  
    if (indice <= this.size) {  
        aumentaCapacidade(this.size+1);  
        System.arraycopy(this.elementos, indice, this.elementos,  
                         indice + 1, this.size - indice);  
        this.elementos[indice] = c;  
        this.size++;  
    }  
}
```

```
/**  
 * Remove do array o elemento que está na posição indicada no parâmetro.  
 * Todos os elementos à direita do índice sofrem um deslocamento  
 * para a esquerda.  
 * @param indice índice do elemento a ser removido  
 * @return o elemento que é removido do array. No caso do índice não  
 * existir devolver-se-á null.  
 */  
public Circulo remove(int indice) {  
    if (indice <= this.size) {  
        Circulo c = this.elementos[indice];  
  
        int deslocamento = this.size - indice - 1;  
        if (deslocamento > 0)  
            System.arraycopy(this.elementos, indice+1, this.elementos,  
                            indice, deslocamento);  
        this.elementos[--this.size] = null;  
        return c;  
    }  
    else  
        return null;  
}
```

```
* Remove a primeira ocorrência do elemento que é passado como parâmetro.  
* Devolve true caso o array contenha o elemento, falso caso contrário.  
*  
* @param c círculo a ser removido do array (caso exista)  
* @return true, caso o círculo exista no array  
*/  
public boolean remove(Circulo c) {  
    if (c != null) {  
        boolean encontrado = false;  
        for (int indice = 0; indice < this.size && !encontrado; indice++)  
            if (c.equals(this.elementos[indice])) {  
                encontrado = true;  
                int deslocamento = this.size - indice - 1;  
                if (deslocamento > 0)  
                    System.arraycopy(this.elementos, indice+1,  
                                     this.elementos, indice, deslocamento);  
                this.elementos[--this.size] = null;  
            }  
        return encontrado;  
    }  
    else  
        return false;  
}
```

```
/**  
 * Método que determina o índice do array onde está localizada a  
 * primeira ocorrência de um objecto.  
 *  
 * @param c círculo de que se pretende determinar a posição  
 * @return a posição onde o círculo se encontra. -1 caso não esteja no  
 * array ou o círculo passado como parâmetro seja null.  
 */  
  
public int indexOf(Circulo c) {  
    int posicao = -1;  
    if (c != null) {  
        boolean encontrado = false;  
        for (int i = 0; i < this.size && !encontrado; i++)  
            if (c.equals(this.elementos[i])) {  
                encontrado = true;  
                posicao = i;  
            }  
    }  
    return posicao;  
}
```

```
/**  
 * Método que determina se um elemento está no array.  
 *  
 * @param c círculo a determinar se está no array  
 * @return true se o objecto estiver inserido na estrutura de dados,  
 * false caso contrário.  
 */  
public boolean contains(Circulo c) {  
    return index0f(c) >= 0;  
}
```

```
/**  
 * Método que determina se o array contém elementos, ou se está vazio.  
 *  
 * @return true se o array estiver vazio, false caso contrário.  
 */  
  
public boolean isEmpty() {  
    return this.size == 0;  
}
```

```
public class TesteGA {  
    public static void main(String[] args) {  
  
        Circulo c1 = new Circulo(2,4,4.5);  
        Circulo c2 = new Circulo(1,4,1.5);  
        Circulo c3 = new Circulo(2,7,2.0);  
        Circulo c4 = new Circulo(3,3,2.0);  
        Circulo c5 = new Circulo(2,6,7.5);  
  
        GrowingArray ga = new GrowingArray(10);  
        ga.add(c1.clone());  
        ga.add(c2.clone());  
        ga.add(c3.clone());  
  
        System.out.println("Num elementos = " + ga.size());  
        System.out.println("Posição do c2 = " + ga.indexOf(c2));  
    }  
}
```

# Colecções Java

- O Java oferece um conjunto de classes que implementam as estruturas de dados mais utilizadas
  - oferecem uma API consistente entre si
  - permitem que sejam utilizadas com qualquer tipo de objecto - são parametrizadas por tipo

- Poderemos representar:
  - `ArrayList<Aluno> alunos`
  - `HashSet<Aluno> alunos;`
  - `HashMap<String,Aluno> turmaAlunos;`
  - `TreeMap<String, Docente> docentes;`
  - `Stack<Pedido> pedidosTransferência;`

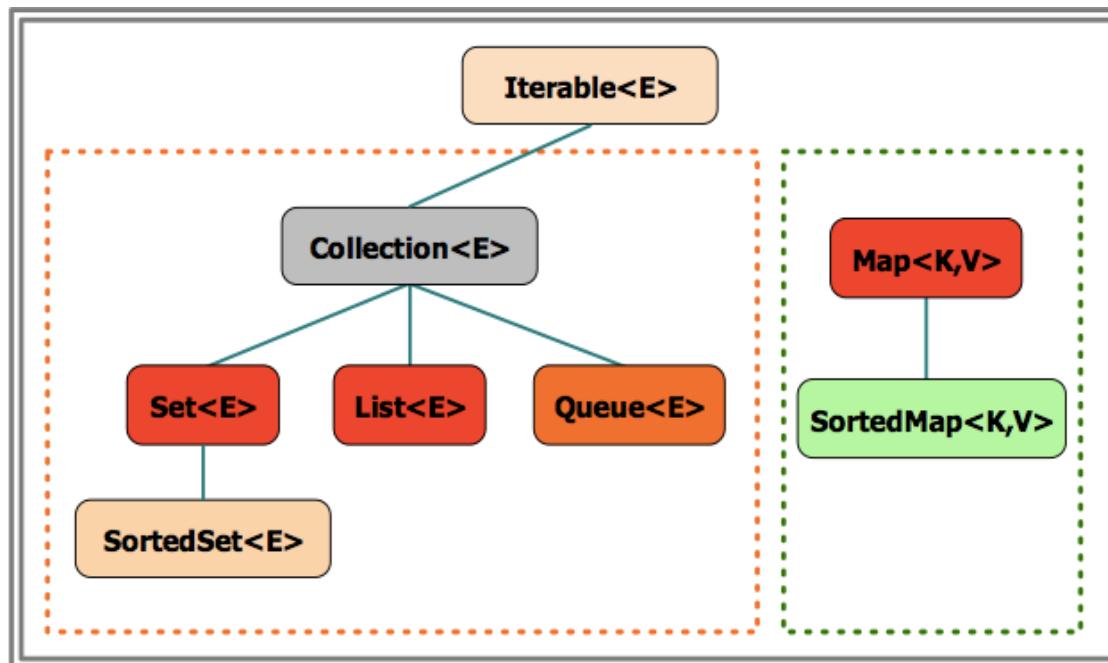
- Ao fazer-se `ArrayList<Circulo>` passa a ser o compilador a testar, e validar, que só são utilizados objectos do tipo `Circulo` no `arraylist`.
- isto dá uma segurança adicional aos programas, pois em tempo de execução não teremos erros de compatibilidade de tipos
- os tipos de dados são verificados em tempo de compilação

# JCF

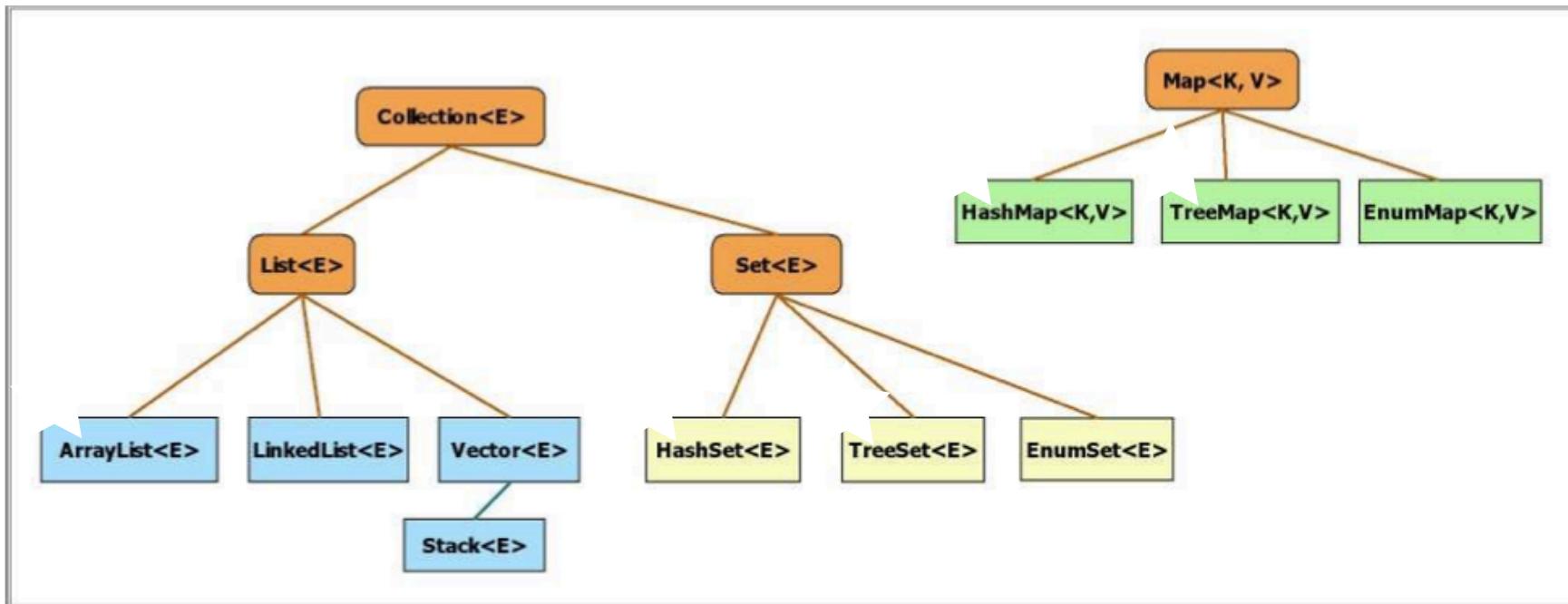
- O JCF (Java Collections Framework) agrupa as várias classes genéricas que correspondem às implementações de referência de:
  - Listas:API de List<E>
  - Conjuntos:API de Set<E>
  - Correspondências unívocas:API de Map<K,V>

# Estrutura da JCF

- Existe uma arrumação por API (interfaces)



- Para cada API (interface) existem diversas implementações (a escolher consoante critérios do programador)



# ArrayList<E>

- As classes da Java Collections Framework são exemplos muito interessantes de codificação
- Como o código destas classes está escrito em Java é possível ao programador observar como é que foram implementadas

# ArrayList<E>: v.i. e construtores

