

Programação
Programação III

Programação Orientada a Objectos – Objectos e Classes

Marco Veloso
marco.veloso@estgoh.ipc.pt

Agenda

Programação Orientada a Objectos – Objectos e Classes

Introdução à Linguagem Java

Paradigmas de Programação

Linguagem Java

Programação Orientada a Objectos

Objectos

Classes

Heranças

Polimorfismo

Tratamento de Excepções

Estruturas de dados

Tabelas unidimensionais

Tabelas multidimensionais

Vectores

Dicionários (Hashtables)

Collections

Ficheiros

Manipulação do sistema de ficheiros

Ficheiros de Texto

Ficheiros Binários

Ficheiros de Objectos

Leitura de dados do dispositivo de entrada

Objectos

Classes

Métodos estáticos

Passagem de parâmetros

Classes Especificas

Objectos

A **programação orientada a objectos (POO)** baseia-se na **construção de programas a partir de objectos**

Desenhar um programa consiste em **definir os objectos necessários, as suas funcionalidades e o modo como comunicam entre si**, por forma a atingir os objectivos pretendidos

Um objecto é uma combinação de **dados** (variáveis) e **acções** (métodos) intimamente relacionados

Um objecto num programa funciona de modo semelhante a um objecto no mundo real

Tem três partes: **identidade**, **atributos** e **comportamento**

É possível (e comum) ter **objectos semelhantes**, com o **mesmo comportamento**, mas com **atributos e identidade diferentes**

Em Java, como em qualquer linguagem OO, **não é possível definir objectos directamente**

É necessário **definir primeiro a classe** a que o objecto pertence

Uma **classe é usada para definir um objecto** (ou um conjunto de objectos semelhantes), especificando a sua **estrutura e comportamento**

Funciona como um molde que pode ser utilizado para criar qualquer número de objectos semelhantes

A definição de uma classe implica a especificação dos seus **atributos (variáveis)** e do seu **comportamento (métodos)**

Um **objecto** é, assim, definido como **instância** de uma classe e tem todas as variáveis e métodos definidos para essa classe, **atribuindo valores concretos (dados)** às variáveis

A diferença entre uma classe e um objecto dessa classe é semelhante à diferença entre uma espécie (ex: `Cao`) e um elemento dessa espécie (ex: `Bob`)

Por convenção, em Java os **nomes de classes começam com maiúscula** (ex: `EstaClasse`), e os **nomes de objectos começam com minúsculas** (ex: `esteObjecto`)

Objectos

- Substantivos
- Coisas reais

Atributos

- **Propriedades** que o objecto tem

Métodos

- **Acções** que o objecto pode fazer

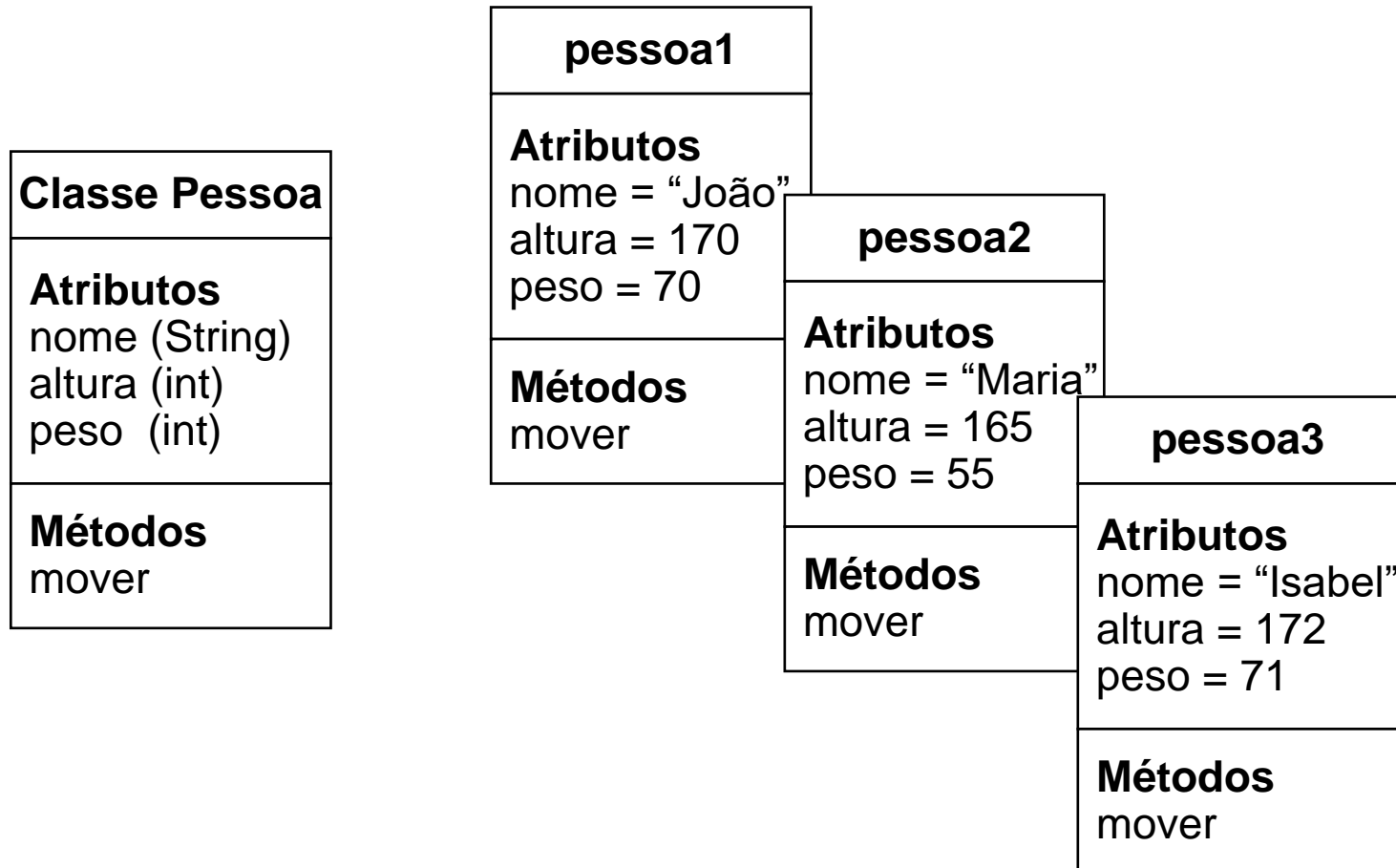
Mensagens

- Comunicação entre objectos
- Um objecto pode pedir a outro para **executar um dado método**

Estrutura de Classes e Objectos

Programação Orientada a Objectos – Objectos e Classes

É possível (e comum) **criar (instanciar) vários objectos a partir de uma mesma classe**



A criação de objectos é conseguida à custa do operador `new`

O operador `new` é seguido do nome da classe a partir da qual se quer criar o objecto e de parêntesis `()` que invoca o respectivo **construtor da classe**. Se a criação necessitar de parâmetros os valores aparecem entre os parêntesis.

Exemplo: `new Pessoa ("Maria", 165, 55);`

Os objectos são criados na memória central, onde reservam o espaço necessário

Quando um objecto é criado é reservado um bloco de memória suficiente para armazenar todos as variáveis do objecto (`nome`, `altura` e `peso`, no nosso exemplo)

Este bloco de memória ficará ocupado até que o objecto seja destruído

A localização de um objecto em memória não é controlada pelo programador

Para interactuar com um objecto é necessário ter alguma maneira de o referenciar. Isto consegue-se à custa de uma **referência**

Uma referência é um tipo especial de valor que identifica um objecto

As referências **podem ser armazenadas em variáveis**, por exemplo:

```
Pessoa atleta; //declaração  
atleta = new Pessoa ("Maria", 165, 55); //inicialização
```

Instanciação de Objectos

Programação Orientada a Objectos – Objectos e Classes

Após a sua declaração, sabe-se que a variável `atleta` referencia objectos da classe `Pessoa`, mas o seu valor é indefinido

A variável só passa a ter um **valor definido** após a **utilização de `new`** (quando o **construtor** é invocado para **inicializar o objecto**), por isso é comum usar-se apenas:

```
Pessoa atleta = new Pessoa ("Maria", 165, 55);
```

A variável `atleta` passa a **armazenar a referência do objecto criado**

Podemos interactuar com o objecto através desta variável

Instanciação de Objectos

Programação Orientada a Objectos – Objectos e Classes

atleta1

AA01	NULL
...	
BB33	NULL
...	
CC10	
CC11	
CC12	
...	
EE55	
EE56	
EE57	

atleta2

As seguintes instruções **apenas declaram duas referências** (armazenadas em variáveis) para objectos, mas não instanciam os dados

Dessa forma, a referência contém **valores nulos**, não podendo ser ainda manipuladas

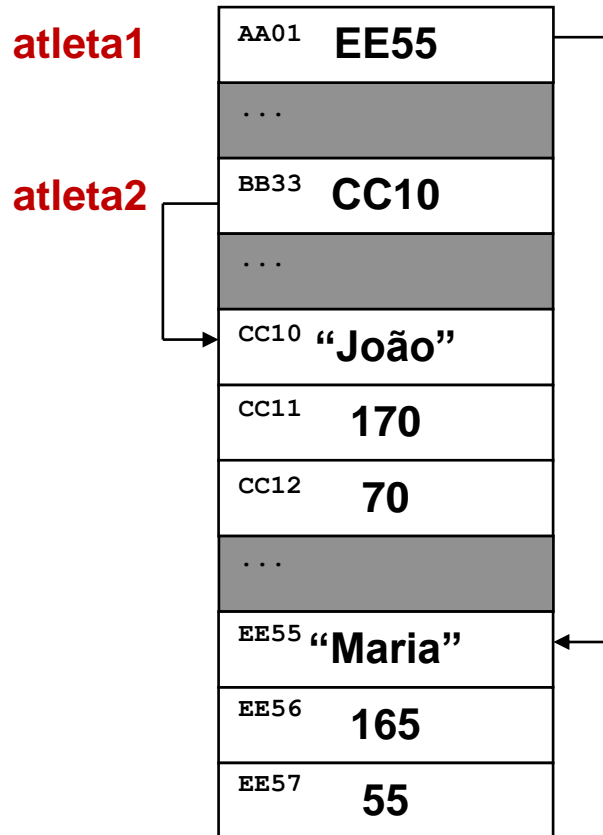
```
Pessoa atleta1;
```

```
Pessoa atleta2;
```

Antes de ser manipulado um objecto tem que ser inicializado (instanciado)

Instanciação de Objectos

Programação Orientada a Objectos – Objectos e Classes



É importante notar a diferença entre a **referência** a um objecto (indicação do local onde o objecto se encontra) e o **próprio objecto**

