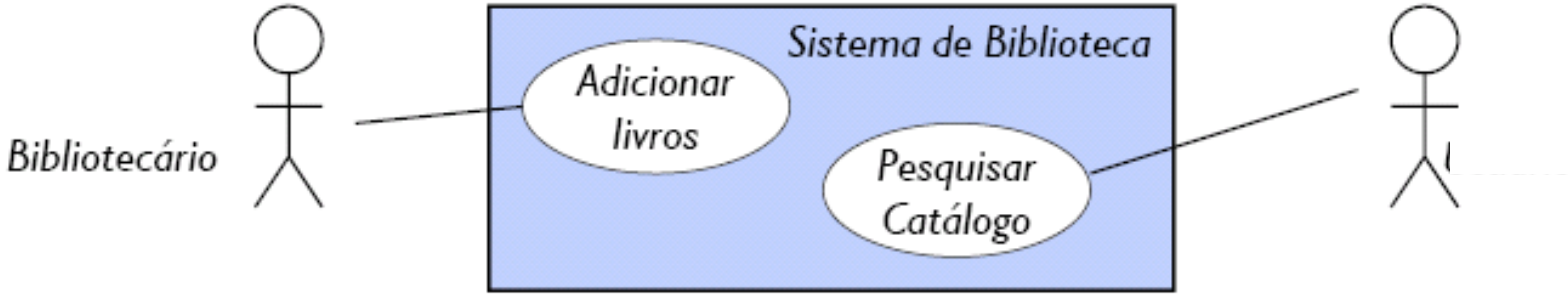
Java
Encapsulamento, Classes e Objetos

UA, DETI, Programação III
José Luis Oliveira, Carlos Costa
2014/15

O que é Orientação por Objetos?

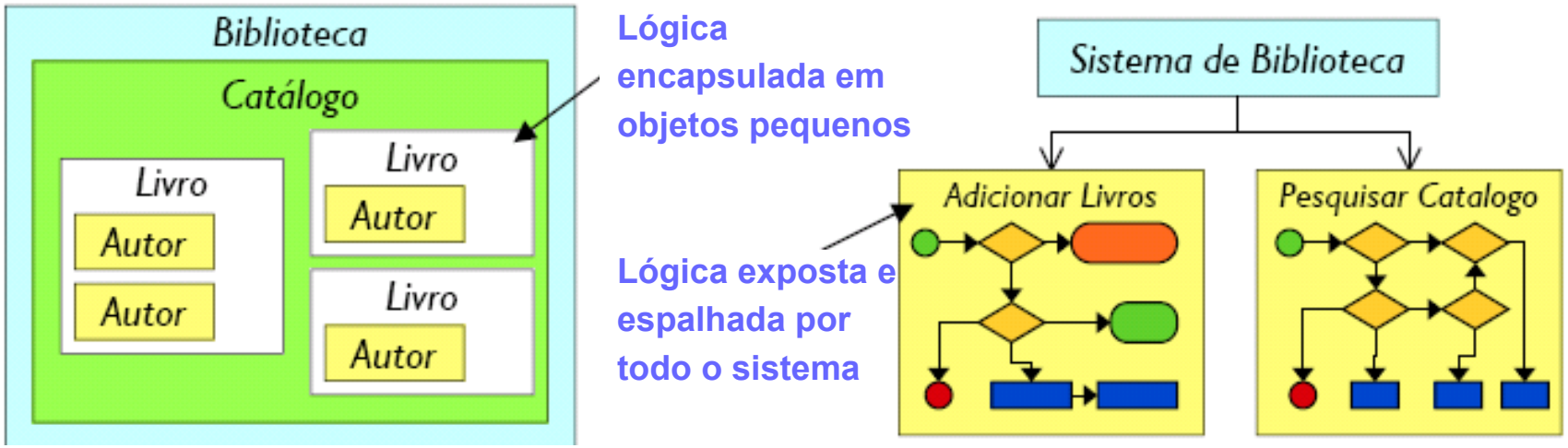
- Paradigma do momento na engenharia de software
  - Afecta análise, projeto (design) e programação
- A **análise** orientada por objetos
  - Determina **o que** o **sistema** deve fazer: Quais os atores envolvidos? Quais as atividades a serem realizadas?
  - Decompõe o sistema em **objetos**: Quais são? Que tarefas cada objeto terá que fazer?
- O **design** orientado por objetos
  - Define **como** o sistema será implementado
  - Modela os relacionamentos entre os objetos e atores (pode-se usar uma linguagem específica como UML)
  - Utiliza e reutiliza abstrações como classes, objetos, funções, frameworks, APIs, padrões de projeto

POO (1) versus Prog. Procedimental (2)



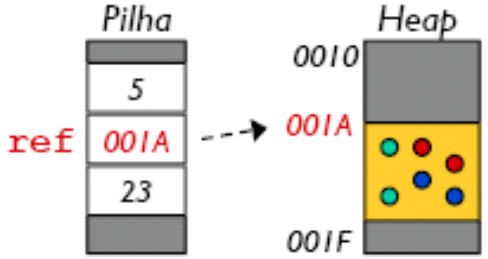
(1) Trabalha no espaço do problema (casos de uso simplificados em objetos)
– abstrações mais simples e mais próximas do mundo real