```
public class ArrayList<E> extends AbstractList<E>
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{
    private static final long serialVersionUID = 8683452581122892189L;

    /**
     * The array buffer into which the elements of the ArrayList are stored.
     * The capacity of the ArrayList is the length of this array buffer.
     */
    private transient Object[] elementData;

    /**
     * The size of the ArrayList (the number of elements it contains).
     *
     * @serial
     */
    private int size;

    /**
     * Constructs an empty list with the specified initial capacity.
     *
     * @param initialCapacity the initial capacity of the list
     * @throws IllegalArgumentException if the specified initial capacity
     *         is negative
     */
    public ArrayList(int initialCapacity) {
        ...
        this.elementData = new Object[initialCapacity];
    }

    /**
     * Constructs an empty list with an initial capacity of ten.
     */
    public ArrayList() {
        this(10);
    }
}
```

# ArrayList<E>.contains()

```
public boolean contains(Object o) {
    return indexOf(o) >= 0;
}

public int indexOf(Object o) {
    if (o == null) {
        for (int i = 0; i < size; i++)
            if (elementData[i]==null)
                return i;
    } else {
        for (int i = 0; i < size; i++)
            if (o.equals(elementData[i]))
                return i;
    }
    return -1;
}
```

# ArrayList<E>: inserir

```
public boolean add(E e) {
    ensureCapacityInternal(size + 1); // Increments modCount!!
    elementData[size++] = e;
    return true;
}

public void add(int index, E element) {
    rangeCheckForAdd(index);

    ensureCapacityInternal(size + 1); // Increments modCount!!
    System.arraycopy(elementData, index, elementData, index + 1,
                     size - index);
    elementData[index] = element;
    size++;
}
```

# ArrayList<E>: get e set

```
public E get(int index) {
    rangeCheck(index);
    return elementData(index);
}

public E set(int index, E element) {
    rangeCheck(index);

    E oldValue = elementData(index);
    elementData[index] = element;
    return oldValue;
}
```

# Colecções Java

- Tipos de colecções disponíveis:
  - listas (definição em List<E>)
  - conjuntos (definição em Set<E>)
  - queues (definição em Queue<E>)
- noção de família (muito evidente) nas APIs de cada um destes tipos de colecções.

# List<E>

- Utilizar sempre que precise de manter ordem
- O método add não testa se o objecto existe na coleção
- O método contains verifica sempre o resultado de equals
- Implementação utilizada: **ArrayList<E>**

# ArrayList<E>

Categoria de Métodos	API de ArrayList<E>
Construtores	<code>new ArrayList&lt;E&gt;()</code> <code>new ArrayList&lt;E&gt;(int dim)</code> <code>new ArrayList&lt;E&gt;(Collection)</code>
Inserção de elementos	<code>add(E o); add(int index, E o);</code> <code>addAll(Collection); addAll(int i, Collection);</code>
Remoção de elementos	<code>remove(Object o); remove(int index);</code> <code>removeAll(Collection); retainAll(Collection)</code>
Consulta e comparação de conteúdos	<code>E get(int index); int indexOf(Object o);</code> <code>int lastIndexOf(Object o);</code> <code>boolean contains(Object o); boolean isEmpty();</code> <code>boolean containsAll(Collection); int size();</code>
Criação de Iteradores	<code>Iterator&lt;E&gt; iterator();</code> <code>ListIterator&lt;E&gt; listIterator();</code> <code>ListIterator&lt;E&gt; listIterator(int index);</code>
Modificação	<code>set(int index, E elem); clear();</code>
Subgrupo	<code>List&lt;E&gt; sublist(int de, int ate);</code>
Conversão	<code>Object[] toArray();</code>
Outros	<code>boolean equals(Object o); boolean isEmpty();</code>

```
import java.util.ArrayList;
public class TesteArrayList {
    public static void main(String[] args) {

        Circulo c1 = new Circulo(2,4,4.5);
        Circulo c2 = new Circulo(1,4,1.5);
        Circulo c3 = new Circulo(2,7,2.0);
        Circulo c4 = new Circulo(3,3,2.0);
        Circulo c5 = new Circulo(2,6,7.5);

        ArrayList<Circulo> circs = new ArrayList<Circulo>();
        circs.add(c1.clone());
        circs.add(c2.clone());
        circs.add(c3.clone());

        System.out.println("Num elementos = " + circs.size());
        System.out.println("Posição do c2 = " + circs.indexOf(c2));

