

Classes e objetos

Atributos e métodos

Encapsulamento

Métodos especiais

Conceito de classe

Um programa = coleção de **objetos** que interagem entre si através de **mensagens**.

Cada **objeto** tem um tipo. Cada objeto é instância de uma **classe**. Classe = tipo.

Uma **classe** define os estados possíveis de objeto, o seu comportamento e o relacionamento com outros objetos.

Pode-se criar qualquer número de objetos numa classe. Vários objetos numa classe têm o mesmo comportamento mas guardam dados diferentes.

Todas as classes em Java são **derivadas** (i.e. são **uma espécie de**) da classe **Object** definida na biblioteca **java.lang**.

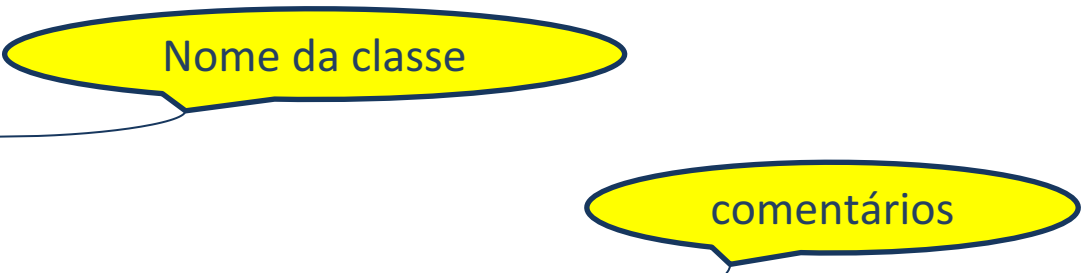
Exemplo:

Imagine que existe uma classe **Pessoa**. Cada **Pessoa** tem nome e data de nascimento. Mas **Pessoas** diferentes têm, em geral, nomes e datas diferentes.

Conceito de classe (cont.)

Definição duma classe (ficheiro `Pessoa.java`):

```
public class Pessoa
{
    //dados que vão compor o objeto
    /*métodos que vão responder às mensagens enviadas ao
    objeto*/
}
```



Java é uma linguagem **case-sensitive**.


O ficheiro que contém código fonte deve ter o nome `Xxx.java`, onde `Xxx` – é o nome da classe principal.

Esta classe deve ser declarada como pública (**public**) de modo a permitir que seja acessível a utilizadores (para que possam criar objetos desta classe).

Dados da classe

Vamos adicionar dois dados (*fields*) à classe **Pessoa**: nome e idade.

```
public class Pessoa
{
    //dados que compõem o objeto
    private String nome; // nome da pessoa
    private int idade;   // idade da pessoa
    /*métodos que vão responder às mensagens enviadas ao
    objeto*/
}
```



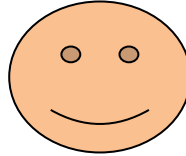
A yellow oval with a blue border contains the text "especificadores de acesso". A blue arrow points from this oval to the `private` keyword in the `private int idade;` line of the code block.

O atributo `nome` é uma **referência** para um objeto do tipo **String**. **String** é uma classe existente nas bibliotecas de Java (**java.lang**) que serve para modelar *strings* (sequências de caracteres).

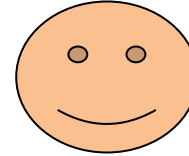
O atributo `idade` é uma variável de **tipo primitivo int** (que especifica um inteiro).

Intervenientes possíveis

Criador da classe

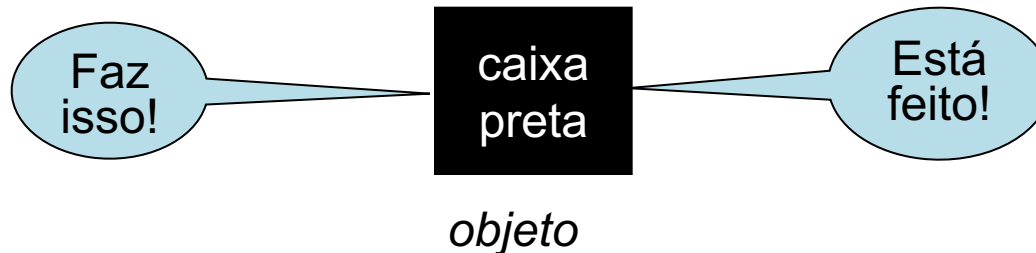


Utilizador da classe



- **Criador da classe** – uma pessoa ou uma equipa que desenvolve classes para serem usadas por outras pessoas/equipas.
- **Utilizador da classe** – pessoa/equipa que utiliza classes já existentes (desenvolvidas por outros) para poupar recursos, tempo, etc.