(2) Trabalha no espaço da solução (casos de uso decompostos em procedimentos algorítmicos)
– abstrações mais próximas do mundo do computador



Objetos

- Em Java os objetos são armazenados na memória heap e manipulados através de uma referência (variável), guardada na pilha.
  - Têm estado (atributos)
  - Têm comportamento (métodos)
  - Têm identidade (a referência)



- Todos os objetos são manipulados através de referências

```
Pessoa nome1, nome2;
nome1 = new Pessoa("Manuel");
nome2 = nome1;
```

- Todos os objetos devem ser explicitamente criados.

```
Circulo c1 = new Circulo(p1, 5);
String s = "Buga";  // Strings são excepção!
```

- Casos especiais: tipos primitivos
  - Não são objetos em Java

Criação explícita - tipos primitivos

- Por questões de eficiência de programação e de código, podemos declarar variáveis automáticas associadas a cada um destes tipos base.
  - não são referências – são armazenadas na pilha
  - Exemplo:

```
char ch = 'B';
int x = 23;
```

- Os tipos de dados primitivos têm associadas classes "wrappers"
  - Neste caso são armazenados no heap (como todos os objetos).

```
Character ch1 = new Character(ch);
Integer idade = new Integer(23);

// Sintaxe melhor: Box/Unbox
Character ch1 = ch;
Integer idade = 23;
```

Tipos primitivos em Java

- Têm apenas identidade (nome da variável) e estado (valor literal armazenado na variável)
  - - dinâmicos; + eficientes
- Classe 'Wrapper' faz transformação, se necessário

Tipo	Tamanho	Mínimo	Máximo	Default	'Wrapper'
boolean	—	—	—	false	Boolean
char	16-bit	Unicode 0	Unicode 2^{16} - 1	\u0000	Character
byte	8-bit	-128	+127	0	Byte
short	16-bit	-2^{15}	$+2^{15}-1$	0	Short
int	32-bit	-2^{31}	$+2^{31}-1$	0	Integer
long	64-bit	-2^{63}	$+2^{63}-1$	0	Long
float	32-bit	IEEE754	IEEE754	0.0	Float
double	64-bit	IEEE754	IEEE754	0.0	Double
void	—	—	—	—	Void

Valores de omissão para tipos primitivos

- Se uma variável for utilizada como membro de uma classe o compilador encarrega-se de inicializá-la por omissão
- Isto não é garantido no caso de variáveis locais pelo que devemos sempre inicializar todas as variáveis

Tipo	Valor por omissão
boolean	**false**
char	**'\u0000' (null)**
byte	**(byte)0**
short	**(short)0**
int	**0**
long	**0L**
float	**0.0f**
double	**0.0d**

Criação explícita - Vectores

- Um vector em Java representa um conjunto de referências
  - aplicam-se as regras anteriores nos valores por omissão

```
int[] x;   // ou int x[] (C style); ref to int array

x = new int[10]; // reserva array

int[] a = new int[10];   // 10 int

int[] a1 = { 1, 2, 3, 4, 5 };

XptoCl[] xC = new XptoCl[10]; // 10 refs
```

Vectores multidimensionais

- Vectores multidimensionais são vectores de vectores
  - É possível criar vectores rectangulares

```
int [][][] prisma = new int [3][2][2];
```

  - ou criar apenas o primeiro nível e depois cada subsequente com dimensões diferentes

```
int [][][] prisma2 = new int [3][][];
prisma2[0] = new int[2][];
prisma2[1] = new int[3][2];
prisma2[2] = new int[4][4];
prisma2[0][0] = new int[5];
prisma2[0][1] = new int[3];
```

Alcance/*Scope*

- Uma variável automática pode ser utilizada desde que é definida até ao final desse contexto
- Cada bloco pode ter os seus próprios objetos

```
{   int k;
    {  int i;
    } // 'i' não é visível aqui
} // 'k' não é visível aqui
```

- Exemplo ilegal

```
{  int x = 12;
   {  int x = 96; /* erro! */
   }
}
```

Tipos Abstractos de Dados

- Construção de modelos de objetos
  - (ADT) consiste em modular entidades de acordo com os seus serviços.
- O conceito de classe é uma forma de implementar tipos abstractos de dados em POO