        for(Circulo c: circs)
            System.out.println(c.toString());
    }
}
```

# Set<E>

- Utilizar sempre que se quer garantir ausência de elementos repetidos
- O método add testa se o objecto existe
- O método contains utiliza a lógica do equals, mas não só...
- Duas implementações: **HashSet<E>** e **TreeSet<E>**

# Set<E>

Categoria de Métodos	API de Set<E>
Inserção de elementos	<code>add(E o); addAll(Collection); addAll(int i, Collection);</code>
Remoção de elementos	<code>remove(Object o); remove(int index); removeAll(Collection); retainAll(Collection)</code>
Consulta e comparação de conteúdos	<code>boolean contains(Object o); boolean isEmpty(); boolean containsAll(Collection); int size();</code>
Criação de Iteradores	<code>Iterator&lt;E&gt; iterator();</code>
Modificação	<code>clear();</code>
Subgrupo	<code>List&lt;E&gt; sublist(int de, int ate);</code>
Conversão	<code>Object[] toArray();</code>
Outros	<code>boolean equals(Object o); boolean isEmpty();</code>

# HashSet<E>

- O método add calcula o valor do hash de E para determinar a posição do objecto na estrutura de dados
- O método contains necessita de saber o hash do objecto para determinar a posição em que o encontra
- Logo, não chega ter o equals definido
  - é necessário ter o método **hashcode()**

# TreeSet<E>

- O método add determina a posição na árvore em que deve colocar o objecto
- É necessário fornecer um método de comparação dos objectos - **compare()** ou **compareTo()**
- sem este método de comparação não é possível utilizar o TreeSet, a não ser para tipos de dados simples (String, Integer, etc.)

# Map<K,V>

- Quando se pretende ter uma associação de um objecto chave a um objecto valor
- Na dimensão das chaves não existem elementos repetidos (é um conjunto!)
- Duas implementações disponíveis:  
HashMap<K,V> e TreeMap<K,V>
  - aplicam-se à dimensão das chaves as considerações anteriores sobre conjuntos

# Percorrer uma colección

- Podemos utilizar o `foreach` para percorrer a colección toda, mas...
  - podemos querer parar antes do fim
  - podemos não ter o acesso à posição do elemento na coleção (caso dos conjuntos)
  - logo, é necessário um mecanismo comum para percorrer coleções

# Iterator

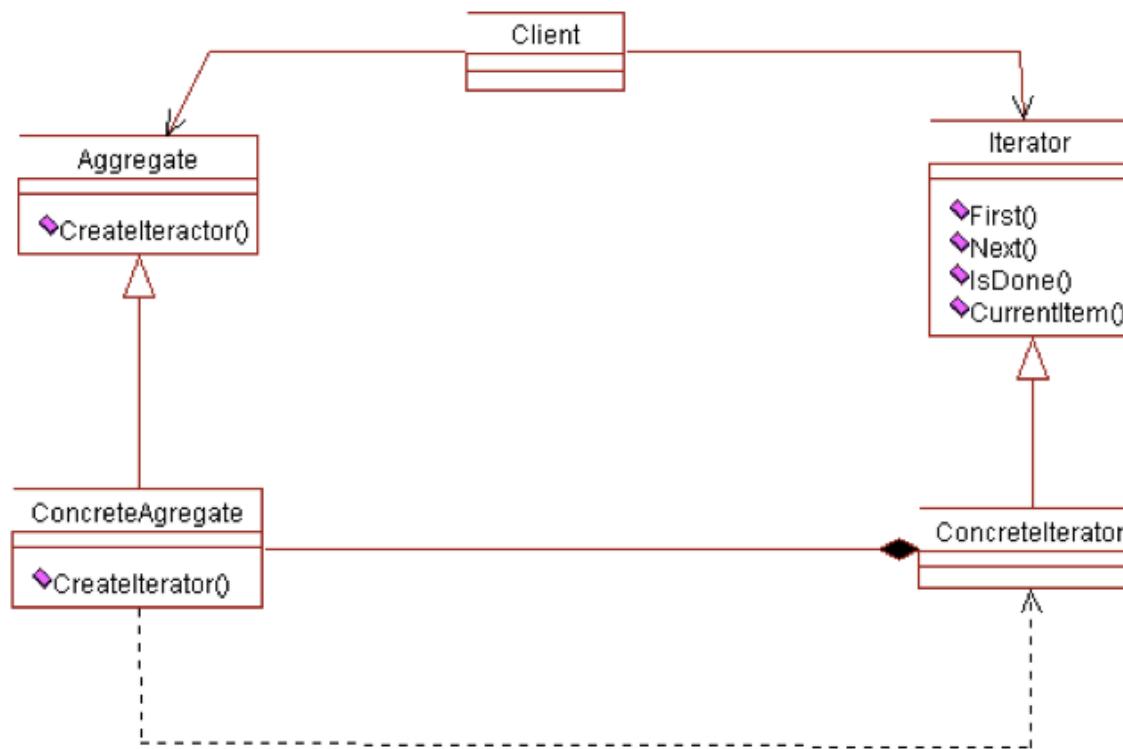
- O Iterator é um padrão de concepção identificado e que permite providenciar uma forma de aceder aos elementos de uma coleção de objectos, sem que seja necessário saber qual a sua representação interna
- basta para tal, que todas as coleções saibam criar um iterator

- Um iterador de uma lista poderia ser:



- o iterator precisa de ter mecanismos para:
  - aceder ao objecto apontado
  - avançar
  - determinar se chegou ao fim

- A estrutura geral do iterator será:



## OUTROS ITERADORES DE COLECÇÕES

```
Iterator<E> iterator(); // método que cria um Iterador sobre a colecção  
// que recebeu a mensagem. Se a colecção tem  
// objectos de tipo E o Iterador é de tipo E
```

- UM `Iterator<E>` É UMA ESTRUTURA COMPUTACIONAL QUE IMPLEMENTA UM ITERADOR SOBRE TODOS OS ELEMENTOS DA COLECÇÃO ORIGEM (SEM ORDEM PREDEFINIDA), USANDO OS MÉTODOS `hasNext()`, `next()` E `remove()`.

### EXEMPLO 1: Uso DE `iterator()` COM `while()` { ... }

```
Iterator<E> it = coleção.iterator();  
E elem; // tipo dos objectos que estão na colecção  
while(it.hasNext()) {  
    elem = it.next();  
    // fazer qq. coisa com elem  
}
```

### EQUIVALENTE A:

```
for(E elem : coleção) { ... }
```

**NOTA:** Mas `foreach` não permite parar procura!!

**ASSIM, EM ALGORITMOS TÍPICOS DE PROCURA TEM QUE SE USAR A CONSTRUÇÃO BASEADA EM iterator(), CF.**

```
Iterator<E> it = colecção.iterator(); // criar o Iterator<E>
E elem ; // variável do tipo dos objectos da colecção
boolean encontrado = false;
while(it.hasNext() && !encontrado) {
    elem = it.next();
    if(elem possui certa propriedade) encontrado = true;
}
```

**EXEMPLO :**

```
// encontrar o 1º ponto com coordenada X igual a Y
Ponto2D pontoCopia = null;
Iterator<Ponto2D> it = linha.iterator(); // criar o Iterator<E>
Ponto2D ponto ; // variável do tipo dos objectos da colecção
boolean encontrado = false;
while(it.hasNext() && !encontrado) {
    ponto = it.next();
    if(ponto.getX() == ponto.getY()) encontrado = true;
}
if(encontrado) pontoCopia = ponto.clone();
```

# Map<K, V>

Categoria de Métodos	API de Map<K, V>
Inserção de elementos	<code>put(K k, V v); putAll(Map&lt;? extends K, ? extends V&gt; m);</code>
Remoção de elementos	<code>remove(Object k);</code>
Consulta e comparação de conteúdos	<code>V get(Object k); boolean containsKey(Object k); boolean isEmpty(); boolean containsValue(Object v); int size();</code>
Criação de Iteradores	<code>Set&lt;K&gt; keySet(); Collection&lt;V&gt; values(); Set&lt;Map.Entry&lt;K, V&gt;&gt; entrySet();</code>
Outros	<code>boolean equals(Object o); Object clone()</code>

TreeMap<K, V> métodos adicionais
<code>TreeMap&lt;K, V&gt;()</code>
<code>TreeMap&lt;K, V&gt;(Comparator&lt;? super K&gt; c)</code>
<code>TreeMap&lt;K, V&gt;(Map&lt;? extends K, ? extends V&gt; m)</code>
<code>K firstKey()</code>
<code>SortedMap&lt;K, V&gt; headMap(K toKey)</code>
<code>K lastKey()</code>
<code>SortedMap&lt;K, V&gt; subMap(K fromKey, K toKey)</code>
<code>SortedMap&lt;K, V&gt; tailMap(K fromKey)</code>

# Regras para utilização de colecções

- Escolher com critério se a colecção a criar deve ser uma lista ou um conjunto (duplicados ou não) ou então uma correspondência entre chaves e valores
- Escolher para sets e maps uma classe de implementação adequada, cf. Hash (sem ordem especial) ou Tree (com comparação pré-definida ou definindo uma ordem de comparação)

# Regras para utilização de colecções

- Nunca usar os métodos pré-definidos **addAll()** ou **putAll()**. Em vez destes, usar antes o iterador for e fazer clone() dos objectos a copiar para a coleção cópia
- Sempre que possível, os resultados dos métodos devem ser generalizados para os tipos List<E>, Set<E> ou Map<K,V> em vez de devolverem classes específicas como ArrayList<E>, HashSet<E> ou TreeSet<E> ou HashMap<K,V>.
  - aumentando-se assim a abstracção

# Hierarquia de Classes e Herança

- *(Grady Booch) The Meaning of Hierarchy:*
  - “*Abstraction is a good thing, but in all except the most trivial applications, we may find many more different abstractions than we can comprehend at one time. Encapsulation helps manage this complexity by hiding the inside view of our abstractions. Modularity helps also, by giving us a way to cluster logically related abstractions. Still, this is not enough. A set of abstractions often forms a hierarchy, and by identifying these hierarchies in our design, we greatly simplify our understanding of the problem.*
  - Logo, “*Hierarchy is a ranking or ordering of abstractions.*



- Até agora só temos visto classes que estão ao mesmo nível hierárquico. No entanto...
- A colocação das classes numa hierarquia de especialização (do mais genérico ao mais concreto) é uma das características mais importantes da POO
- Esta hierarquia é importante:
  - ao nível da reutilização de variáveis e métodos
  - da compatibilidade de tipos

- No entanto, a tarefa de criação de uma hierarquia de conceitos (classes) é complexa, porque exige que se **classifiquem** os conceitos envolvidos
- A criação de uma hierarquia é do ponto de vista operacional um dos mecanismos que temos para criar novos conceitos a partir de conceitos existentes
  - a este nível já vimos a composição de classes

- Exemplos de composição de classes
  - um segmento de recta (exemplo da Ficha 3) é composto por duas instâncias de Ponto2D
  - um Triângulo pode ser definido como composto por três segmentos de recta ou por um segmento e um ponto central, ou ainda por três pontos

- Uma outra forma de criar classes a partir de classes já existentes é através do mecanismo de herança.
- Considere-se que se pretende criar uma classe que represente um Ponto 3D
  - quais são as alterações em relação ao Ponto2D?
  - mais uma v.i. e métodos associados

- A classe Ponto2D (incompleta):