Por exemplo,

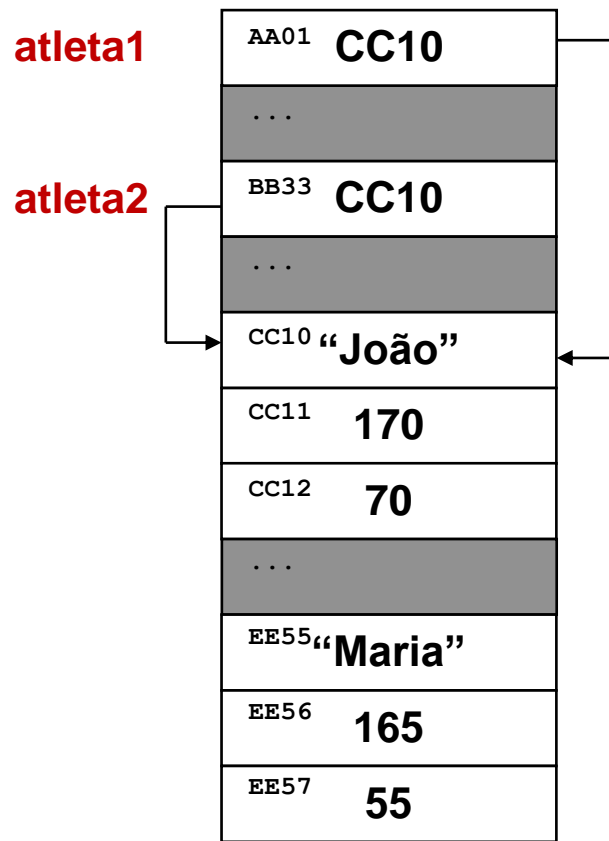
```
Pessoa atleta1 = new Pessoa("Maria",165,55);
```

```
Pessoa atleta2 = new Pessoa("João",170,70);
```

pode provocar a figura anexa

Instanciação de Objectos

Programação Orientada a Objectos – Objectos e Classes



Em consequência, a **alteração da referência não provoca alteração no objecto.**

Por exemplo:

```
atleta1 = atleta2;
```

provoca o resultado da figura

Os **objectos permanecem inalteráveis** e a única consequência neste caso é o facto de o **segundo objecto ficar sem referência**, o que provoca a sua destruição

Quando um objecto já não tem qualquer referência válida para si, não pode ser acedido pelo programa

É inútil e, por isso, chamado lixo ou *garbage*

O Java efectua periodicamente recolha de lixo (***garbage collection***), devolvendo a memória ocupada por estes objectos ao sistema, de modo a que possa voltar a ser utilizada

Noutras linguagens é o programador que tem que se preocupar em fazer a *garbage collection* (por exemplo, na linguagem C o programador é responsável por invocar explicitamente a função `free()` para libertar a memória reservada)

Instanciação de Objectos e Variáveis primitivas

Programação Orientada a Objectos – Objectos e Classes

atleta

AA01	CC10
...	
...	
...	
CC10	"João"
CC11	170
CC12	70
...	
EE55	10
EE56	
EE57	

soma

Os **objectos** possuem assim um espaço para a **referência** e outro espaço para os **dados**. **Igualar ou copiar objectos implica igualar ou copiar referências de memória**, não os dados.

As **variáveis primitivas** reservam apenas espaço para os **dados**. Logo, copiar ou igualar objectos implica **copiar directamente os dados**.

Por exemplo:

```
Pessoa atleta = new Pessoa("João",170,70);  
int soma = 10;
```

Instanciação de variáveis primitivas

Programação Orientada a Objectos – Objectos e Classes

a	AA01	5
	...	
	...	
b	CC10	10
	CC11	
	CC12	
	...	
soma	EE55	5
	EE56	
	EE57	

//iteração 1:

```
int a = 5;
```

```
int b = 10;
```

```
int soma = a;
```

a	AA01	20
	...	
	...	
b	CC10	40
	CC11	
	CC12	
	...	
soma	EE55	5
	EE56	
	EE57	

//iteração 2:

```
a = 20;
```

```
b = 40;
```

Como já foi referido, a definição de uma **classe** implica a especificação dos seus **atributos** (variáveis) e do seu **comportamento** (métodos)

Cada objecto criado a partir de uma dada classe **tem também esses atributos e comportamentos**

Para conseguir que um objecto mostre um dado comportamento (execute um dado método) é necessário **enviar-lhe uma mensagem**

Por exemplo, temos usado `System.out.println()` para escrever no ecrã.

O que acontece é que o Java tem pré-definida uma classe chamada **PrintStream** que representa o monitor

System.out é uma **referência a um objecto dessa classe**

Para que se produza a escrita é necessário **enviar uma mensagem a este objecto** (**println** ou **print**)

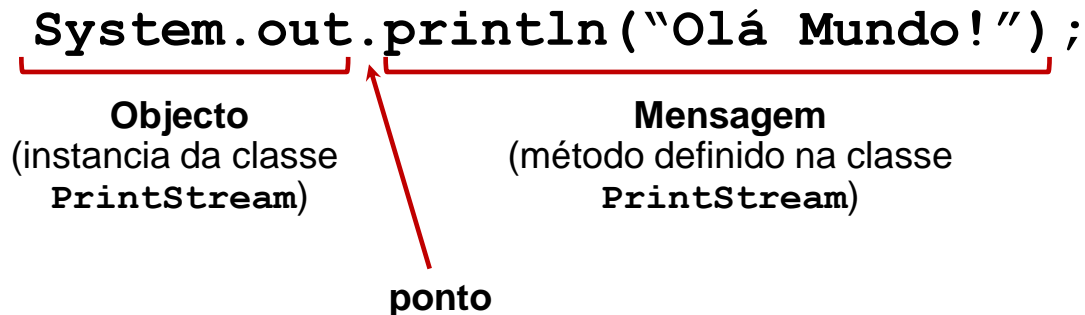
Estas mensagens necessitam de receber a frase a ser escrita

Uma **mensagem** em Java **é um pedido que se faz a um objecto para que apresente um dado comportamento** (o objecto terá que ser instância de uma classe com esse comportamento definido)

Assim, para enviar uma mensagem a um objecto é necessária uma instrução que consiste em:

- Uma **referência ao objecto receptor** (ex: `System.out`)
- Um **ponto**
- A **mensagem** que se pretende mandar (ex: `println`)

Exemplo:



Classes

Estrutura de uma classe

Para além de utilizar **classes pré-definidas**, num projecto é normal que seja necessário definir novas classes que ajudem a atingir o fim pretendido

A sintaxe para definir uma classe é a seguinte:

```
class NomeDaClasse {  
    declarações de atributos  
    (...)  
    construtores  
    (...)  
    métodos  
}
```

As **variáveis**, **construtores** e **métodos** de uma classe são genericamente chamados **membros** dessa classe

Objectivo de uma classe

Programação Orientada a Objectos – Objectos e Classes

Considere-se que se pretendia um programa que leia os dados de um conjunto de **estudantes** (nome e um conjunto de notas), calcule a sua média e ordene os estudantes por ordem decrescente das médias

O primeiro passo será **definir uma nova classe**, a classe **Estudante**, para representar um aluno:

- que terá como **atributos** o nome, as notas e a média
- e como **comportamentos** (ou seja, as acções possíveis) a inicialização, o cálculo da média, o acesso à média e a escrita dos seus dados

Estrutura de uma classe: exemplo

Programação Orientada a Objectos – Objectos e Classes

```
class Estudante {  
    //Atributos  
  
    private String nome;  
    private int [] notas;  
    private float media;  
  
    // Construtor da classe, promove a inicialização dos atributos  
    public Estudante() {  
        (...)  
    }  
  
    //Comportamentos (métodos)  
    public void calculaMedia() { (...)  
        (...)  
    }  
}
```

Os **construtores** definem-se por:

- Cada classe tem pelo menos **um construtor**
- É um tipo especial de método utilizado na **criação de objectos da classe**
- Muitas vezes são usados para **inicializar as variáveis** (atributos)
- Têm o **mesmo nome da classe**
- **Não têm valor de retorno**, nem mesmo `void` (retornam um objecto da classe)
- Não podem ser invocados como os restantes métodos

Os construtores são **invocados pelo operador new**

Assim,

```
String palavra = new String ("Olá");
```

é uma invocação ao **construtor da classe String** que recebe um objecto do tipo `String` como argumento

A criação de um novo objecto da classe `Estudante` pode ser conseguida por:

```
Estudante novoEstudante = new Estudante();
```

Por defeito, existe um construtor que não recebe argumentos (designado '*construtor vazio*').

No entanto, **caso seja criado um**, ou mais construtores explícitos, **este construtor por defeito deixa de existir**

Exemplo de construtores

Programação Orientada a Objectos – Objectos e Classes

O construtor da classe **Estudante** poderá ser:

```
public Estudante () {  
    //A interacção com o utilizador deveria ser realizada noutro ponto  
    //da aplicação, e.g.: Estudante (String, int[], float )  
    System.out.print("Nome do estudante: ");  
    nome = LeituraDados.leituraString();  
  
    System.out.print("Quantas notas? ");  
    int numNotas = LeituraDados.leituraInt();  
  
    //Cria a tabela notas com o número de elementos necessário  
    notas = new int[numNotas];  
    for (int i=0;i<numNotas;i++) {  
        System.out.print("Nota "+(i+1)+" deste aluno: ");  
        notas[i] = LeituraDados.leituraInt();  
    }  
    media = calculaMedia();  
}
```

De notar que o exemplo anterior não é o mais adequado, uma vez que para boa gestão de código, **os construtores não devem possuir interacção com o utilizador** (seja para ler ou mostrar dados). Como regra, apenas o *View* (do modelo MVC) deve interagir com o utilizador.