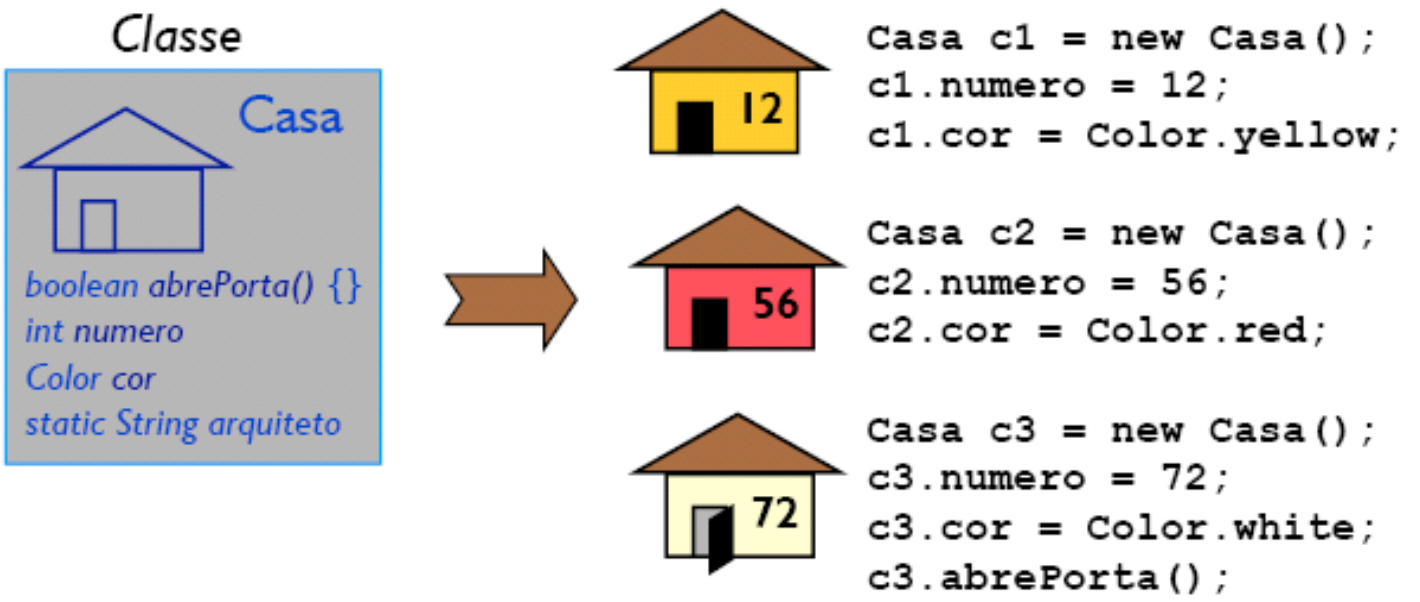
```
Class UserDefinedType { /* ... */ }
```

- Esta declaração introduz um novo tipo de dados

```
UserDefinedType myObj = new UserDefinedType();
```

O que é uma classe?

- Classes são especificações para criar objetos
- Uma classe representa um tipo de dados complexo
- Classes descrevem
  - Tipos dos dados que compõem o objeto (o que podem armazenar)
  - Métodos que o objeto pode executar (o que podem fazer)



Exemplo de classe

Definição da classe (tipo) Mensagem em Java

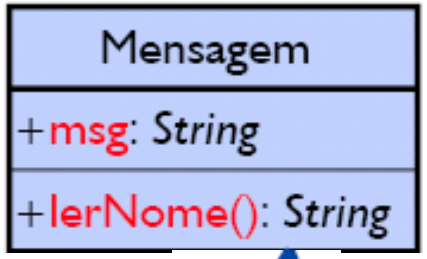
Membros de classe

```
public class Mensagem {

    public String msg = "";

    public String lerNome() {
        String nomeEmMaiusculas =
                        msg.toUpperCase();
        return nomeEmMaiusculas;
    }
}
```