```
public class Ponto2D {  
    // Construtores  
    public Ponto2D(int cx, int cy) { x = cx; y = cy; }  
    public Ponto2D(){ this(0, 0); }  
    // Variáveis de Instância  
    private int x, y;  
    // Métodos de Instância  
    public int getX() { return x; }  
    public int getY() { return y; }  
    public void desloca(int dx, int dy) {  
        x += dx; y += dy;  
    }  
    public void somaPonto(Ponto2D p) {  
        x += p.getX(); y += p.getY();  
    }  
    public Ponto2D somaPonto(int dx, int dy) {  
        return new Ponto2D(x += dx, y+= dy);  
    }  
    public String toString() {  
        return new String("Pt= " +x + ", " + y);  
    } }
```

- o esforço de codificação consiste em acrescentar uma v.i. (z) e getZ() e setZ()

- O mecanismo de herança proporciona um esforço de programação diferencial

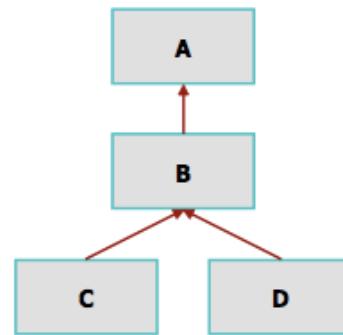
$$\begin{aligned}\text{Ponto3D} &= \text{Ponto2D} + \Delta_{\text{prog}} \\ 1 \text{ ponto3D} &\Leftrightarrow 1 \text{ ponto2D} + \Delta_{\text{var}} + \Delta_{\text{met}}\end{aligned}$$

- ou seja, para ter um Ponto3D precisamos de tudo o que existe em Ponto2D e acrescentar um *delta* que consiste nas características novas
- A classe Ponto3D aumenta, refina, detalha, especializa, a classe Ponto2D

- Como se faz isto?
  - de forma ad-hoc, sem suporte, através de um mecanismo de copy&paste
  - usando composição, isto é, tendo como v.i. de Ponto3D um Ponto2D
  - através de um mecanismo existente de base nas linguagens por objectos que é a noção de hierarquia e herança

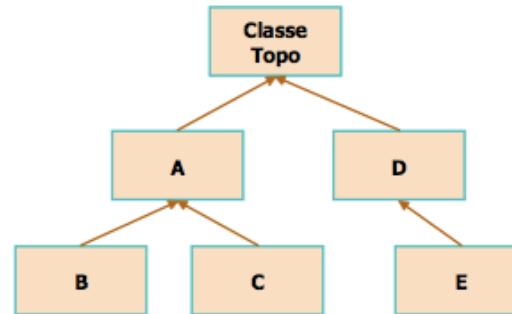
```
/**  
 * Ponto3D através de composição.  
 */  
  
public class Ponto3D {  
    private Ponto2D p;  
    private int z;  
  
    public Ponto3D() {  
        this.p = new Ponto2D();  
        this.z = 0;  
    }  
  
    public Ponto3D(int x, int y, int z) {  
        this.p = new Ponto2D(x,y);  
        this.z = z;  
    }  
  
    //...  
    public int getX() {return this.p.getX();}  
    public int getY() {return this.p.getY();}  
    public int getZ() {return this.z;}
```

- Exemplo:



- A é superclasse de B
- B é superclasse de C e D
- C e D são subclasses de B
- B especializa A, C e D especializam B ( e A!)

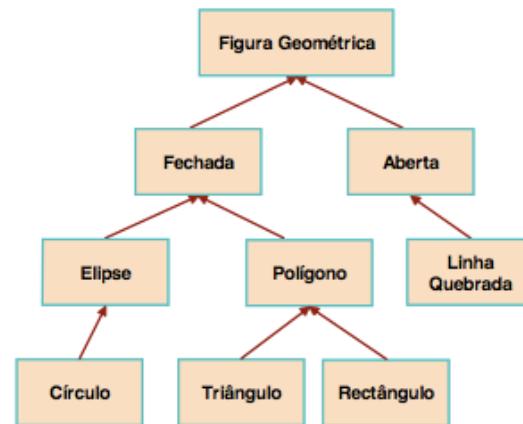
- Hierarquia típica em Java:



- hierarquia de herança simples (por oposição, p.ex., a C++)
- O que significa do ponto de vista semântico dizer que duas classes estão hierarquicamente relacionadas?

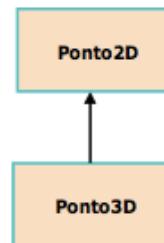
- no paradigma dos objectos a hierarquia de classes é uma hierarquia de especialização
- uma subclasse de uma dada classe constitui uma especialização, sendo por definição mais detalhada que a classe que lhe deu origem
- isto é, possui **mais** estado e **mais** comportamento

- A exemplo de outras taxonomias, a classificação do conhecimento é realizada do geral para o particular



- a especialização pode ser feita nas duas vertentes: estrutural e comportamental

- voltando ao Ponto3D:



- ou seja, Ponto3D é subclasse de Ponto2D