Os **comportamentos** (métodos) definidos podem ser implementados por:

```
//Escreve os dados de um estudante  
public void imprimeEstudante() {  
    System.out.print("As notas de "+nome+" são: ");  
    for (int i=0; i<notas.length; i++) {  
        System.out.print(notas[i]+ " ");  
    }  
    System.out.println();  
    System.out.println("A média é "+media);  
}
```

Exemplo de métodos

Programação Orientada a Objectos – Objectos e Classes

```
//Método para cálculo da média
private float calculaMedia() {
    //Calcula a soma dos elementos
    float soma = 0;
    for (int i = 0; i<notas.length; i++) {
        soma += notas[i];
    }
    //Devolve a média
    return soma / notas.length;
}

//Método de acesso externo à média
public float getMedia() {
    return media;
}
```

Os **atributos** (nome, notas e media neste exemplo) chamam-se **variáveis de instância**, ou ***instance variables*** (pois são instâncias da classe), porque pertencem ao objecto (instância) como um todo e não a um método em particular (contrapondo às **variáveis de classe** analisadas mais à frente)

As declarações de ***instance variables*** começam geralmente com a palavra **private**, significando que são **internas ou privadas do objecto**, não sendo acessíveis directamente do seu exterior

Uma classe **define o tipo de dados** para um objecto, mas não armazena valores

Cada objecto tem o seu único **espaço de dados em memória**

Todos os métodos de uma classe têm **acesso às *instance variables*** dessa classe

Os **métodos de uma classe são partilhados** por todos os objectos dessa classe

De uma perspectiva externa, **um objecto é uma entidade encapsulada**, fornecendo um dado conjunto de serviços

Estes serviços são conseguidos à custa dos **métodos públicos** do objecto

Ao conjunto de serviços fornecidos por um objecto chama-se **interface desse objecto**

Um objecto deve ser auto-contido, ou seja qualquer alteração do seu estado (das suas variáveis) deve ser provocada apenas pelos seus métodos

Um objecto não deve permitir que outro objecto altere o seu estado

O cliente de um objecto **deve poder requisitar os seus serviços**, mas **sem saber como isso será conseguido**

Encapsulamento

Programação Orientada a Objectos – Objectos e Classes

O **encapsulamento** consegue-se à custa da **utilização dos modificadores de visibilidade** disponíveis na linguagem

Um **modificador** é uma palavra reservada que serve para **alterar as características de um elemento do programa**

Já utilizámos o modificador `final` para definir **constantes**

Em Java há **três modificadores de visibilidade**: **public**, **private** e **protected**

Ainda é considerado o tipo **package** ou **friendly**, que representa os elementos sem modificador de visibilidade explícito

Encapsulamento

Programação Orientada a Objectos – Objectos e Classes

Os membros de uma classe que sejam declarados com o modificador **public** podem ser acedidos a partir de qualquer ponto

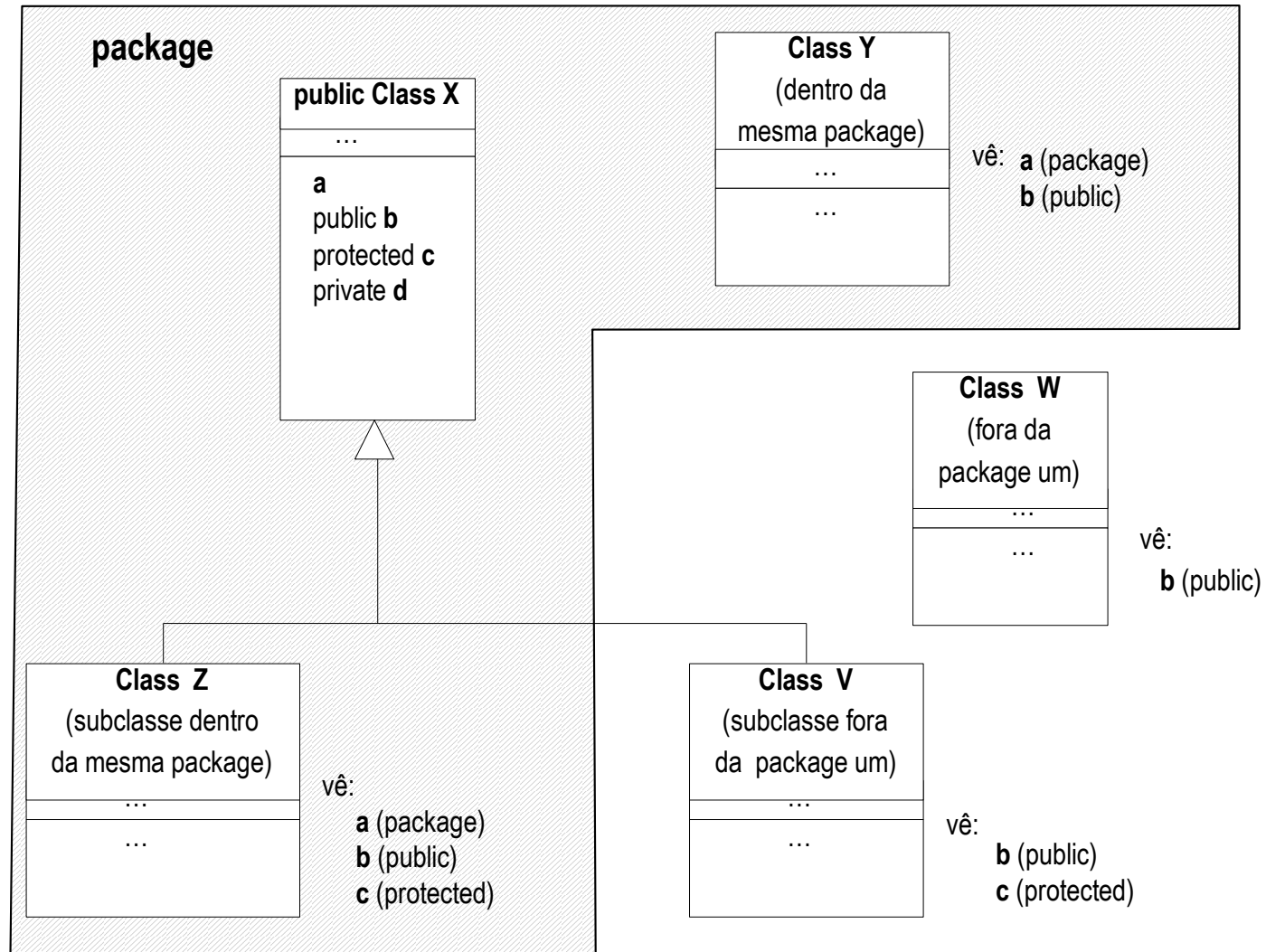
Os membros declarados com **private** só podem ser acedidos de dentro da classe

O modificador **protected** será visto mais tarde, permitindo o acesso dentro da própria classe e sub-classes através do mecanismo de herança

Quando não é definido um modificador explícito, é considerado o tipo **package** ou **friendly**, sendo o valor por defeito, podendo ser acedido a partir da classe e de classes da mesma *package* (uma *package* pode ser vista, de forma simples, como uma directoria ou pasta com classes relacionadas)

Encapsulamento

Programação Orientada a Objectos – Objectos e Classes



Métodos (*getters* e *setters*)

Programação Orientada a Objectos – Objectos e Classes

Os membros declarados **sem qualquer modificador têm visibilidade por defeito** e podem ser acedidas a partir de qualquer classe dentro da mesma *package*

De uma forma geral, **as *instance variables* dos objectos não devem ser declaradas com visibilidade `public`**, antes com **`private`**, devendo ser criados métodos públicos de acesso quando se justifique (designados ***getters***, para aceder ao conteúdo, e ***setters***, para alterar os dados, e.g. `getNomeVar()`, `setNomeVar()`)

Os métodos que implementam os serviços fornecidos pelo objecto são declarados como `public`, por forma a poderem ser chamados a partir de outros objectos

Os métodos públicos chamam-se **serviços**

Os restantes métodos são **métodos de suporte**, servem como auxiliares dos serviços e não devem ser declarados como públicos

Métodos estáticos

Em Java existe o modificador **static** que pode ser aplicado a variáveis ou métodos

Serve para **associar a variável ou método à classe e não a cada objecto** da classe (a cada instancia)

Normalmente **cada objecto tem o seu espaço de dados**

Se uma variável é declarada como **static**, **apenas existe uma cópia para todos os objectos dessa classe**

Isto implica que alterar essa variável num objecto, **provoca a sua alteração também nos restantes objectos dessa classe** (só existe uma variável em memória)

```
private static int conta;
```

Ou seja, uma variável declarada com o modificador **static** **apresenta o mesmo valor em todos os objectos** (é partilhada), e uma **alteração realizado por um objecto a essa variável é visível aos restantes objectos**

Estas variáveis chamam-se também **variáveis de classe**, sendo os métodos denominados por **métodos de classe** (métodos estáticos)

Um **método de classe** (static) caracteriza-se por o seu **comportamento não modificar quando é invocado** (contrariamente aos métodos de instancia que mudam os eu comportamento consoante o objecto sobre o qual são invocados)

Os métodos de classe são assim independentes dos objectos, sendo **invocados através do nome da classe** e não através de objectos

Adicionalmente, **um método de classe não pode aceder a variáveis de instância**, específicas de cada objecto

Elementos *static*

Programação Orientada a Objectos – Objectos e Classes

```
class Estudante {  
    private String nome;           //variável de instância  
    private static int contador = 0; //variável de classe  
  
    Estudante(String aNome) {  
        nome = aNome;  
        contador++;  
    }  
    // método de instância  
    public void setNome(String aNome) { // método setter  
        this.nome = aNome;  
    }  
    // método de classe  
    public static void reiniciaContador() {  
        contador = 0;  
        this.nomeX = "António"  
    }  
}
```

Elementos *static*

Programação Orientada a Objectos – Objectos e Classes

```
class TesteEstudante {  
    public static void main (String[] args) {  
        Estudante est1;  
    }  
}
```

est1

AA01	NULL
...	
BB33	
...	
CC10	
...	
...	
EE77	“0”
...	