atributo

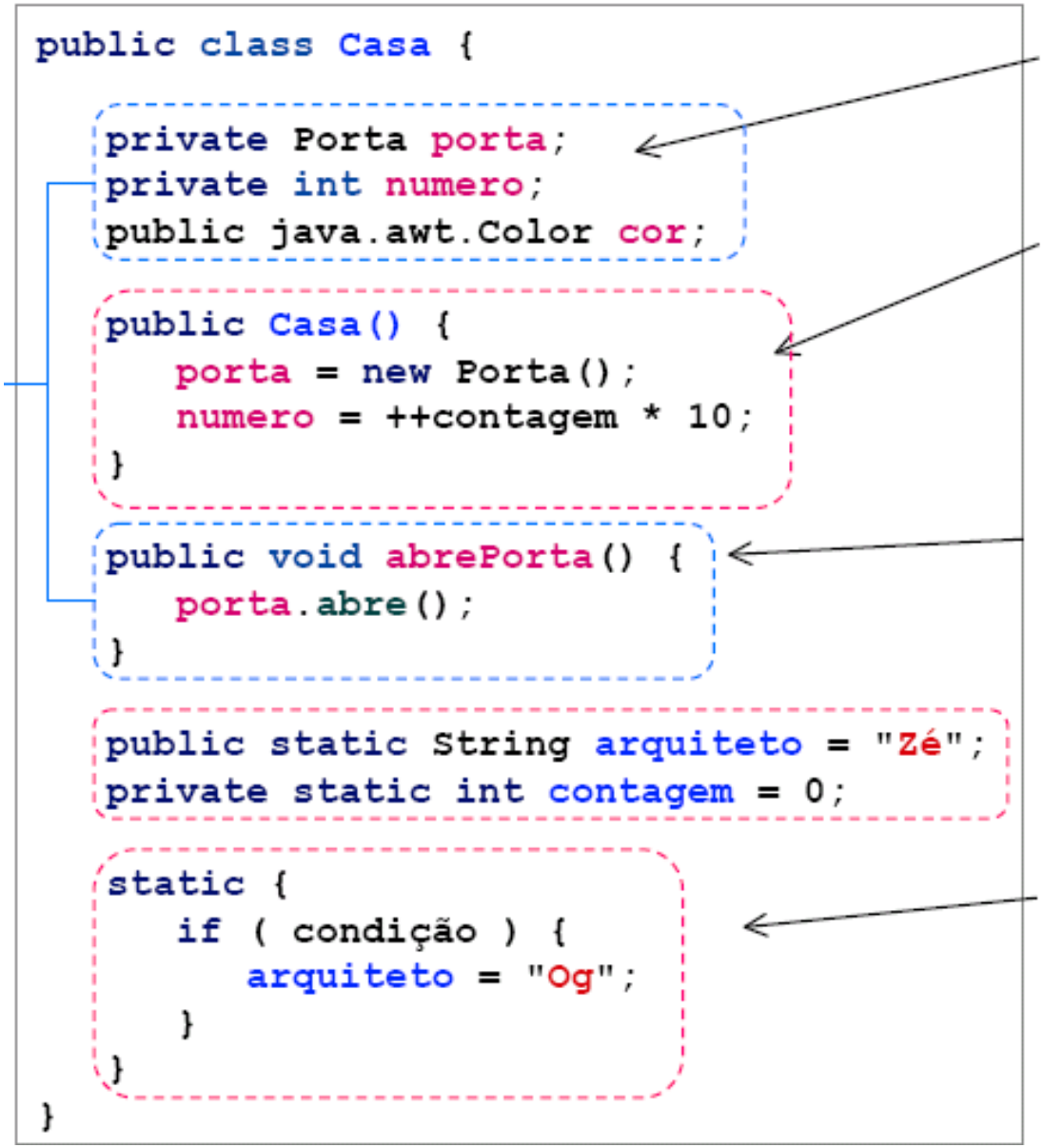
método

Representação em UML



*Esta é a **interface pública** da classe. É só isto que interessa a quem vai usá-la. Os detalhes (código) estão **encapsulados**.*

Exemplo de classe



```
public class Casa {

    private Porta porta;
    private int numero;
    public java.awt.Color cor;

    public Casa() {
        porta = new Porta();
        numero = ++contagem * 10;
    }

    public void abrePorta() {
        porta.abre();
    }

    public static String arquiteto = "Zé";
    private static int contagem = 0;

    static {
        if ( condição ) {
            arquiteto = "Og";
        }
    }
}
```

***Atributos de instância**: cada objeto poderá armazenar valores diferentes nessas variáveis.*

***Procedimento de inicialização de objetos (Construtor)**: código é executado após a criação de cada novo objeto. Cada objeto terá um número diferente.*

***Método de instância**: só é possível chamá-lo se for através de um objeto.*

***Atributos estáticos**: não é preciso criar objetos para usá-los. Todos os objetos os compartilham.*

***Procedimento de inicialização estático**: código é executado uma única vez, quando a classe é carregada. O arquitecto será um só para todas as casas: ou Zé ou Og.*

Métodos

Métodos, mensagens, funções, procedimentos

- A invocação é sempre efectuada através da notação de pontos.

```
myObj.add(25);
deti.abrePorta();
```

- O receptor da mensagem está sempre à esquerda.
- O receptor é sempre uma classe ou uma referência para um objeto.

Elementos estáticos

- As variáveis estáticas, ou variáveis de classe, são comuns a todos os objetos dessa classe.
- A sua declaração é precedida por static.
- A invocação é feita através do identificador da classe

```
class S {
  public static int a=23;
  public static void class_function() { … }
  // …
}

S.class_function();  // invocada sobre a classe
S s1 = new S();
S s2 = new S();
System.out.println(S.a);
S.a++; // s1.a e s2.a será 24
```

Espaço de Nomes - Package

- Em Java a gestão do espaço de nomes é efectuado através do conceito de package.

- Porque gestão de espaço de nomes?

→Evita conflitos de nomes de classes

  - Não temos geralmente problemas em distinguir os nomes das classes que construímos.
  - Mas como garantimos que a nossa classe Book não colide com outra que eventualmente possa já existir?

Package e import

- Utilização
  - As classes são referenciadas através dos seus nomes absolutos ou utilizando a primitiva import.

```
import java.util.ArrayList
import java.util.*
```

  - A cláusula *import* deve aparecer sempre na primeira linha de um programa.
- Quando escrevemos,

```
import java.util.*;
```

  estamos a indicar um caminho para um pacote de classes permitindo usá-las através de nomes simples:

```
ArrayList al = new ArrayList();
```

- De outra forma teríamos de escrever:

```
java.util.ArrayList al = new java.util.ArrayList();
```

Criar um package

- Na primeira linha de código:

```
package p3;
```

  - garante que a classe pública dessa unidade de compilação (Sock por exemplo) fará parte do package p3.
- O espaço de nomes é baseado numa estrutura de sub-directórios
  - Este package vai corresponder a uma entrada de directório:

```
$CLASSPATH/p3
```

  - Boa prática usar DNS invertido

```
pt.ua.deti.p3
```

- A sua utilização será na forma:

```
p3.Sock sr = new p3.Sock();
```

  - ou

```
import p3.*
Sock sr = new Sock();
```

Características de uma Aplicação

- Qualquer aplicação é uma classe
- Tem um método estático designado por **main**

```
// BomDia.java
import java.util.Date;

public class BomDia {
   public static void main(String args[]) {
      System.out.print("Hello, current date is ");
      System.out.println(new Date());
   }
}
```

- Este ficheiro deve ser gravado com o nome da sua classe pública com a extensão **.java** (**BomDia.java**)
- A compilação vai gerar ficheiros .class - um por cada classe constante no programa.