```
public class B extends A { ...  
public class Ponto3D extends Ponto2D { ...
```

- como foi dito, uma subclasse herda estrutura e comportamento da sua classe:



# O mecanismo de herança

- se uma classe B é subclasse de A, então:
  - B é uma especialização de A
  - este relacionamento designa-se por “é um” ou “é do tipo”, isto é, uma instância de B pode ser designada como sendo um A
  - implica que aos atributos e métodos de A se acrescentou mais informação

- Se uma classe B é subclasse de A:
  - se B **pertence** ao mesmo package de A, B herda todas as variáveis e métodos de instância que não são private.
  - se B **não pertence** ao mesmo package de A, B herda todas as variáveis e métodos de instância que não são private ou package. Herda tudo o que é public ou protected.

- B pode **definir** novas variáveis e métodos de instância próprios
- B pode **redefinir** variáveis e métodos de instância herdados
- variáveis e métodos de classe não são herdados: podem ser redefinidos.
- métodos construtores não são herdados

- na definição que temos utilizado nesta unidade curricular, as nossas variáveis de instância são declaradas como **private**
- que impacto é que isto tem no mecanismo de herança?
  - total, vamos deixar de poder referir as v.i. da superclasse que herdamos pelo nome
  - vamos utilizar os métodos de acesso, `getX()`, para aceder aos seus valores

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

# 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 declarada na subclasse diz-se que 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

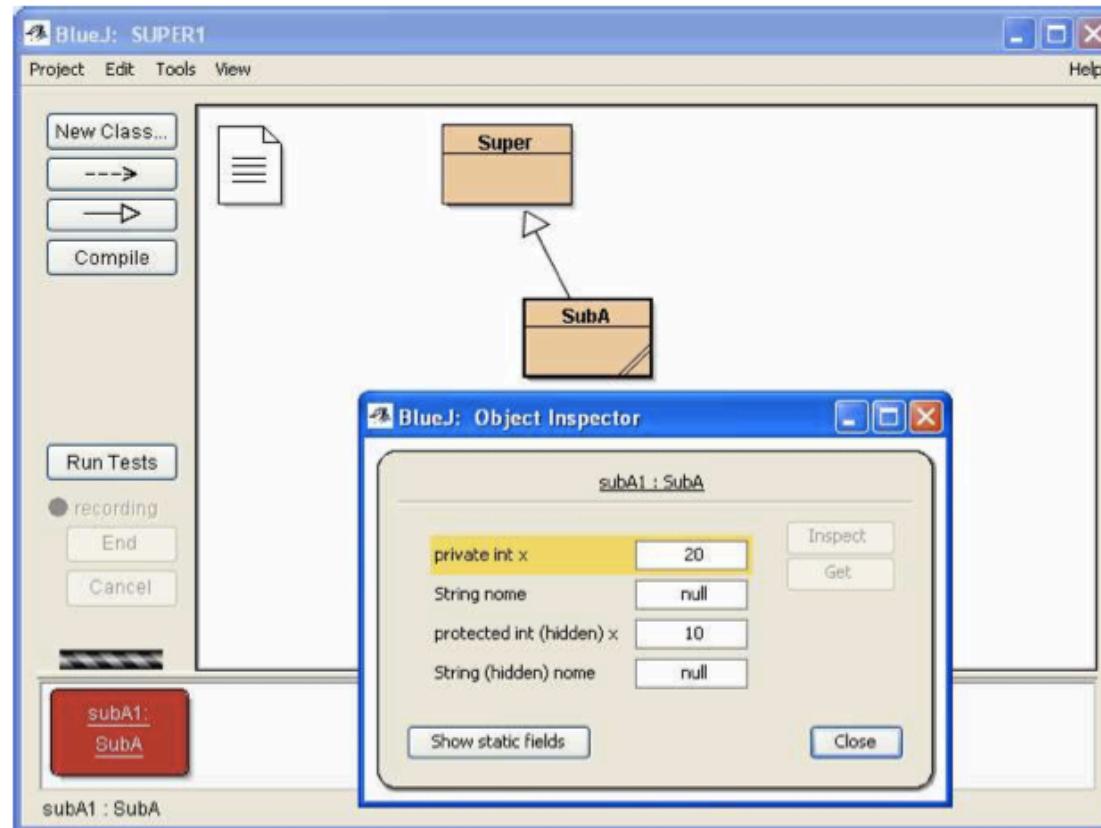
```
public class Super {  
    protected int x = 10;  
    String nome;  
    // Métodos  
    public int getX() { return x; }  
    public String classe() { return "Super"; }  
    public int teste() { return this.getX(); }  
}
```

- e uma subclasse SubA

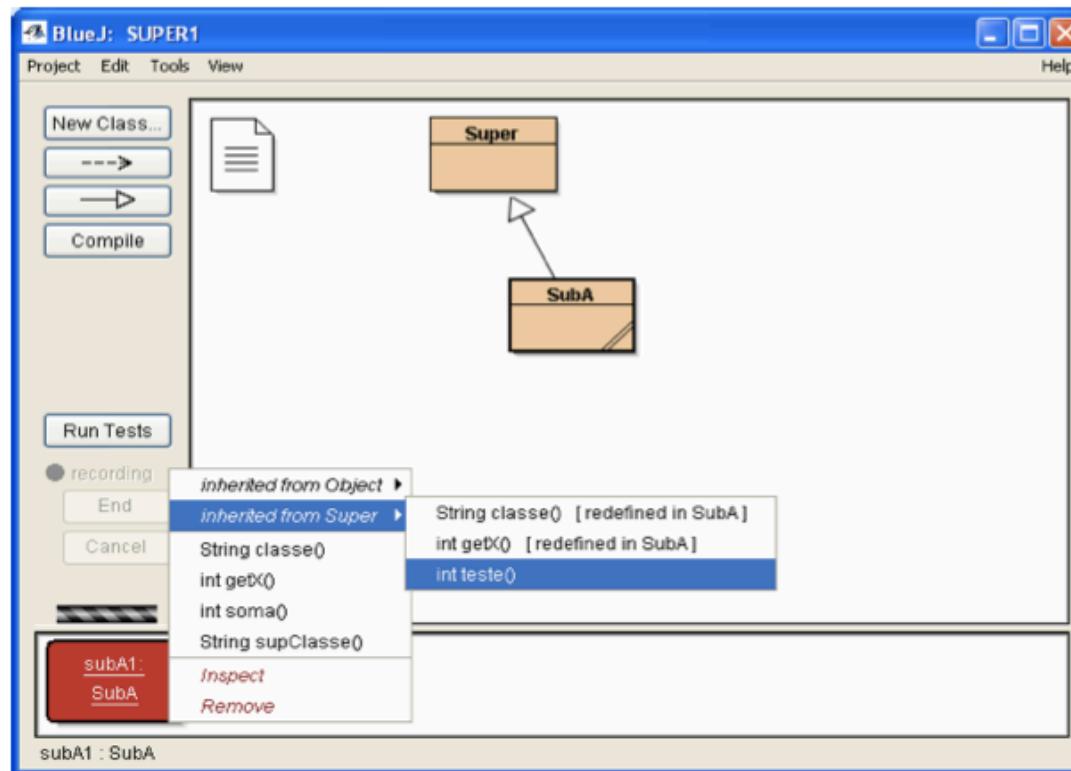
```
public class SubA extends Super {  
    private int x = 20; // "shadow"  
    String nome; // "shadow"  
    // Métodos  
    public int getX() { return x; }  
    public String classe() { return "SubA"; }  
    public String superClass() { return super.classe(); }  
    public int soma() { return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

- 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



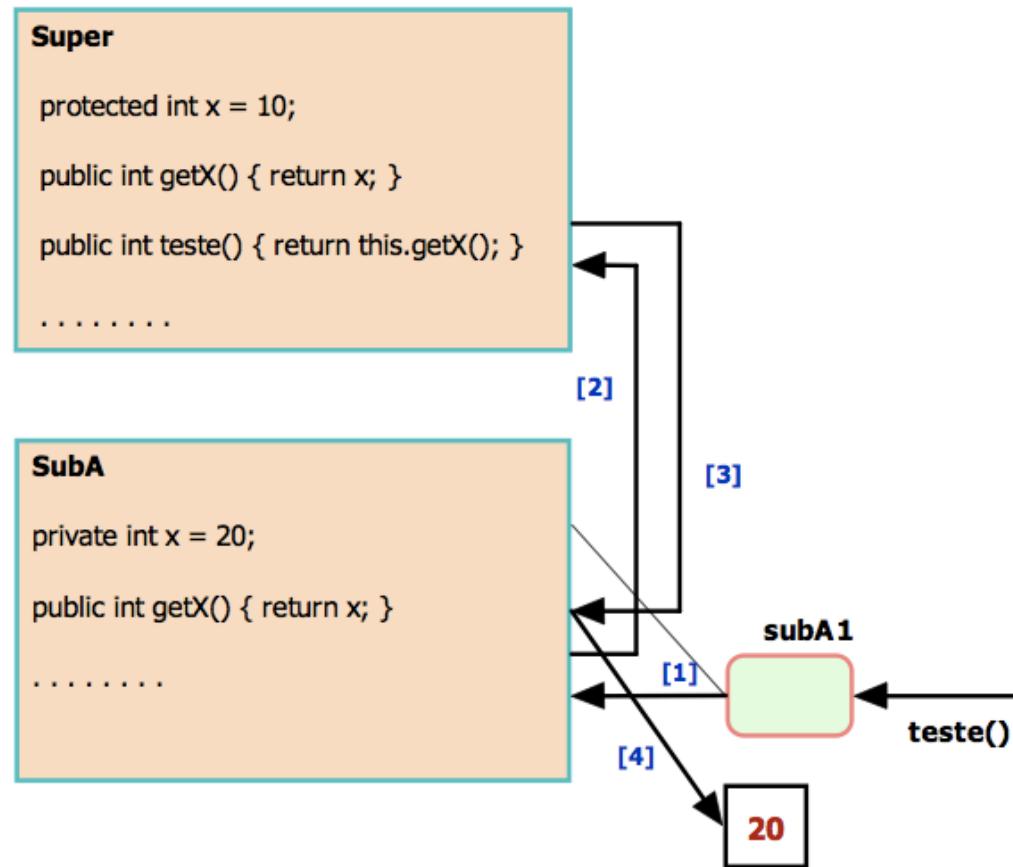
- no BlueJ é possível ver os métodos definidos na classe e os herdados



- 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 avaliação de this.m()

- de forma geral, a expressão **this.m()**, onde quer que seja encontrada no código de um método de uma classe (independentemente da localização na hierarquia), corresponde sempre à execução do método **m()** da classe do receptor da mensagem

# Modificadores e redefinição de métodos

- a possibilidade de redefinição de métodos está condicionada pelo tipo de modificadores de acesso do método da superclasse (`private`, `public`, `protected`, `package`) e do método redefinidor
- o método redefinidor não pode diminuir o nível de acessibilidade do método redefinido

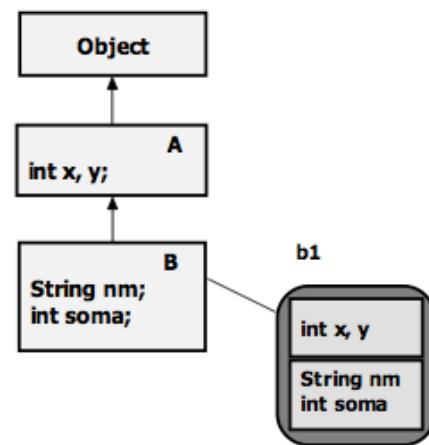
- os métodos public podem ser redefinidos por métodos public
- métodos protected por public ou protected
- métodos package por public ou protected ou package