contador

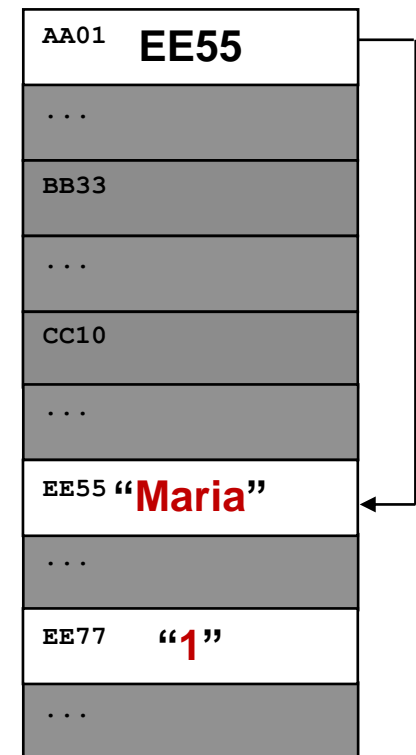
Elementos *static*

Programação Orientada a Objectos – Objectos e Classes

```
class TesteEstudante {  
    public static void main (String[] args) {  
        Estudante est1 =  
            new Estudante ("Maria");  
    }  
}
```

est1

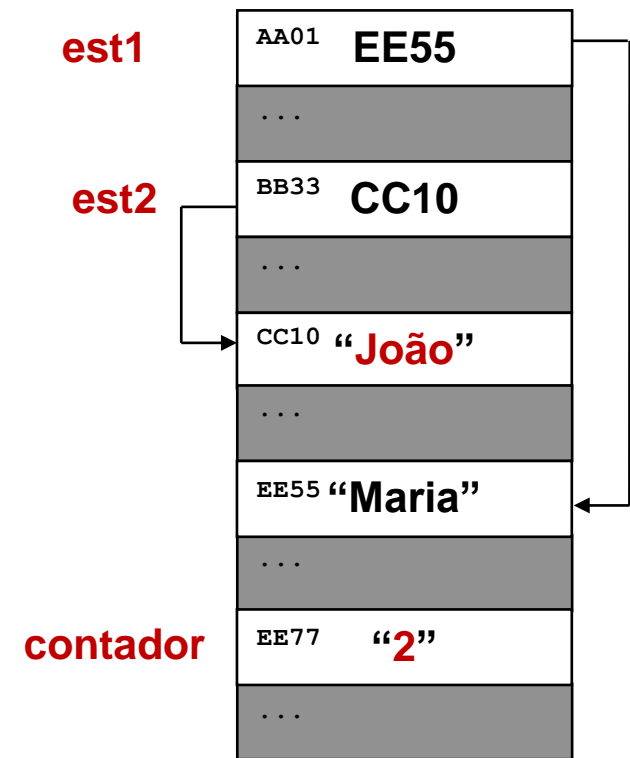
contador



Elementos *static*

Programação Orientada a Objectos – Objectos e Classes

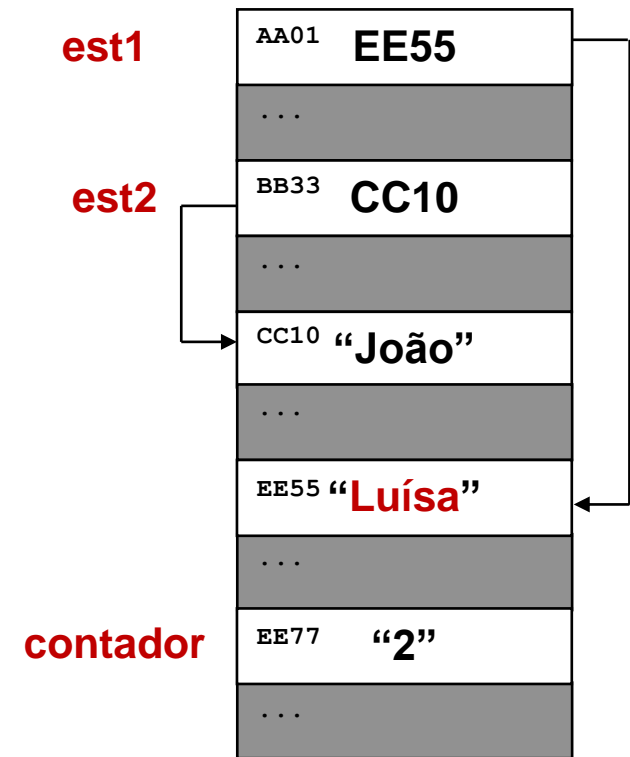
```
class TesteEstudante {  
    public static void main (String[] args) {  
        Estudante est1 =  
            new Estudante ("Maria");  
        Estudante est2 =  
            new Estudante ("João") ;  
    }  
}
```



Elementos *static*

Programação Orientada a Objectos – Objectos e Classes

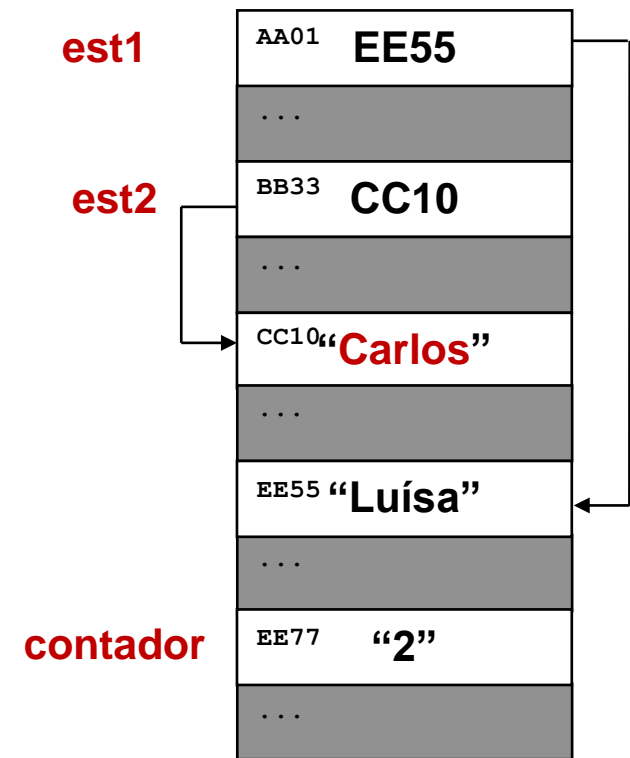
```
class TesteEstudante {  
    public static void main (String[] args) {  
        Estudante est1 =  
            new Estudante ("Maria");  
        Estudante est2 =  
            new Estudante ("João");  
  
        est1.setNome("Luísa") ;  
    }  
}
```



Elementos *static*

Programação Orientada a Objectos – Objectos e Classes

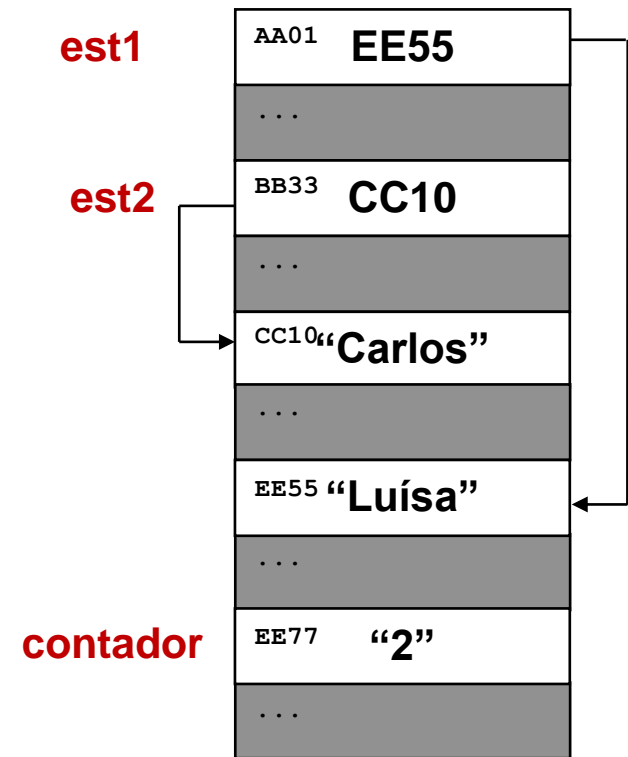
```
class TesteEstudante {  
    public static void main (String[] args) {  
        Estudante est1 =  
            new Estudante ("Maria");  
        Estudante est2 =  
            new Estudante ("João");  
  
        est1.setNome("Luísa");  
  
        est2.setNome("Carlos");  
  
    }  
}
```



Elementos *static*

Programação Orientada a Objectos – Objectos e Classes

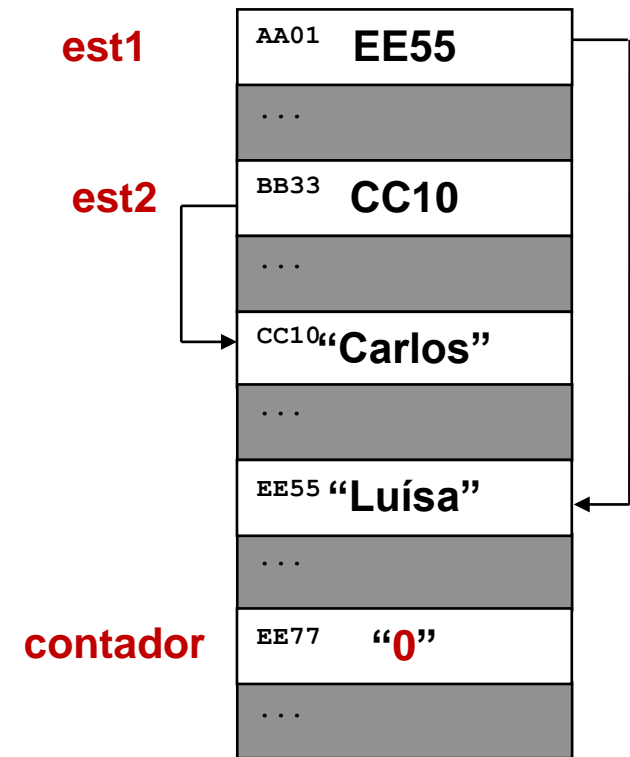
```
class TesteEstudante {  
    public static void main (String[] args) {  
        Estudante est1 =  
            new Estudante ("Maria");  
        Estudante est2 =  
            new Estudante ("João");  
  
        est1.setNome ("Luísa");  
  
        est2.setNome ("Carlos");  
  
        est1.reiniciaContador();  
  
    }  
}
```



Elementos *static*

Programação Orientada a Objectos – Objectos e Classes

```
class TesteEstudante {  
    public static void main (String[] args) {  
        Estudante est1 =  
            new Estudante ("Maria");  
        Estudante est2 =  
            new Estudante ("João");  
  
        est1.setNome ("Luísa");  
  
        est2.setNome ("Carlos");  
  
        est1.reiniciaContador();  
  
        Estudante.reiniciaContador();  
  
    }  
}
```



Normalmente **chamamos um método através de uma instância da classe (objecto)**

Se um **método é declarado como `static` pode (deve) ser chamado através do nome da classe**, não sendo necessário que exista um objecto dessa classe

Por exemplo, a classe `Math` tem vários métodos estáticos que podem ser chamados sem que seja criado um objecto desta classe:

- `Math.abs(num)` - valor absoluto
- `Math.sqrt(num)` - raiz quadrada
- `Math.pow(x, y)` - potência

O método **main** é estático. É chamado a partir do sistema sem que este tenha que criar um objecto

Os métodos estáticos não podem aceder a *instance variables*, uma vez que estas (as variáveis de instância) apenas existem nos objectos

Podem utilizar variáveis declaradas com *static* (variáveis de classe) **e também variáveis locais ao método**

Como referido, estes métodos são normalmente designados por **métodos de classe**

Referencia *this*

A **palavra reservada this é um *handle* (referência) para o próprio objecto (instância) que está a invocar o método.** Esta referência pode ser usada como qualquer outra

Apenas pode ser utilizado dentro de métodos e construtores

Nos construtores toma um **comportamento ligeiramente distinto**, sendo **uma referência para outros construtores da classe da qual o objecto é uma instancia**