I/O - Leitura

- Classe Scanner

```
Scanner in = new Scanner(System.in);
String name = in.nextLine();
```

- System.in

```
BufferedReader in =
   new BufferedReader(
       new InputStreamReader(System.in));
String name = in.readLine();
```

I/O - Escrita

```
//..
    System.out.println("Bom Dia!");
//..
```

- System é o nome de uma classe
- out é um objeto estático do tipo PrintStream membro de System
  - System.out
  - System.err
  - System.in
- println é um método estático da classe PrintStream

String

- **String** é uma classe que manipula cadeias de caracteres.
- Não é um array de char

```
String v = "Aveiro";
for (int i=0; i<v.length(); i++)
    System.out.print(v[i]); // ERRO ! usar v.charAt(i)
```

- É imutável

```
v[1] = 'b';  //errado!
```

  - Se precisarmos de strings mutáveis, devemos usar a classe StringBuilder (ou StringBuffer, thread-safe)
- Podemos criar String de várias formas

```
String s2 = "ABCD";
// ou
String s1 = new String("ABCD");  // Não é boa ideia! Porquê?
char[] caracteres = {'A','B','C','D'};
String s3 = new String(caracteres);
```

  - → devemos evitar criar objetos duplicados

String - operações

- Concatenação

```
String s1 = "Programação" + " 3";
```

- Conversão implícita

```
String s1 = "Programação" + 3;
System.out.println(s1+112);
System.out.println(112+s1);
```

- Comparação

```
if  (s1.equals(s2))
  //... true or false

int greater = s1.compareTo("Programação 4");
  // -1 (s1 menor), 0(iguais), 1 (s1 maior)

if  (s1 == s2)
  //... s1 e s2 referenciam o mesmo objecto.
        String s1 = "AAAA";
        String s2 = "AAAA";
        if (s1 == s2) System.out.println("Iguais!!"); //Porquê?
```

toString

- Todos os objetos em Java entendem a mensagem toString()
- Exemplo

```
c1 = new Circulo(0, 0, 5);
System.out.println(c1);  // c1.toString()
```

```
Circle@1afa3
```

- Geralmente é necessário redefinir este método de modo a fornecer um resultado mais adequado.

```
@Override
public String toString() { // Circulo.toString()
   return "Centro: ("+ x + "," + y + ") Raio: " +raio;
}
```

```
Centro: (0,0) Raio: 5
```

Compilar e executar

- J2SE, J2EE

```
javac BomDia.java
java BomDia
```

- IDEs
  - Eclipse
  - NetBeans
  - vim
  - *... e muitos outros*

Comentários

- múltiplas linhas

```
/* Isto é um comentário em
   múltiplas linhas
 */
```

- até ao fim de linha

```
// Isto é um comentário até ao final de linha
```

- javadoc
  - Permite gerir comentários e documentação simultaneamente
  - O compilador de javadoc gera documentos HTML

```
/** Todos os comandos javadoc são delimitados desta forma */
```

Javadoc

- Existem 3 tipos de comentários de documentação

```
/** A class comment */
public class docTest {
  /** A variable comment */
  public int i;
  /** A method comment */
  public void f() {}
}
```

- É possível utilizar HTML dentro destes comentários
  - excepto headings que são inseridos pelo compilador javadoc.
- Outro modo é utilizar "doc tags"

Javadoc - tags

- @see - referência a outras classes ("See Also")

```
@see classname
@see fully-qualified-classname
@see fully-qualified-classname#method-name
```

- Documentação de classes

```
@version
@author
@since
```

- Documentação de métodos

```
@param
@return
@throws
@deprecated
```

JavaDoc - example

```
/**
 * Returns an Image object that can then be painted on the screen.
 * The url argument must specify an absolute {@link URL}. The name
 * argument is a specifier that is relative to the url argument.
 * <p>
 * This method always returns immediately, whether or not the
 * image exists. When this applet attempts to draw the image on
 * the screen, the data will be loaded. The graphics primitives
 * that draw the image will incrementally paint on the screen.
 *
 * @param  url  an absolute URL giving the base location of the image
 * @param  name the location of the image, relative to the url argument
 * @return      the image at the specified URL
 * @see         Image
 */
public Image getImage(URL url, String name) {
        try {
            return getImage(new URL(url, name));
        } catch (MalformedURLException e) {
            return null;
        }
}
```

JavaDoc - result

getImage

```
public Image getImage(URL url,
                      String name)
```

Returns an `Image` object that can then be painted on the screen. The `url` argument must specify an absolute URL. The `name` argument is a specifier that is relative to the `url` argument.

This method always returns immediately, whether or not the image exists. When this applet attempts to draw the image on the screen, the data will be loaded. The graphics primitives that draw the image will incrementally paint on the screen.

Parameters:

`url` - an absolute URL giving the base location of the image.

`name` - the location of the image, relative to the `url` argument.

Returns:

the image at the specified URL.