# 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 subclass 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 seja, os construtores de A
- dessa forma, o construtor de B deve invocar os construtores de A para inicializar as v.i. declaradas em A

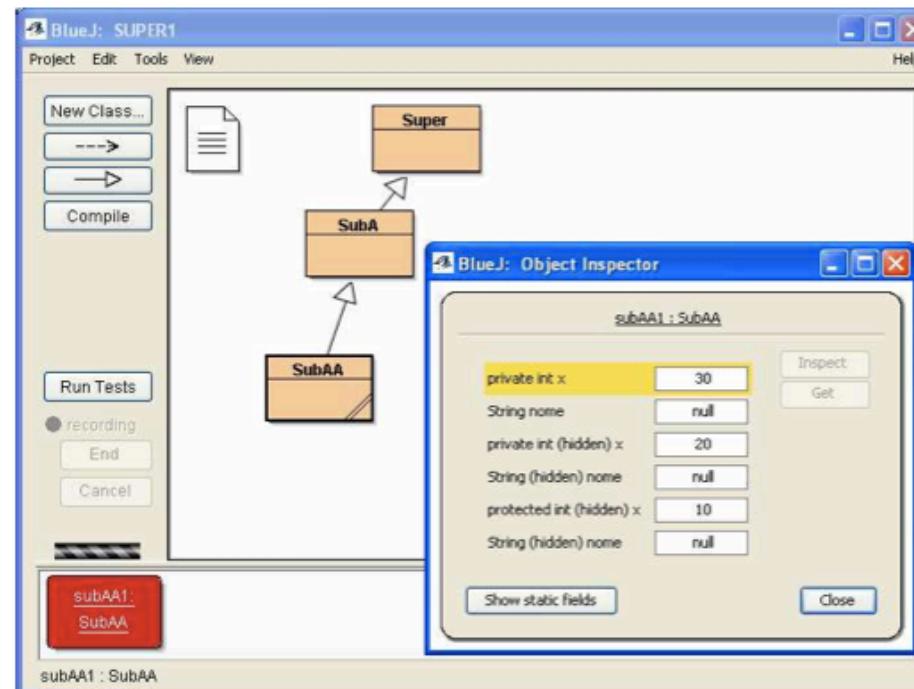
- em Java para que seja possível a invocação do construtor de uma superclasse esta deve ser feita logo no início do construtor da subclasse
- recorrendo a super(...,...), em que a verificação do construtor a invocar se faz pelo matching dos parâmetros e respectivos tipos de dados
- de facto a invocação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

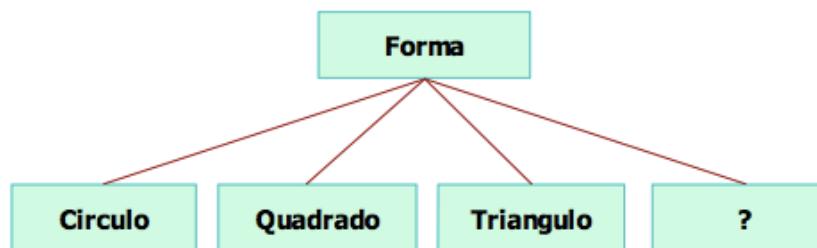


# Classes Abstractas

- 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 I: 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 `perímetro()` 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; }
}
```

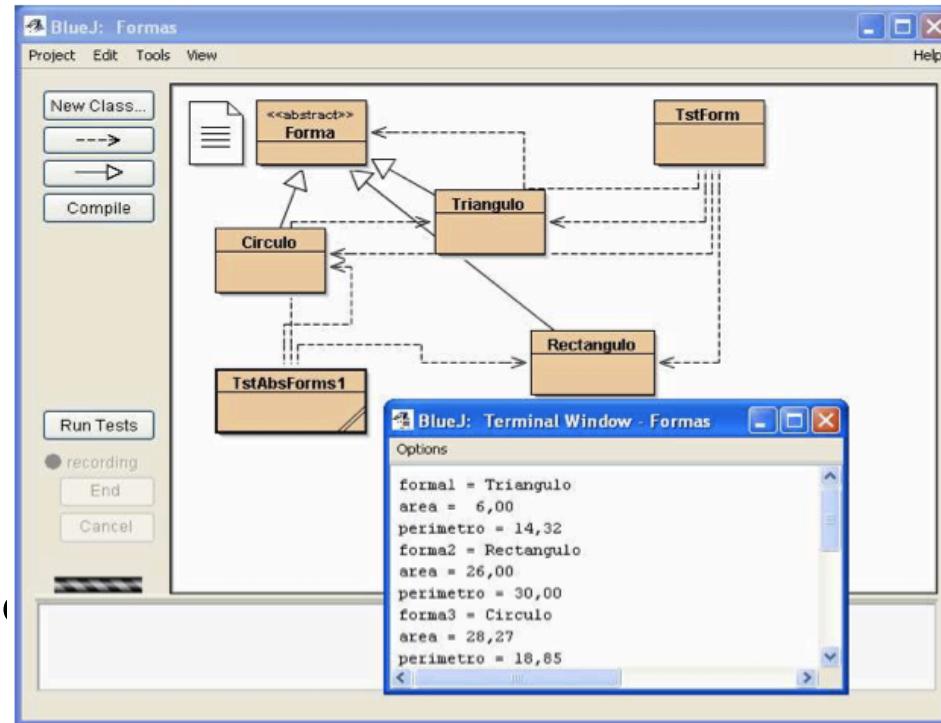
## ● Classe Rectangulo

```
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!!**



# 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

- é preciso saber qual a compatibilidade entre os tipos das diferentes classes (superclasses e subclasses)
- a questão importante é saber se 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 variável e a atribuição de valor
  - B é uma subclasse de A, pelo que está correcto
  - mas o que acontece quando se executa a1.m( )?

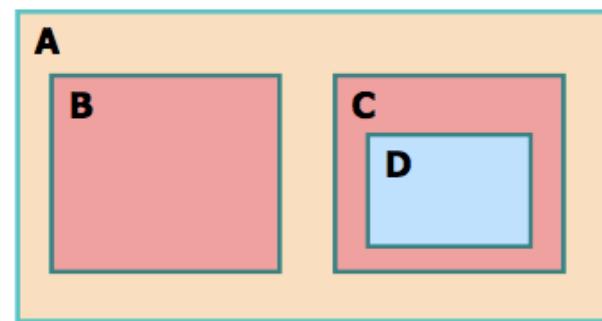
- o compilador tem de verificar se `m()` existe em A ou numa sua superclasse
- se existir é como se estivesse declarado em B
- a expressão é correcta do ponto de vista do compilador
- em tempo de execução terá de ser determinado qual é o método a ser invocado.

- o interpretador, em tempo de execução, faz o *dynamic binding* procurando determinar em função do valor contido qual é o método que deve invocar
- se várias classes da hierarquia implementarem o método m(), então o interpretador executa o método associado ao tipo de dados da **classe do objecto**
- Na expressão f.`toString()`, da declaração anterior, qual é o método que é invocado?  
O `toString` da classe Triangulo!!

- Seja o seguinte código

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

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



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

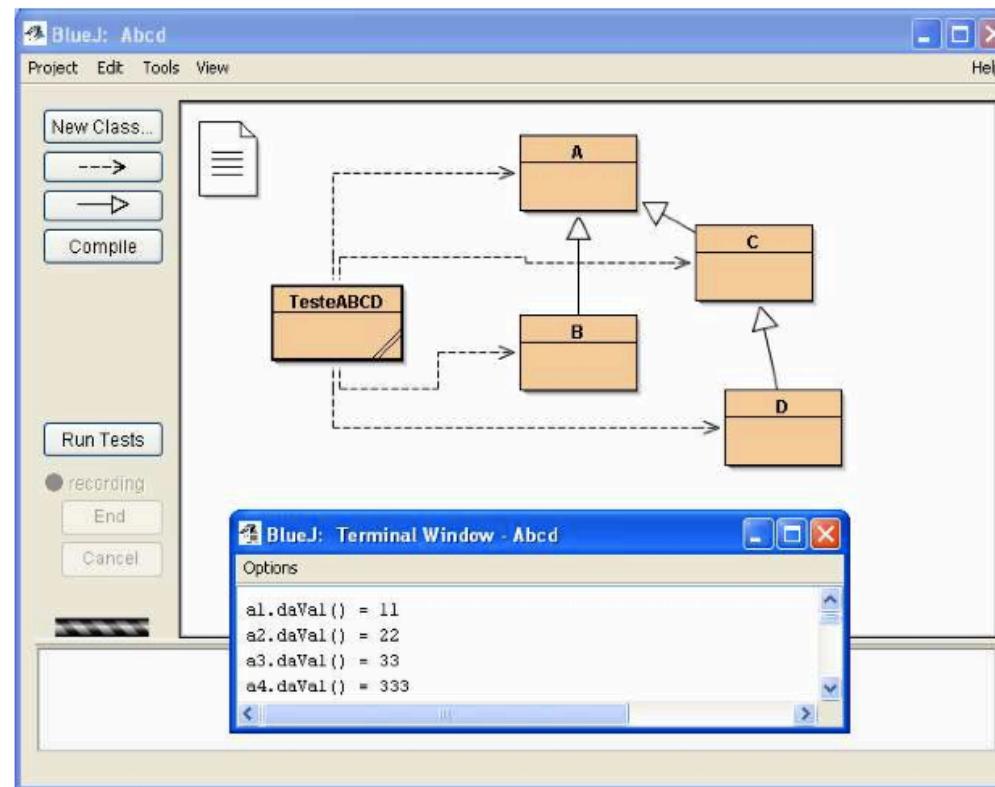
- e as seguintes inicializações de variáveis

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

- qual é o resultado deste programa?