Caso queiramos **referenciar o construtor por omissão, a partir de um outro construtor**, basta referenciar `this()`, devendo ser a **primeira instrução a executar no construtor**, antes de qualquer outra inicialização

Assim, o uso da instrução **this** num construtor invoca outro construtor da mesma classe, enquanto o seu uso num método (que não o construtor) referencia o próprio objecto

Um **método static** invocado da forma tradicional

(e.g.: `nomeClasse.nomeMetodo(argumentos)`)


não tem acesso à palavra reservada *this*, excepto quando uma instancia tenha sido associada ao método

(e.g.: `nomeInstancia.nomeMétodo(argumentos)`)

Referencia *this*

Programação Orientada a Objectos – Objectos e Classes

```
class Estudante {  
    //Atributos  
  
    private String nome;  
  
    // Construtor da classe, promove a inicialização dos atributos  
    public Estudante(String nome) {  
        ? nome = nome; ? // desconheço a que corresponde a var nome  
    }  
  
    public Estudante() {  
        this("Ana");  
    }  
    (...)  
}
```



Referencia *this*

Programação Orientada a Objectos – Objectos e Classes

```
class Estudante {  
    //Atributos  
    private String nome;  
  
    // Construtor da classe, promove a inicialização dos atributos  
    public Estudante(String nome) {  
        ? nome = nome; ? // desconheço a que corresponde a var nome  
        this.nome = nome // clara a correspondência da var nome  
    }  
  
    public Estudante() {  
        this("Ana");  
    }  
    (...)  
}
```

Refere-se à variável nome *deste objecto*, ou seja, o objecto que neste instante está a ser invocado

Passagem de parâmetros

Passagem de objectos por argumento

Programação Orientada a Objectos – Objectos e Classes

Em Java sempre que trabalhamos com **objectos** trabalhamos com as **referências** (*references*) para esses objectos.

Uma *referência* é um identificador que usamos para nos referirmos a determinado objecto, não guarda informação sobre o objecto em si, mas sim, sobre o **local em memória onde o objecto reside**.

Assim, uma *referência* para um objecto da classe Integer, **guarda o endereço onde se poderá encontrar esse objecto**.

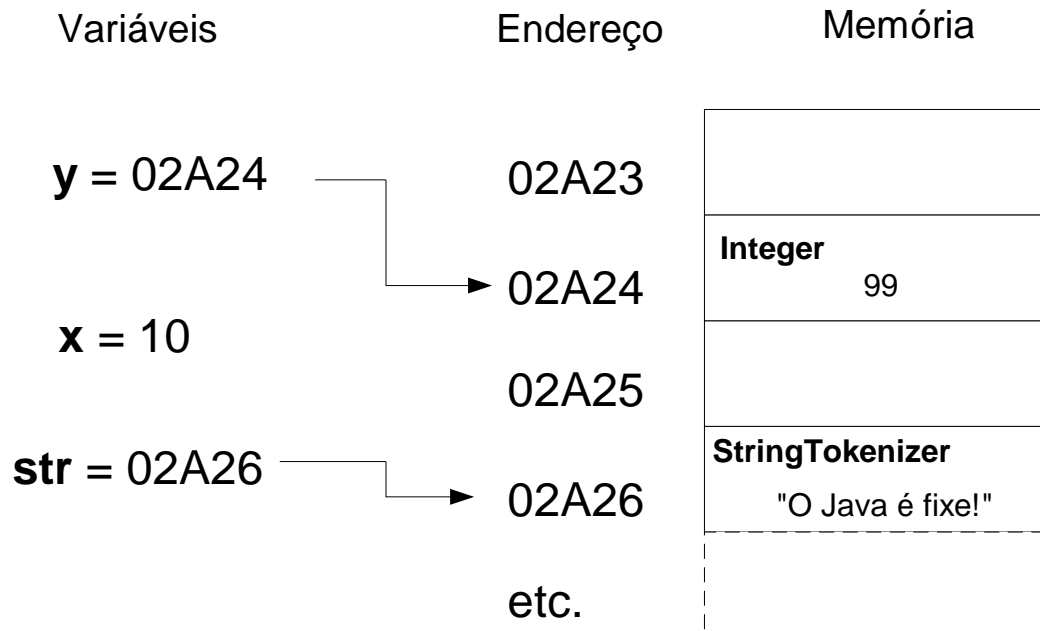
Passagem de objectos por argumento

Programação Orientada a Objectos – Objectos e Classes

Exemplo:

```
int    x    = 10;           // Variável primitiva
Integer y    = new Integer(99); // Objecto
String str = new String("O java é fixe!"); // Objecto
```

Como **y** e **str** são **referências**, ficam com **endereços de memória**, enquanto **x** fica com o valor inteiro **10**:



Passagem de objectos por argumento

Programação Orientada a Objectos – Objectos e Classes

Quando passamos um argumento para um método, na realidade **passamos a referência do objecto para o método**.

Por exemplo, na código seguinte, ao enviar `str` como argumento para um método, não passamos a *string* “*O java é fixe!*”, mas sim o valor 02A26, ou seja, `s` vai ter o valor 02A26 (zona de memória onde está o objecto)

Quer isto dizer que o que o **método recebe** é uma cópia da **referência original**, cópia essa que **vai apontar para o mesmo objecto** que a *referência* original.

Isto quer dizer que **as alterações que se fizerem ao objecto dentro do método, vão-se reflectir cá fora**

Passagem de objectos por argumento

Programação Orientada a Objectos – Objectos e Classes

```
import java.util.*;

class Alias{

    public static void main (String args[]) {
        StringTokenizer str = new StringTokenizer("0 java é fixe!");
        mostraPrimeiraPalavra(str);           //mostra "0"
        mostraPrimeiraPalavra(str);           //mostra "java"
    }

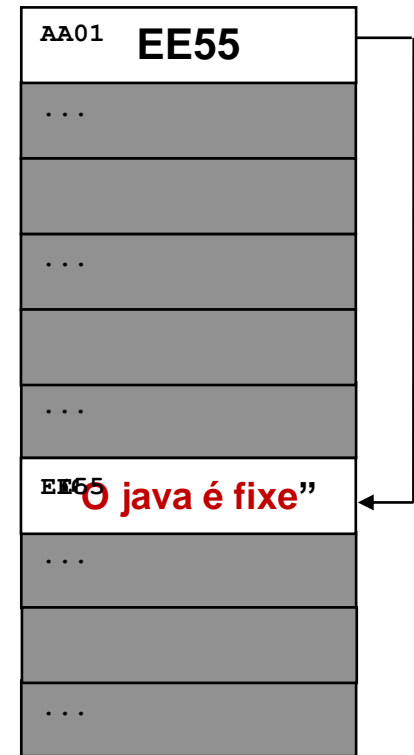
    public static void mostraPrimeiraPalavra(StringTokenizer s) {
        System.out.println(s.nextToken());    //extraí a primeira palavra
    }
}
```

Passagem de objectos por argumento

Programação Orientada a Objectos – Objectos e Classes

```
class Alias{  
    public static void main (String args[]) {  
        StringTokenizer str =  
            new StringTokenizer("O java é fixe!");  
        mostraPrimeiraPalavra(str);  
        mostraPrimeiraPalavra(str);  
    }  
  
    public static void mostraPrimeiraPalavra(StringTokenizer s) {  
        //extrai a primeira palavra  
        System.out.println(s.nextToken());  
    }  
}
```

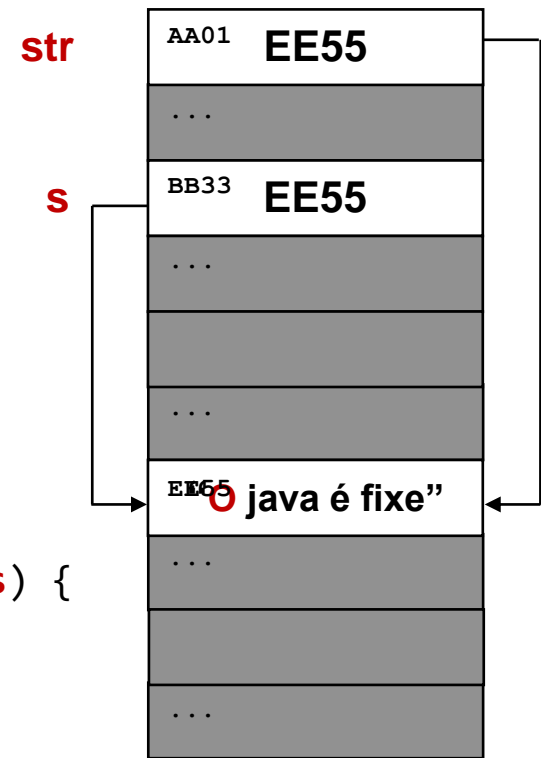
str



Passagem de objectos por argumento

Programação Orientada a Objectos – Objectos e Classes

```
class Alias{  
    public static void main (String args[]) {  
        StringTokenizer str =  
            new StringTokenizer("O java é fixe!");  
        mostraPrimeiraPalavra(str);    //mostra "O"  
        mostraPrimeiraPalavra(str);  
    }  
  
    public static void mostraPrimeiraPalavra(StringTokenizer s) {  
        //extrai a primeira palavra  
        System.out.println(s.nextToken());  
    }  
}
```



Passagem de objectos por argumento

Programação Orientada a Objectos – Objectos e Classes

```
class Alias{  
    public static void main (String args[]) {  
        StringTokenizer str =  
            new StringTokenizer("O java é fixe!");  
        mostraPrimeiraPalavra(str);    //mostra "O"  
        mostraPrimeiraPalavra(str);  
    }  
  
    public static void mostraPrimeiraPalavra(StringTokenizer s) {  
        //extrai a primeira palavra  
        System.out.println(s.nextToken());  
    }  
}
```

str

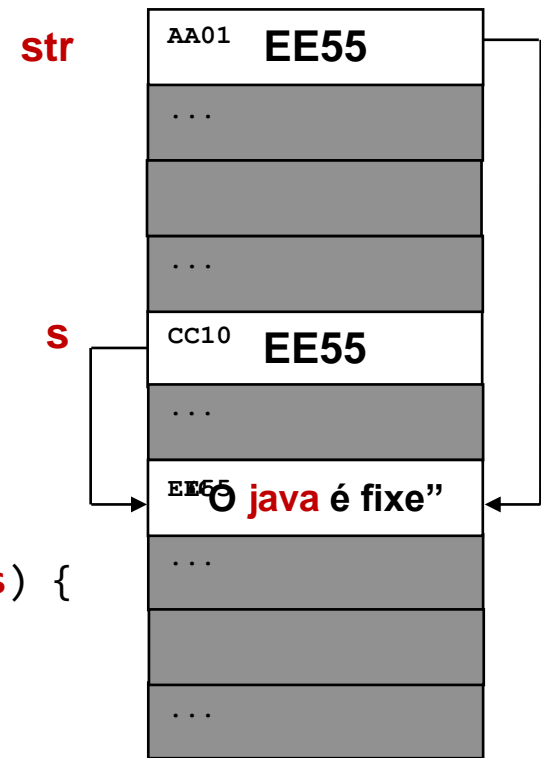


Passagem de objectos por argumento

Programação Orientada a Objectos – Objectos e Classes

```
class Alias{  
    public static void main (String args[]) {  
        StringTokenizer str =  
            new StringTokenizer("O java é fixe!");  
        mostraPrimeiraPalavra(str);    //mostra "O"  
        mostraPrimeiraPalavra(str);    //mostra "Java"  
    }  
}
```

```
public static void mostraPrimeiraPalavra(StringTokenizer s) {  
    //extrai a primeira palavra  
    System.out.println(s.nextToken());  
}  
}
```