See Also:

`Image`

Resultado javadoc - métodos

Constructor Summary	
HelloApplet()	

Method Summary	
static void	**main**(java.lang.String[] args)
void	**paint**(java.awt.Graphics g)

Methods inherited from class java.applet.Applet
destroy, getAppletContext, getAppletInfo, getAudioClip, getAudioClip, getCodeBase, getDocumentBase, getImage, getImage, getLocale, getParameter, getParameterInfo, init, isActive, newAudioClip, play, play, resize, resize, setStub, showStatus, start, stop

Estilo de escrita de código

- Classes

  `String, Player, Student, AveiroStudent`

- Métodos e variáveis

  `area, fillArea()`

- Constantes

  `PI, SPEED_LIMIT`

- Estrutura do código

```
class AllTheColorsOfTheRainbow {
    int anIntegerRepresentingColors;
    void changeTheHueOfTheColor(int newHue) {
        // ...
    }
    // ...
}
```

- Bons exemplos:
  - https://google-styleguide.googlecode.com/svn/trunk/javaguide.html

Sumário

- Elementos genéricos da linguagem
- Referências
- Tipos primitivos
- Classes
- Escrita de um programa
- Documentação

Java
Inicialização e Limpeza de Objetos

Programação Insegura

- Muitos dos erros de programação resultam de:
  - dados não inicializados - alguns programas/bibliotecas precisam de inicializar componentes e fazem depender no programador essa tarefa.
  - gestão incorreta de memória dinâmica - "esquecimento" em libertar memória, reserva insuficiente,...
- Para resolver estes dois problemas a linguagem Java utiliza os conceitos de:
  - construtor
  - *garbage collector*

Construtor

- A inicialização de um objeto pode implicar a inicialização simultânea de diversos tipos de dados.
- Uma função membro especial, construtor, é invocada sempre que um objeto é criado.
- A instanciação é feita através do operador new.

```
Carro c1 = new Carro();
```

- O construtor é identificado pelo mesmo nome da classe.
- Este método pode ser overloaded (sobreposto) de modo a permitir diferentes formas de inicialização.

```
Carro c2 = new Carro("Ferrari", "430");
```

Construtor

- Não retorna qualquer valor
- Assume sempre o nome da classe
- Pode ter parâmetros de entrada
- É chamado apenas uma vez: na criação do objeto

```
public class Produto {
    public static int total = 0;
    public int serie = 0;
    public Produto() {
        serie = total + 1;
        total = serie;
    }
}
```

```
public class Livro {
    private String titulo;
    public Livro() {
        titulo = "Sem título";
    }
    public Livro(String umTitulo) {
        titulo = umTitulo;
    }
}
```

Construtor - dummy example

```
// SimpleConstructor.java
// Demonstration of a simple constructor.

class Rock {
  Rock() { // This is the constructor
    System.out.println("Creating Rock");
  }
}

public class SimpleConstructor {
  public static void main(String[] args) {
    for(int i = 0; i < 10; i++)
      new Rock();
  }
}
```

```
Creating Rock
Creating Rock
Creating Rock
Creating Rock
Creating Rock
Creating Rock
Creating Rock
Creating Rock
Creating Rock
Creating Rock
```

Construtor por omissão

- Um construtor sem parâmetros é designado por *default constructor* ou construtor por omissão.
- Este tipo de construtor é automaticamente criado pelo compilador caso não seja especificado nenhum construtor.

```
class Machine {
  int i;
}
Machine m = new Machine(); // ok
```

- No entanto, se houver pelo menos um construtor associado a uma dada classe, o compilador já não cria o de omissão.

```
class Machine {
  int i;
  Machine(int ai) { i= ai; }
}
Machine m = new Machine(); // erro!
```

Questões?

- Qual o valor dos atributos de um objeto quando não foi definido nenhum construtor?

```
class Point
{
  public void display() {…}
  private double x, y;
};

Point p1 = new Point();
```

  - Quais os valores de x e y ?
  - É obrigatório iniciá-los no construtor?

Sobreposição (Overloading)

- Podemos usar o mesmo nome em várias funções, desde tenham argumentos distintos e que conceptualmente executem a mesma ação

```
void sort(int[] a);
void sort(Lista l);
void sort(Set s);
```

- A ligação estática verifica a assinatura da função (nome + argumentos)
- Não é possível distinguir funções pelo valor de retorno
  - porque é permitido invocar, p.e., void f() ou int f() na forma f() em que o valor de retorno não é usado

Construtores sobrepostos

- Permitem diferentes formas de iniciar um objeto de uma dada classe.

```
public class Circulo {
   public Circulo(double x, double y, double r)  {…}
   public Circulo(double r) {…}
   public Circulo() {…}
}

Circulo c1 = new Circulo(12, 43, 2.5);
Circulo c2 = new Circulo(2.5);
Circulo c3 = new Circulo();
```

Sobreposição com base em tipos primitivos

- Alguns tipos primitivos podem ser automaticamente promovidos para outros de maior dimensão
  - podemos encontrar alguns problemas com a sobreposição de métodos.
- Exemplo:

```
void f1(char x) {System.out.println("char");}
void f1(byte x) {System.out.println("byte");}
void f1(short x) {System.out.println("short");}
void f1(int x) {System.out.println("int");}
void f1(long x) {System.out.println("long");}
void f1(float x) {System.out.println("float");}
void f1(double x) {System.out.println("doub");}
```

  - Se não existir a função *f1(tipoN x)* a ligação estática vai promover o *tipoN* ao seguinte maior que esteja definido