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

- como o interpretador executa o algoritmo de procura dinâmica de métodos, executando `meth()` 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;  
    else {  
        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;  
    else {  
        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

# Herança vs Composição

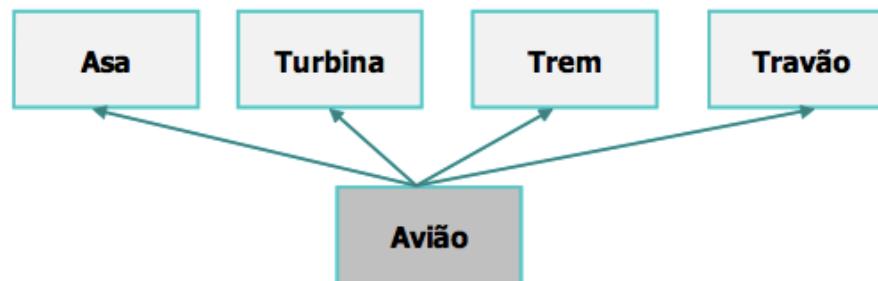
- Herança e composição são duas formas de relacionamento entre classes
  - são no entanto abordagens muito distintas e constitui um erro muito comum achar que podem ser utilizadas para o mesmo fim
- existe uma tendência para se confundir herança com composição

- 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:



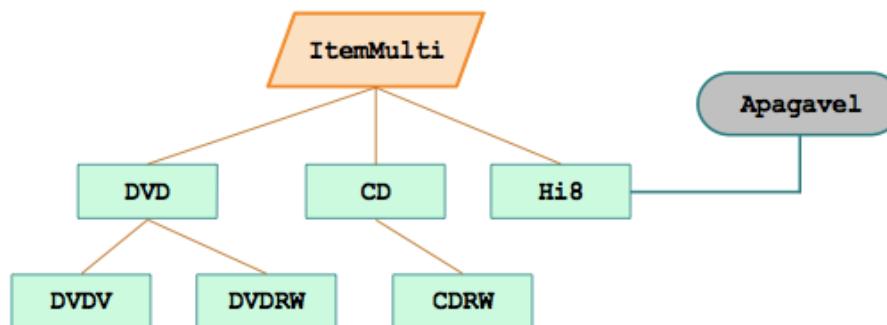
- embora o que se pretende ter é composição. Na solução apresentada o avião apenas tem **uma** asa, **uma** turbina, **um** trem de aterragem e **um** travão.

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

- quando uma classe (apesar de 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**

# Ainda sobre interfaces...

- Uma hierarquia típica



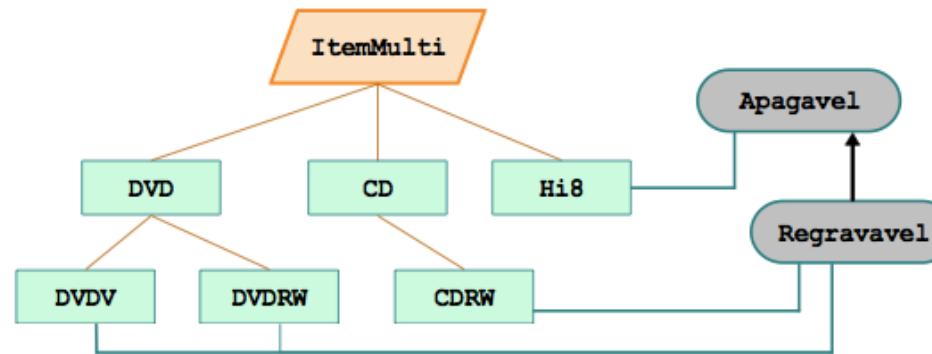
```
public class Hi8 extends ItemMulti implements Apagavel {  
    //  
    private int minutos;  
    private double ocupacao;  
    private int gravacoes;  
    . . .  
    // implementação de Apagavel  
    public void apaga() { ocupacao = 0.0; gravacoes = 0; }  
}
```

- Qualquer instância de Hi8 é também do tipo Apagavel, ou seja:

```
Hi8 filme1 = new Hi8("A1", "2005", "obs1", 180, 40.0, 3);
Apagavel apg1 = filme1;
apg1.apaga();
```

- no entanto, a uma instância de Hi8 que vemos como sendo um Apagavel, apenas lhe poderemos enviar métodos definidos nessa interface (i.e. nesse tipo de dados)

- Temos também a possibilidade de ter vários tipos de dados válidos para diferentes objectos.



- Por vezes, o que provavelmente acontece com **Regravavel**, a interface é apenas um marcador.

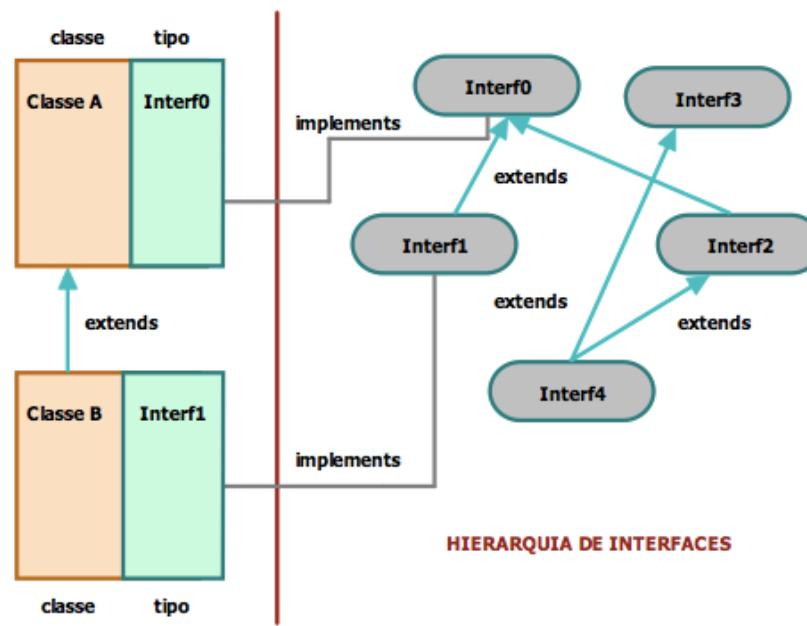
- A verificação de tipo pode ser feita da mesma forma que fazemos para as classes, com instanceof

```
ItemMulti[] filmes = new ItemMulti[ 500];
// código para inserção de filmes no array...
int contaReg = 0;
for(ItemMulti filme : filmes)
    if (filme instanceof Regravavel) contaReg++
out.printf("Existem %d items regraváveis.", contaReg);
```

- na expressão acima não se está a validar a classe, mas sim o tipo de dados estático

- Do ponto de vista da concepção de arquitecturas de objectos, as interfaces são importantes para:
  - reunirem similaridades comportamentais, entre classes não relacionadas hierarquicamente
  - definirem novos tipos de dados
  - conterem a API comum a vários objectos, sem indicarem a classe dos mesmos