Passagem de objectos por argumento

Programação Orientada a Objectos – Objectos e Classes

No entanto, **se dentro do método alterarmos a própria referência**, (isto é se colocarmos a cópia local da referência a apontar para outro sítio) **estas alterações já só vão ser locais ao método**:

```
import java.util.*;
class AlteraRef{
    public static void main (String args[]) {
        StringTokenizer str= new StringTokenizer("O java é fixe!");
        System.out.println(str.nextToken()); //Mostra "O"
        mostraPrimeiraPalavra(str);          //Mostra "Viva"
        mostraPrimeiraPalavra(str);          //Mostra "Viva"
        System.out.println(str.nextToken()); //Mostra "java"
    }

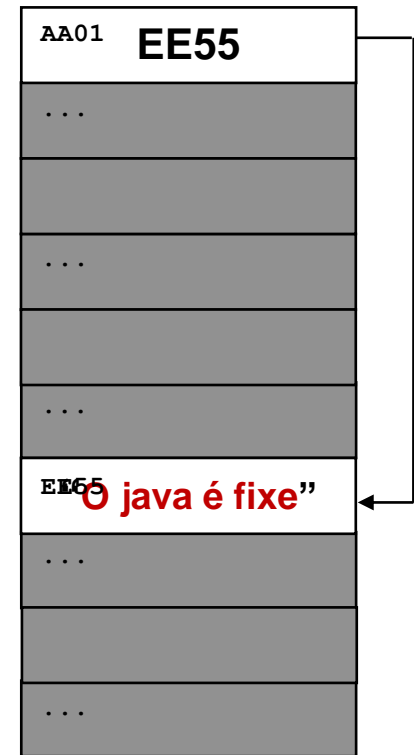
    public static void mostraPrimeiraPalavra(StringTokenizer s) {
        s = new StringTokenizer("Viva o C++!"); //Como redefine s, str não é
                                                alterada!
        System.out.println(s.nextToken());
    }
}
```


Passagem de objectos por argumento

Programação Orientada a Objectos – Objectos e Classes

```
class Alias{  
    public static void main (String args[]) {  
        StringTokenizer str =  
            new StringTokenizer("O java é fixe!");  
        System.out.println(str.nextToken());  
        mostraPrimeiraPalavra(str);  
        mostraPrimeiraPalavra(str);  
        System.out.println(str.nextToken());  
    }  
  
    public static void mostraPrimeiraPalavra(StringTokenizer s) {  
        s = new StringTokenizer("Viva o C++!");  
  
        System.out.println(s.nextToken());  
    }  
}
```

str

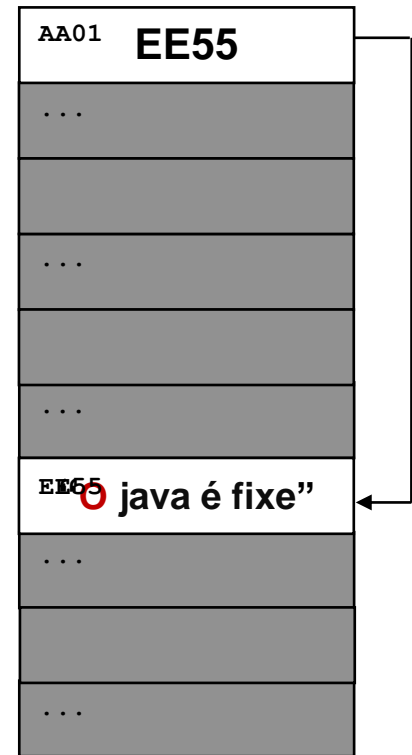


Passagem de objectos por argumento

Programação Orientada a Objectos – Objectos e Classes

```
class Alias{  
    public static void main (String args[]) {  
        StringTokenizer str =  
            new StringTokenizer("O java é fixe!");  
        System.out.println(str.nextToken()); //mostra "O"  
        mostraPrimeiraPalavra(str);  
        mostraPrimeiraPalavra(str);  
        System.out.println(str.nextToken());  
    }  
  
    public static void mostraPrimeiraPalavra(StringTokenizer s) {  
        s = new StringTokenizer("Viva o C++!");  
  
        System.out.println(s.nextToken());  
    }  
}
```

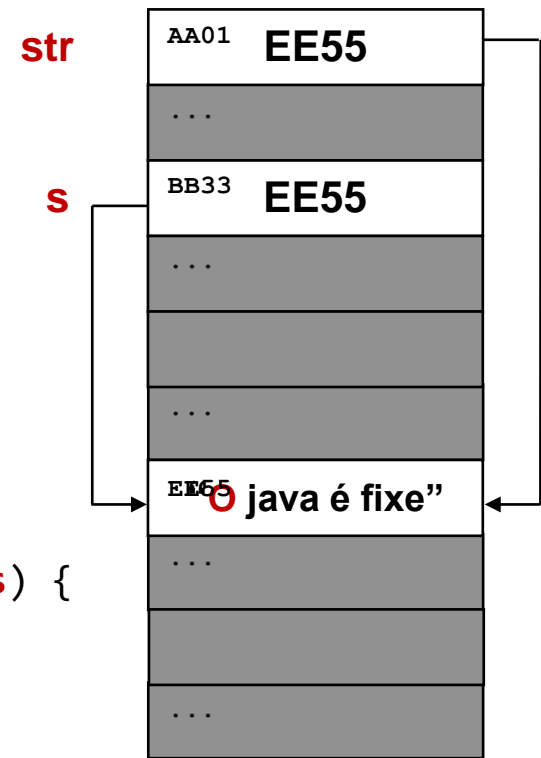
str



Passagem de objectos por argumento

Programação Orientada a Objectos – Objectos e Classes

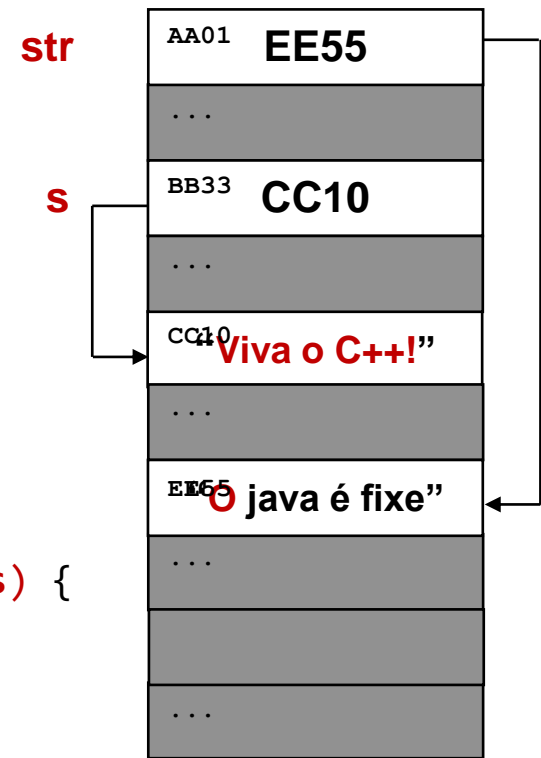
```
class Alias{  
    public static void main (String args[]) {  
        StringTokenizer str =  
            new StringTokenizer("O java é fixe!");  
        System.out.println(str.nextToken()); //mostra "O"  
        mostraPrimeiraPalavra(str);  
        mostraPrimeiraPalavra(str);  
        System.out.println(str.nextToken());  
    }  
  
    public static void mostraPrimeiraPalavra(StringTokenizer s) {  
        s = new StringTokenizer("Viva o C++!");  
  
        System.out.println(s.nextToken());  
    }  
}
```



Passagem de objectos por argumento

Programação Orientada a Objectos – Objectos e Classes

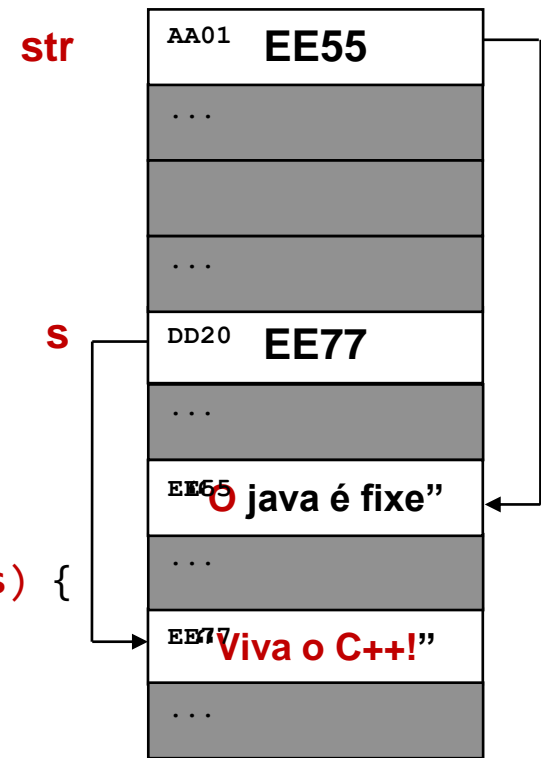
```
class Alias{  
    public static void main (String args[]) {  
        StringTokenizer str =  
            new StringTokenizer("O java é fixe!");  
        System.out.println(str.nextToken()); //mostra "O"  
        mostraPrimeiraPalavra(str);          //mostra "Viva"  
        mostraPrimeiraPalavra(str);  
        System.out.println(str.nextToken());  
    }  
  
    public static void mostraPrimeiraPalavra(StringTokenizer s) {  
        s = new StringTokenizer("Viva o C++!");  
        //Como redefine s, str não é alterada!  
        System.out.println(s.nextToken());  
    }  
}
```