A referência this

- A referência this pode ser utilizada internamente a cada objeto de forma a referenciar esse mesmo objeto

```
class Torneira {
  void fecha() { /* ... */ }
  void tranca() { fecha(); /* ou this.fecha() */ }
}

class Circulo {
   double x, y, raio;
   public Circulo(double x, double y, double r)
   {
      this.x = x; this.y = y; raio = r;
   }
}
```

A referência this

- Outra utilização da referência this é para retornar, num dado método, a referência para esse objeto.
  - pode ser usado em cadeia (lvalue).

```
public class Contador {
  int i = 0;
  Contador increment() {
    i++; return this;
  }
  void print() {
    System.out.println("i = " + i);
  }
  public static void main(String[] args) {
    Contador x = new Contador();
    x.increment().increment().increment().print();
  }
}
```

Invocar um construtor dentro de outro

- Quando escrevemos vários construtores podemos chamar um dentro de outro.
  - a referência this permite invocar sobre o mesmo objeto um outro construtor.

```
public Circulo(double x, double y, double r) {
      this.x = x; this.y = y; raio = r;
}
public Circulo(double r) { this(0.0, 0.0, r); }
public Circulo() { this(0.0, 0.0, 1.0); }
```

Duas formas:

- this.método()
- this(..)

- Esta forma só pode ser usada dentro de construtores;
  - neste caso this deve ser a primeira instrução a aparecer;
  - não é possível invocar mais do que um construtor this.

O conceito *static*

- Os métodos estáticos não tem associado a referência *this*.
- Assim, não é possível invocar métodos não estáticos a partir de métodos estáticos.
- É possível invocar um método estático sem que existam objetos dessa classe.
- Os métodos **static** têm a semântica das funções globais (não estão associadas a objetos).

Destrutor

- Mecanismo complementar ao construtor
  - Responsável por desfazer reservas de memória (por exemplo)
- Em algumas linguagens este é um método explícito semelhante ao construtor
  - Em C++ (por exemplo)

```
class String {
public:
    ~String();    // destrutor
    // ...
};

String::~String() { delete [] str; }
```

  - Nesta situação o destrutor é invocado sempre que um objeto é destruído (implícita ou explicitamente)
- Em Java a limpeza de objetos é realizada pelo *garbage collector*

Finalização

- O *garbage collector* é responsável por limpar os objetos não referenciados.
  - Só liberta memória
- Em Java é possível utilizar (não invocar!) uma função, finalize(), que é chamada sempre que o g.c. inicia a destruição do objeto.
- Esta função não funciona 95% das vezes - não deve ser usada!
  - Uma vez que não temos controlo sobre a actuação do g.c. não há garantia que um dado objeto seja libertado antes de o programa terminar.
- Se for necessário finalização, o código deve ser colocado num bloco try {...} finally {...}
- Podemos ter necessidade de construir, numa dada classe, um método explícito de limpeza
  - Connection.close()

Finalização

```
public class Example {
  public synchronized void cleanup() {
    // put your clean up code here, delete files, close connections..
  }
  public void finalize() {
    cleanup();
    super.finalize();
  }
}
// ...
Example example;
try {
  example = new Example();
  //  ... do whatever logic you need to
}
finally {
  example.cleanup();
}
```

Questões?

- Quando é que um objeto pode ser eliminado pelo g.c.?

Inicialização de membros

- A inicialização de variáveis (de tipos primitivos ou não primitivos) pode ser feita na sua declaração.

```
class Measurement {
  boolean b = true; // primitivos
  char c = 'x';
  byte B = 47;
  short s = 0xff;
  int i = 999;
  long l = 1;
  float f = 3.14f;
  double d = 3.14159;
  Superficie s = new Superficie(); // class
  int i = f();    // método
  //. . .
```

Inicialização no Construtor

- A inicialização anterior é comum a todos os objetos da classe
- Outra solução (mais comum) é utilizar o construtor.

```
class Measurement {
    int i;
    char c;
    Measurement(int im, char ch) {
       i = im; c = ch;
 }
```

- Dentro de uma classe a ordem de inicialização é determinada pela ordem das definições das variáveis.
- Mesmo que as variáveis apareçam misturada com os métodos serão sempre inicializadas primeiro.

Exemplo

```
class Tag { Tag(int m) { System.out.println(m);}
}

class Card {
  Tag t1 = new Tag(1); // Before constructor
  Card() { t3 = new Tag(33); // Reinitialize t3
  }
  Tag t2 = new Tag(2); // After constructor
  void f() { System.out.println("f()");
  }
  Tag t3 = new Tag(3); // At end
}

public class OrderOfInitialization {
  public static void main(String[] args) {
    Card t = new Card();
    t.f(); // Shows that construction is done
  }
}
```

1 (1)
4 (Card())
2 (t2)
6 (f())
3 (t3)
0 (main)
5 (t.f())

1
2
3
33
f()

Inicialização de membros estáticos

- Se existir inicialização de membros estáticos esta toma prioridade sobre todas as outras.
- Um membro estático só é inicializado quando a classe é carregada (e só nessa altura)
  - quando for criado o primeiro objeto dessa classe ou quando for usada pela primeira vez.
- Podemos usar um bloco especial - inicializador estático - para agrupar as inicializações de membros estáticos

```
class Circulo {
   static private double lista[ ] = new double[100];
   static {  // inicializador estático
      // inicialização de lista[ ]
   }
}
```