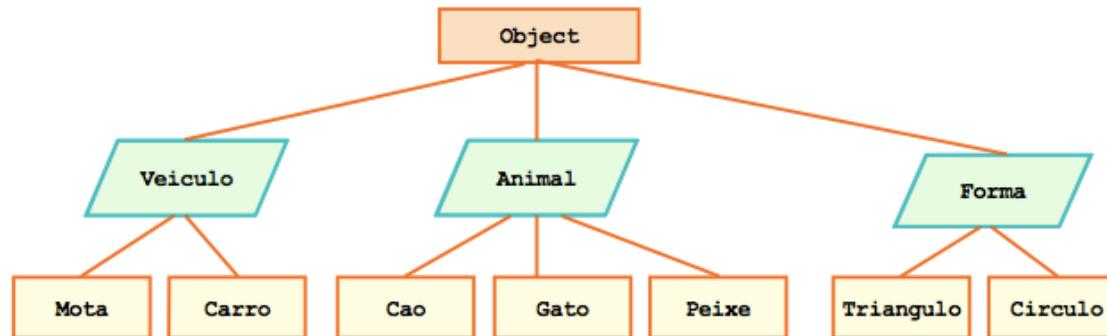
- O modelo geral é assim:



- onde coexistem as noções de classe e interface, bem assim como as duas hierarquias

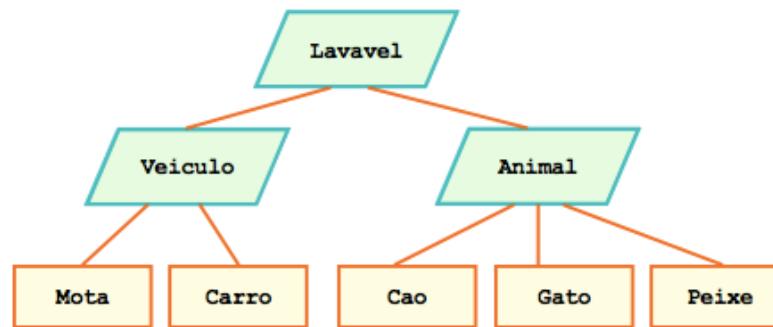
- Uma classe passa a ter duas vistas (ou classificação) possíveis:
  - é subclasse, por se enquadrar na hierarquia normal de classes, tendo um mecanismo de herança simples de estado e comportamento
  - é subtípo, por se enquadrar numa hierarquia múltipla de definições de comportamento abstracto (puramente sintático)

- Existem situações que apenas são possíveis de satisfazer considerando as duas hierarquias.



- se fosse importante saber os objectos desta hierarquia que poderiam ser lavados, então...

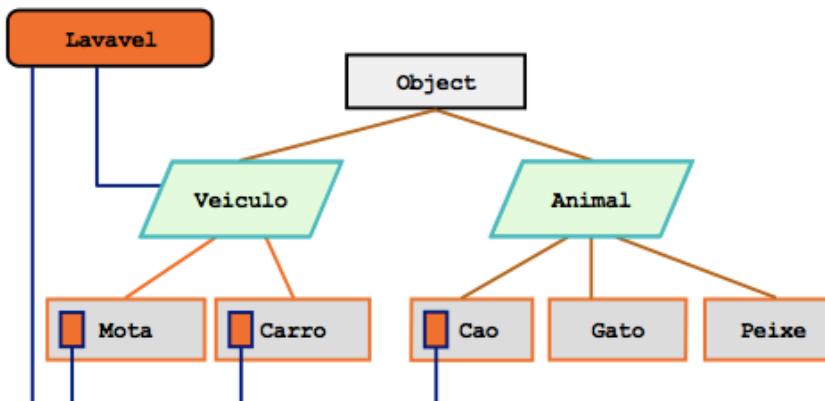
- A hierarquia “correcta” (que dava mais jeito) seria:



- no entanto, esta solução obrigaría objectos não “laváveis”, a ter um método aLavar( )

- Com a utilização de ambas as hierarquias poderemos ter:

```
public interface Lavavel {  
    public void aLavar();  
}
```

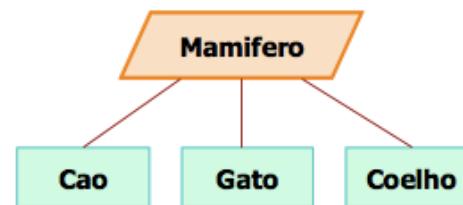


# Em resumo...

- As interfaces Java são especificações de tipos de dados. Especificam o conjunto de operações a que respondem objectos desse tipo
- Uma instância de uma classe é imediatamente compatível com:
  - o tipo da classe
  - o tipo da interface (se estiver definido)

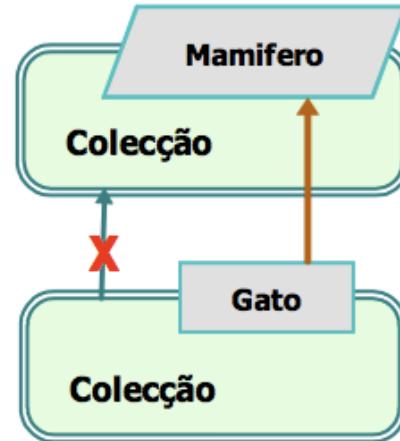
# Tipos Parametrizados

- Seja a seguinte hierarquia:

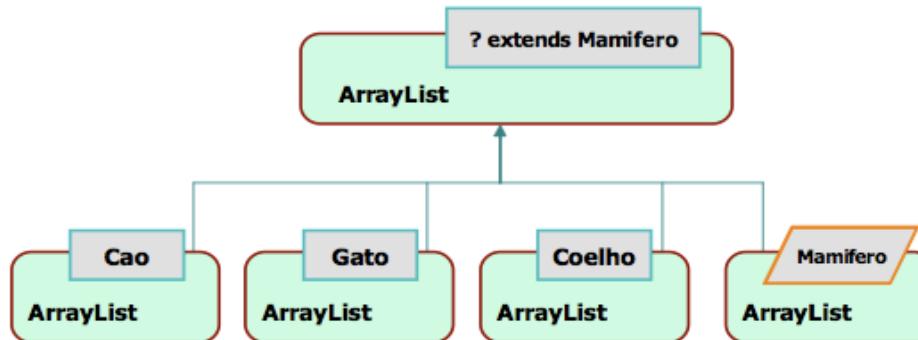


- e considere-se uma coleção de elementos do tipo Mamífero

- Um arraylist de mamíferos,  
`ArrayList<Mamifero>` pode conter  
instâncias de Cão, Gato, Coelho, etc.
- no entanto esse arraylist não é supertipo  
dos arraylist de subtipos de mamíferos!!
- A hierarquia de `ArrayList<E>` não tem a  
mesma estruturação da hierarquia de E



- o arrayList de todos os arraylists de Mamifero e dos seus subtipos é o arrayList que se declara como:
- ***ArrayList<? extends Mamifero>***



```
Coleccao<? extends Mamifero> =  
Coleccao<Gato> ou  
Coleccao<Cao> ou  
Coleccao<Coelho>
```

```
public void juntaMamif(Set<? extends Mamifero> cm) {  
    ...  
}
```

- Desta forma passa a ser possível ter declarações como:

```
ArrayList<Mamifero> mamifs = new ArrayList<Mamifero>();  
mamifs.addAll(criaCaes()); // junta ArrayList<Cao>  
mamifs.addAll(criaGatos()); // junta ArrayList<Gato>
```

- o que era impossível no modelo anterior, na medida em que um `ArrayList<Cao>` não é compatível com `ArrayList<Mamifero>`

# Programação Orientada aos Objectos

-- fecho do semestre --

# Resultados de aprendizagem

- Compreender os conceitos fundamentais da PPO( Objectos, Classes, Herança e Polimorfismo);
- Compreender como os conceitos básicos da PPO são implementados em construções JAVA;
- Compreender princípios e técnicas a empregar em programação de larga escala;
- Desenvolver o modelo de classes e interfaces para um dado problema de software (modelação);
- Desenvolver e implementar aplicações Java de média escala, seguras, robustas e extensíveis;

# Programa: teórica

- Paradigma da programação por Objectos: Abstracção de Dados, Encapsulamento e Modularidade.
- Objectos: estrutura e comportamento.
- Mensagens. Classes, hierarquia e herança. Classes abstractas.
- Herança versus Composição.
- Compatibilidade de tipos. O princípio da substituição. Dynamic binding. Polimorfismo.

# Programa prática

- JAVA: Plataforma J2SE: JDK, JVM e byte-code.
- Construções básicas: tipos primitivos e operadores. Estruturas de controlo. I/O básico. Arrays.
- Nível dos objectos: Classes e instâncias. Construtores. Métodos e variáveis de instância. Modificadores de acesso. Métodos e variáveis de classe.
- Colecções genéricas. Interfaces parametrizadas. Iteradores. Tipos List, Map e Set.
- Hierarquia de classes e herança. Overloading e overriding de métodos. Classes Abstractas. Interfaces e tipos definidos pelo utilizador. Tipo estático e dinâmico. Procura dinâmica de métodos. Polimorfismo e extensibilidade.
- Streams: de caracteres, de bytes e de objectos.
- Excepções.