Passagem de objectos por argumento

Programação Orientada a Objectos – Objectos e Classes

```
class Alias{  
    public static void main (String args[]) {  
        StringTokenizer str =  
            new StringTokenizer("O java é fixe!");  
        System.out.println(str.nextToken());    //mostra "O"  
        mostraPrimeiraPalavra(str);             //mostra "Viva"  
        mostraPrimeiraPalavra(str);             //mostra "Viva"  
        System.out.println(str.nextToken());  
    }  
  
    public static void mostraPrimeiraPalavra(StringTokenizer s) {  
        s = new StringTokenizer("Viva o C++!");  
        //Como redefine s, str não é alterada!  
        System.out.println(s.nextToken());  
    }  
}
```

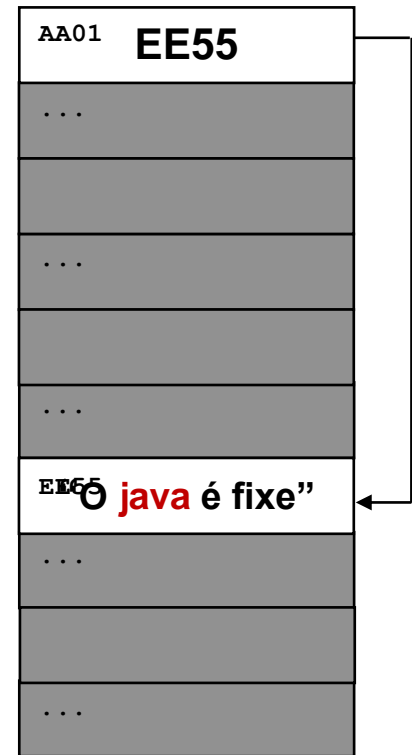


Passagem de objectos por argumento

Programação Orientada a Objectos – Objectos e Classes

```
class Alias{  
    public static void main (String args[]) {  
        StringTokenizer str =  
            new StringTokenizer("O java é fixe!");  
        System.out.println(str.nextToken());    //mostra "O"  
        mostraPrimeiraPalavra(str);             //mostra "Viva"  
        mostraPrimeiraPalavra(str);             //mostra "Viva"  
        System.out.println(str.nextToken());    //mostra "java"  
    }  
  
    public static void mostraPrimeiraPalavra(StringTokenizer s) {  
        s = new StringTokenizer("Viva o C++!");  
  
        System.out.println(s.nextToken());  
    }  
}
```

str



Passagem de argumentos por valor e referência

Programação Orientada a Objectos – Objectos e Classes

É de notar que esta **passagem de argumentos “por referência” só se verifica com objectos**

Quando temos uma **variável de um dos tipos primitivos** (int, boolean, etc.), ou seja, uma variável que não seja um objecto, **o nome da variável guarda o valor da variável em si** (sem *handles*).

Quando se passa uma variável de um tipo primitivo a um método, **passa-se uma cópia do valor** da variável e as alterações que se façam à (nova) variável, dentro do método, **não se reflectem fora do método**

Passagem de argumentos por valor e referência

Programação Orientada a Objectos – Objectos e Classes

Exemplo de passagem de argumentos por **valor** e **referência**:

```
import java.util.*;

class Exemplo{
    public static void main (String args[]){
        int z = 10;                                //Variável primitiva
        String [] str = {"java", "fixe!"};          //Objecto

        System.out.println(str[0]);                 //Mostra "java"
        System.out.println("z="+z);                 //Mostra 10
        mostraPrimeiraPalavra(str, z);
        System.out.println(str[0]);                 //Mostra "Viva" pois o objecto foi alterado
                                                    //no método, sendo as alterações visíveis
        System.out.println("z="+z);                 //Mostra 10 porque as alterações a uma variável
                                                    //primitiva não se reflectem fora do método
    }

    public static void mostraPrimeiraPalavra(String [] s, int z){
        s[0] = "Viva";                               //Não posso reinicializar o objecto: s = new String[2];
        s[1] = "java!";
        z++;                                           //Altera para 11
    }
}
```


Overloading de métodos

Programação Orientada a Objectos – Objectos e Classes

Podemos ter vários **métodos**, dentro da mesma classe, com o **mesmo nome**, mas com **listas de argumentos diferentes**.

Quando se cria um novo significado (comportamento) para um método que já existe, está-se a fazer o que é designado por ***method overloading***.

Por exemplo, assuma que numa dada classe existe um método que permite somar dois inteiros:

```
int    soma(int a, int b)
```

Atendendo a que é possível efectuar a operação de *overloading*, observa-se que qualquer um dos seguintes métodos terá significado e será possível de implementar:

```
int    soma(int a, int b, int c)
```

```
float  soma(float a, float b)
```

Overloading de métodos

Programação Orientada a Objectos – Objectos e Classes

Uma vez que um método é reconhecido pelo seu **nome e tipo dos seus argumentos** (a sua **signature**), então à mesma assinatura podem corresponder várias implementações distintas, **dependendo dos argumentos (tipo e número)** em causa.

Note, no entanto, que efectuar o *overloading* da seguinte forma já **não seria válido** (o tipo de retorno de um método não é considerado)

```
int    soma(int x, int y)
```

```
float soma(int a, int b)
```

Classes Internas

Classes por ficheiro java

Programação Orientada a Objectos – Objectos e Classes

Por definição, **cada classe deve ser armazenado num ficheiro .java com o mesmo nome**. No entanto, **é possível num único ficheiro java incluir várias classes**

Neste cenário, **apenas pode existir uma classe pública num ficheiro java** (aquela que corresponde ao nome do ficheiro)

```
// ficheiro Class1.java
```

```
public class Class1 {
```

```
    /**/
```

```
}
```

```
class Class2 {
```

```
    /**/
```

```
}
```

```
class Class3 {
```

```
    /**/
```

```
}
```

De notar que as **classes não públicas não podem ser acedidas fora do ficheiro**. Ou seja, as classes Class2 e Class3 apenas podem ser acedidas a partir da classe Class1.

É possível **criar classes dentro de classes**, designadas por **classes internas** (ou *inner* ou *nested* classes). Este procedimento permite:

- **Agrupar de forma lógica** funcionalidades próximas ou relacionadas da aplicação/código;
- **Aumentar o encapsulamento**;
- Melhorar a gestão do código

As classes internas apresentam a seguinte estrutura:

```
class ClasseExterna {  
    ...  
    class ClasseInterna {  
        ...  
    }  
}
```

Classes Internas

Programação Orientada a Objectos – Objectos e Classes

```
class Outer {  
    // Simple nested inner class  
    class Inner {  
        public void show() {  
            System.out.println("In a nested class method");  
        }  
    }  
}  
  
class Main {  
    public static void main(String[] args) {  
        Outer.Inner in = new Outer().new Inner();  
        in.show();  
    }  
}
```

Output: In a nested class method

Classes Internas Estáticas

Programação Orientada a Objectos – Objectos e Classes

De notar que **não é possível criar métodos estáticos (*static*) nas classes internas**, uma vez que as classes internas fazem parte da instanciação dos objectos da classe externa.

Pode-se, no entanto, **criar classes internas estáticas**, podendo estas possuir métodos estáticos:

```
class OuterClass {  
    ...  
    static class StaticNestedClass {  
        static void inMethod()  
        ...  
    }  
}  
class InnerClass {  
    ...  
}
```

A criação de objectos da classe interna seguirá a seguinte instrução:

```
OuterClass.StaticNestedClass nestedObject  
    = new OuterClass.StaticNestedClass();
```

Classes Internas Locais (Métodos)

Programação Orientada a Objectos – Objectos e Classes

Para agrupar código, é possível **criar classes dentro de métodos**, garantindo que certas funcionalidades **só podem operar a partir do método**:

```
class Outer {
    void outerMethod() {
        System.out.println("inside outerMethod");
        // Inner class is local to outerMethod()
        class Inner {
            void innerMethod() {
                System.out.println("inside innerMethod");
            }
        }
        Inner in = new Inner();
        in.innerMethod();
    }
}

class MethodDemo {
    public static void main(String[] args) {
        Outer out = new Outer();
        out.outerMethod();
    }
}
```

Output: Inside outerMethod
Inside innerMethod

Classes Anónimas

Programação Orientada a Objectos – Objectos e Classes

Também é possível criar **classes anónimas**, sendo **classes** que não têm **identificação**.

Caracterizam-se por definir no momento a estrutura do objecto criado, podendo ser de dois tipos: **subclasses de um tipo específico** ou **implementação de uma interface específica**. Abordamos aqui o primeiro tipo

```
class Demo {
    void show() {
        System.out.println("i am in show method of super class");
    }
}

class Flavor1Demo {
    // An anonymous class with Demo as base class
    static Demo d = new Demo() {
        void show() {
            super.show();
            System.out.println("i am in Flavor1Demo class");
        }
    };

    public static void main(String[] args){
        d.show();
    }
}
```

Output: i am in show method of super class
i am in Flavor1Demo class

Classes Anónimas

Programação Orientada a Objectos – Objectos e Classes

Classes anónimas como implementação de uma interface:

```
class Flavor2Demo {  
    // An anonymous class that implements Hello interface  
    static Hello h = new Hello() {  
        public void show() {  
            System.out.println("i am in anonymous class");  
        }  
    };  
  
    public static void main(String[] args) {  
        h.show();  
    }  
}  
  
interface Hello {  
    void show();  
}
```

Output: i am in anonymous class

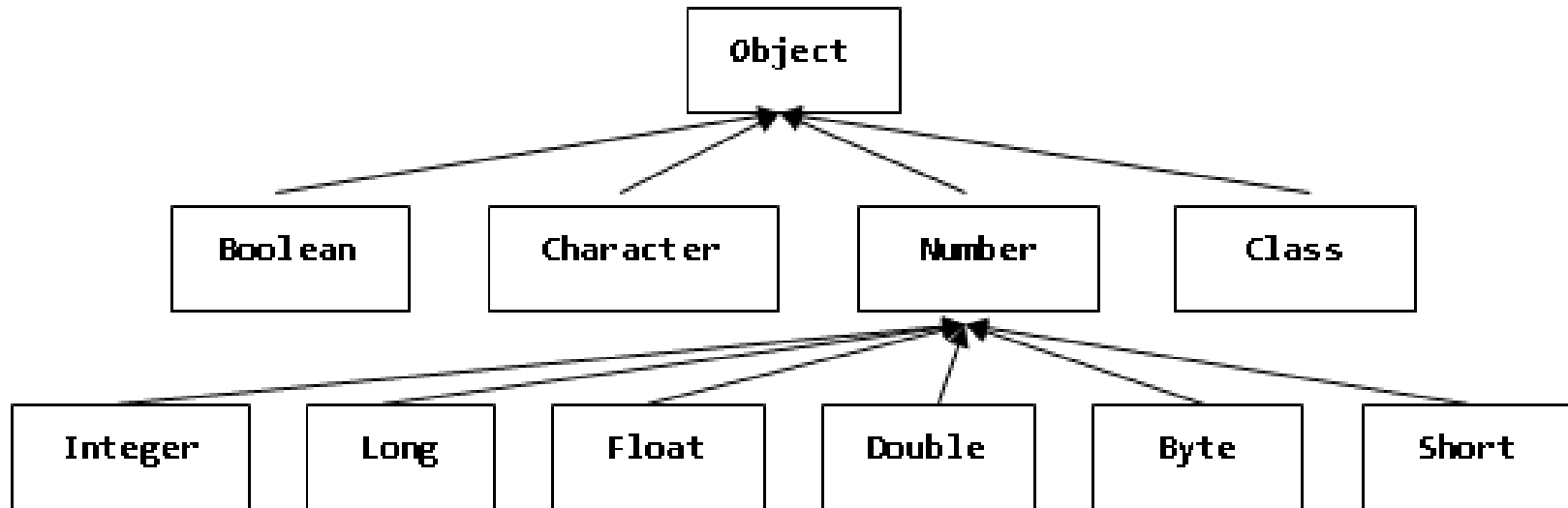
Classes Especificas

Wrapper Classes

Programação Orientada a Objectos – Objectos e Classes

Como se poderá lembrar, existe em java um conjunto de tipos predefinidos: **byte, boolean, char, short, int, float, long e double**.