Sumário

- Inicialização e Limpeza
- Construtores
- Garbage collector
- Finalização
- Referência *this*
- Inicialização estática

Java
Encapsulamento

Encapsulamento

- Ideias fundamentais da POO
  - **Encapsulamento (*Information Hiding*)**
  - Herança
  - Polimorfismo

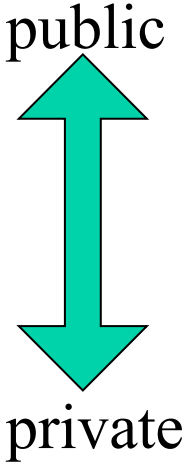
- Encapsulamento
  - Separação entre aquilo que não pode mudar (interface) e o que pode mudar (implementação)
  - **Controlo de visibilidade** da interface (*public*, *default*, *protected*, *private*)

Encapsulamento

- Permite criar diferentes níveis de acesso aos dados e métodos de uma classe.
- Os níveis de controlo de acesso que podemos usar são, do maior para o menor acesso:
  - public - pode ser usado em qualquer classe
  - "omissão" – visível dentro do mesmo package
  - protected – visível dentro do mesmo package e classes derivadas
  - private – apenas visível dentro da classe

Exemplo

```
class X {
  public void pub1( ) { /* . . . */ }
  public void pub2( ) { /* . . . */ }
  private void priv1( ) { /* . . . */ }
  private void priv2( ) { /* . . . */ }
  private int i;
  // . . .
}

class XUser {
  private X myX = new X();
  public void teste() {
     myX.pub1(); // OK
     // myX.priv1();  Errado!
  }
}
```



- Um método de uma classe tem acesso a toda a informação e a todos os métodos dessa classe

Modificadores/Selectores

- O encapsulamento permite esconder os dados internos de um objeto
  - Mas, por vezes é necessário aceder a estes dados diretamente (leitura e/ou escrita).

- **Regras importantes!**

1. Todos os atributos deverão ser privados.

2. O acesso à informação interna de um objeto (parte privada) deve ser efectuada sempre, através de funções da interface pública.

porquê?

Modificadores/Selectores

- Selector
  - Devolve o valor actual de um atributo

```
public float getRadius() { // ou public float radius()
    return radius;
}
```

- Modificador
  - Modifica o estado do objeto

```
public void setRadius(float newRadius) {
                    // ou public void setRadius(float newRadius)
    this.radius = newRadius;
}
```

Métodos privados

- Internamente uma classe pode dispor de diversos métodos privados que só são utilizados internamente por outros métodos da classe.

```
// exemplo de funções auxiliares numa classe
class Screen {
    private int row();
    private int col();
    private int remainingSpace();
    // ...
};
```

O que pode conter uma classe

- A definição de uma classe pode incluir:
  - zero ou mais declarações de **atributos de dados**
  - zero ou mais definições de **métodos**
  - zero ou mais **construtores**
  - zero ou mais **blocos de inicialização static**
  - zero ou mais definições de **classes ou interfaces internas**
- Esses elementos só podem ocorrer **dentro** do bloco '**class NomeDaClasse { ... }**'
  - "tudo pertence" a alguma classe
  - apenas 'import' e 'package' podem ocorrer fora de uma declaração 'class' (ou 'interface')

Métodos - resumo

```
class Point {

    public Point() {…}
    public Point(double x, double y) {…}

    public void set (double newX, double newY) {…}
    public void move (double deltaX, double deltaY) {…}

    public double getX() {…}
    public double getY() {…}
    public double DistanceTo(Point p) {…}
    public void Display() {…}

    private double x;
    private double y;

}
```

Questões?

- No caso particular da classe Point será esta uma boa implementação?

```
class Point {

    public Point() {…}
    public Point(double x, double y) {…}

    public void set (double newX, double newY) {…}
    public void move (double deltaX, double deltaY) {…}

    public double getX() {…}
    public double getY() {…}
    public double distanceTo(Point p) {…}
    public void display() {…}

    private double x;
    private double y;

}
```

Boas práticas

- A semântica de construção de um objeto deve fazer sentido

```
Pessoa p = new Pessoa();   ☹
Pessoa p = new Pessoa("António Nunes"); 😐
Pessoa p = new Pessoa("António Nunes", 12244, dataNasc); ☺
```

- Devemos dar o mínimo de visibilidade pública no acesso a um objeto
  - Apenas a que for estritamente necessária
- Por vezes, faz mais sentido criar um novo objeto do que mudar os atributos existentes

```
Point p1 = new Point(2,3);
p1.set(4,5); ☹
```

Boas práticas

- Juntar membros do mesmo tipo
  - Não misturar métodos estáticos com métodos de instância
- Declarar as variáveis antes ou depois dos métodos
  - Não misturar métodos, construtores e variáveis
- Manter os construtores juntos, de preferência no início
- Se for necessário definir blocos static, definir apenas um no início ou no final da classe.

- A ordem dos membros não é importante, mas seguir convenções melhora a legibilidade do código

Sumário

- Encapsulamento
- Níveis de visibilidade
- Métodos
  - Modificadores
  - Selectores
  - Privados