Cada um dos tipos básicos tem uma **wrapper class**, respectivamente, **Byte, Boolean, Character, Short, Integer, Float, Long e Double**.



```
Integer x = new Integer(4);
```

```
int x = 4;
```

*//cria um **objecto integer** com o valor '4'*

*//cria uma **variável inteira** (não um objecto) com o valor '4'*

As classes **Boolean**, **Character**, **Number** e **Class**, são descendentes directas da classe **Object** e herdam desta classe alguns métodos, nomeadamente

Método	Funcionalidade
<code>clone()</code>	Cria um novo objecto da mesma classe que o objecto original.
<code>equals(Object)</code>	Compara se dois objectos são iguais.
<code>toString()</code>	Devolve uma <i>String</i> com a representação do objecto.

Por sua vez as classes **Byte**, **Short**, **Integer**, **Float**, **Long** e **Double** são descendentes da classe **Number**, o que quer dizer que para além de herdarem os métodos e variáveis membro da classe **Object**, herdam ainda os específicos da classe **Number**, por exemplo:

`byteValue()`, `doubleValue()`, `floatValue()`, `intValue()`, `longValue()`, `shortValue()`, responsáveis pela conversão do **Number** (ou seu descendente) em **byte**, **double**, **float**, **int**, etc.

No topo da hierarquia de classes (*root*) encontra-se a classe **Object** (`java.lang.Object`)

Directa ou indirectamente, **todas as classes herdam da classe Object**

É a sua existência e os respectivos métodos implementados (acções associadas) que permite acções por defeito, como por exemplo comparar dois objectos (método `equals()`) ou imprimir qualquer objecto (método `toString()`)

Método *toString*

Programação Orientada a Objectos – Objectos e Classes

Sempre que se realiza a impressão de um objecto através dos métodos `print()` ou `println()` (implementados na classe `PrintStream` e acedidos através do modificador `out` da classe `System`: e.g. `System.out.print()`) o compilador procura o método **public String toString()** na classe do objecto, pelo que **este método deve ser sempre reescrito** em qualquer classe

Este método deve **devolver uma representação do objecto no formato de uma *String***

Caso não determine o método, irá procurar nas superclasses as respectivas implementações

Em última instância, **usa a implementação do método que se encontra na classe `Object`** (raiz de toda a hierarquia de classes).

Método *toString*

Programação Orientada a Objectos – Objectos e Classes

A implementação na classe object, por defeito, retorna uma *String* com o **nome da classe do objecto** e o **endereço de memória** onde o objecto se encontra, no seguinte formato:

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

Exemplo:

```
Estudante est = new Estudante();
```

```
System.out.println(est);
```

Output:

```
Estudante@659e0bfd
```

Neste exemplo, como a classe **Estudante** não possui uma implementação (*Overriding*) do método **toString()**, é usada a implementação da superclasse **Object**

Método *main*

Programação Orientada a Objectos – Objectos e Classes

Um programa necessita de uma classe que possua um **método main** para execução, com a seguinte assinatura:

```
public static void main (String [] args) {/* Code */}
```

public método público, acessível por classes externas;

static método estático, único, não pode ser instanciado, aplicável apenas a esta classe;

void método não retorna qualquer tipo de valor ou objecto;

(String [] args) argumentos de entrada do método, neste caso concreto o método recebe uma tabela (ou *array*) do tipo *String*. Os argumentos são passados no momento de execução, quando é invocado o comando

The diagram shows a command line: `C:\> java OlaMundo 1 "dados"`. Annotations with dotted lines point to specific parts: 'Comando' points to 'java'; 'Ficheiro com instruções java compiladas para execução' points to 'OlaMundo'; 'Argumento 1' points to '1'; and 'Argumento 2' points to '"dados"'. Each argument is enclosed in a dashed box.

Comando

C:\> java OlaMundo 1 "dados"

Ficheiro com instruções java compiladas para execução

Argumento 1

Argumento 2

Package java.lang

Programação Orientada a Objectos – Objectos e Classes

A *package* (coleção de classes numa pasta) **java.lang** é **importada automaticamente** para todos os programas java

Desta *package* fazem parte classes essenciais ao funcionamento de qualquer programa, nomeadamente:

- Object** Raiz da hierarquia de classes, com métodos por defeito
- System** Controlo de funcionamento, incluindo gestão de entrada e saída de dados
- Math** Operações numéricas básicas (e.g. exponencial, logaritmo, raiz quadrada)
- Thread** Permite um sistema *Multi-threading*
- Exception** Condições de erro que podem ser apanhadas por um *catch*
- String** Representação e manipulação de cadeias de caracteres
- StringBuffer** Uma implementação de *String* que permite modificação em tempo real
- Wrapper class** Byte, Boolean, Character, Short, Integer, Float, Long e Double

Classe *String*

Programação Orientada a Objectos – Objectos e Classes

Método	Funcionalidade
<code>public char charAt (int index)</code>	Devolve o caracter armazenado na posição index.
<code>public int compareTo (String anotherString)</code>	Compara uma string com anotherString. Devolve 0 se forem iguais, um valor negativo se string < anotherString, um valor positivo no caso contrário. ^[1]
<code>public boolean equals (String anotherString)</code>	Compara uma string com anotherString. Devolve <i>true</i> se forem iguais e <i>false</i> se forem distintas. É case sensitive.
<code>public boolean equalsIgnoreCase (String anotherString)</code>	Compara uma string com anotherString. Devolve <i>true</i> se forem iguais e <i>false</i> se forem distintas independentemente de serem maiúsculas ou minúsculas.
<code>public String concat (String str)</code>	Concatena uma string com str. ^[2]
<code>public int indexOf (int ch String ch)</code>	Devolve a posição numa string do caracter ou string ch.
<code>public int length ()</code>	Devolve o número de caracteres de uma string.
<code>public String substring (int beginIndex, int endIndex)</code>	Devolve uma substring de outra string - a partir da posição beginIndex (inclusive) e até à posição endIndex (exclusive)
<code>public String toUpperCase ()</code>	Converte uma string para maiúsculas.

^[1] Se quiser só verificar se duas *strings* (str1 e str2) são iguais, basta fazer `str1.equals(str2)`, que devolve *true* se se verificar a igualdade. Este método é *case sensitive*, verificando se os caracteres estão em maiúscula ou minúscula. Para realizar uma comparação ignorando este aspecto, deve-se recorrer ao método `equalsIgnoreCase(String aString)`, ou seja, `str1.equalsIgnoreCase(str2)`.

^[2] Também pode concatenar com o operador “+”, mas deverá ser cuidadoso na sua utilização.

Classe *StringTokenizer*

Programação Orientada a Objectos – Objectos e Classes

A classe **StringTokenizer** (*package java.util*) que facilita a divisão de uma *String* em partes ou '*tokens*'. Os separadores por defeito são o espaço, '*tab*', '*newline*' e '*carriage return*'

Método	Funcionalidade
<code>public int countTokens ()</code>	Devolve o número de ' <i>tokens</i> ' que existem num <i>stringTokenizer</i> . Este valor decrementa à medida que são retirados os diversos ' <i>tokens</i> '.
<code>public boolean hasMoreElements ()</code>	Devolve <i>true</i> se ainda existirem ' <i>tokens</i> ' num <i>stringTokenizer</i> , <i>false</i> no caso contrário.
<code>public String nextToken ()</code>	Devolve o próximo ' <i>token</i> ' de um <i>stringTokenizer</i> .

```
StringTokenizer tokens = new StringTokenizer (frase);
StringTokenizer tokens = new StringTokenizer (frase, separador);

while (tokens.hasMoreElements()) {
    String newToken = tokens.nextToken();
}
```

Decompor uma *String*

Programação Orientada a Objectos – Objectos e Classes

Em alternativa à classe **StringTokenizer**, é possível recorrer ao método **split(String separador)** da classe **String**, que retorna uma tabela com os vários ‘tokens’ que compõem a *string* original:

```
String[] tokens;  
String frase= "Programa java é divertido!";  
tokens = frase.split(" "); //neste caso, o separador é o espaço  
  
for(int i=0; i< tokens.length; i++) {  
    System.out.println("Token "+i+" = "tokens[i]);  
}
```

Referências

Programação Orientada a Objectos – Objectos e Classes

“Programação Orientada a Objectos”

António José Mendes

Departamento de Engenharia Informática, Universidade de Coimbra

“Java in a Nutshell”, Capítulo 3 “Object-Oriented Programming in Java”

David Flanagan

O'Reilly, ISBN: 0596002831

“Thinking in Java, ”

Capítulo 1 “Introduction to Objects”; Capítulo 2 “Everything is an Object”

Bruce Eckel

Prentice Hall, ISBN: 0131872486

“The Java Tutorial - Learning the Java Language: Object-Oriented Programming Concepts”

Java Sun Microsystems

<http://java.sun.com/docs/books/tutorial/java/concepts/index.html>

“The Java Tutorial – Learning the Java Language: Classes and Objects ”

<http://java.sun.com/docs/books/tutorial/java/javaOO/index.html>

"Fundamentos de Programação em Java 2", Capítulo 8 "*Classes e Objectos*"

António José Mendes, Maria José Marcelino

FCA, ISBN: 9727224237

"Java 5 e Programação por Objectos",

Capítulo 1 "*Paradigma da Programação por Objectos*", Capítulo 3 "*Classes e Instâncias*"

F. Mário Martins

FCA, ISBN: 9727225489