Em ideal, o cenário de utilização dum objeto deveria ser este:



Especificadores de acesso

Tal cenário é atingido com **especificadores de acesso** que servem para controlar acesso aos membros (atributos e métodos) de uma classe:

- **public** – significa que o membro pode ser acedido por **todos**
- **protected** – será considerado mais tarde
- **private** – significa que ninguém tem acesso para além dos métodos-membros da própria classe.

Os membros de uma classe por omissão (se se esquecer de colocar um especificador de acesso à frente dum membro) são de acesso **package**. Isso significa que o membro pode ser acedido por **todas as outras classes que pertencem ao mesmo pacote** (para já – que estejam definidas com ficheiros na mesma pasta).

Regra geral: deve declarar todos os atributos (membros-dados) como privados (**private**).

Porquê?

Há três razões principais que explicam porque se deve esconder ao máximo a estrutura e implementação duma classe:

- 1) Para garantir que os clientes, por engano, não estraguem algo importante. Se o cliente não tem acesso a um membro – torna-se impossível que mude o seu valor, por exemplo.
- 2) Para tornar claras as partes da classe que servem para o cliente usar e separar as partes que suportam a implementação interna da classe. Logo o cliente consegue facilmente filtrar o que é importante e separar a parte que pode ignorar.
- 3) O criador da classe tem a liberdade completa de mudar a estrutura e implementação interna da classe sem se preocupar que isto afete clientes. A única restrição: não mudar a interface da classe, i.e. mensagens a que a classe deve reagir (métodos públicos).

Métodos da classe

Vamos adicionar métodos que permitem consultar e alterar o nome e idade da pessoa.

```
public class Pessoa
{
    ...
    //métodos
    public String getName() { //lê nome
        return nome;
    }
    public void setName(String nome_novo) { //altera nome
        nome = nome_novo;
    }
    public int getAge() { //lê idade
        return idade;
    }
    public void setAge(int idade_nova) { //altera idade
        idade = idade_nova;
    }
}
```


Especificação de métodos em Java

Especificador de acesso. Já que o método é público pode ser usado por todos.

Tipo de valor que o método retorna.

Nome do método.

Parâmetros que o método recebe.

```
public int getAge() { //lê idade  
    return idade;  
}
```

```
public void setAge(int idade_nova) { //altera idade  
    idade = idade_nova;  
}
```

Implementação (corpo) do método.

Regra geral: deve declarar métodos que definem a interface da classe (que sejam usados por clientes) como públicos (**public**).

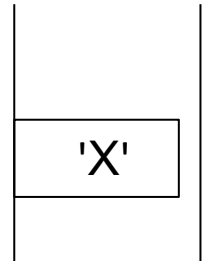
Objetos

Os **objetos** em Java são manipulados através de **referências**.

Para criar um objeto deve-se especificar o seu tipo e fornecer parâmetros necessários à sua construção.

Exemplo: para criar um objeto do tipo **Character**:

```
new Character('X');
```

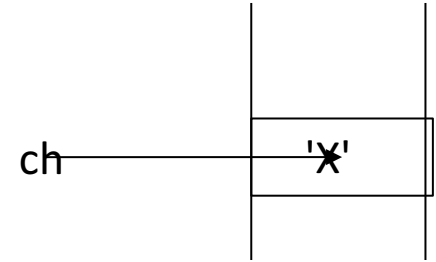


Este código reserva na **memória dinâmica** (*heap*) espaço necessário para armazenar um **Character** e a seguir executa um código especial responsável pela inicialização deste **Character** com o conteúdo fornecido na lista de parâmetros ('X'). Como resultado, será devolvida uma **referência** para o objeto criado.

Mas não a gravámos em lado nenhum. Logo não podemos utilizar o objeto criado ☹.

Objetos (cont.)

Solução:



```
Character ch = new Character('X');
```

Aqui `ch` é uma **referência** para o objeto **Character** criado dinamicamente (com o operador **new**).

Através de `ch` podemos facilmente localizar o nosso **Character** na memória.

Através da referência `ch` podemos agora manipular o objeto, i.e. enviar mensagens para ele.

Por exemplo o método `hashCode` permite determinar o código do **Character**:

```
int code = ch.hashCode(); //o resultado é 88
```

Objetos (cont.)

Tabela **Unicode** – *standard* para codificação e representação de símbolos usados em sistemas de computação:

| | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|-----|----|----|----|----|----|----|
| 0 | | | | | | | | | | | | | | | | | | | | |
| 20 | | | | | | | | | | | | | | ! | " | # | \$ | % | & | ' |
| 40 | (|) | * | + | , | - | . | / | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | : | ; |
| 60 | < | = | > | ? | @ | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| 80 | P | Q | R | S | T | U | V | W | X | Y | Z | [| \ |] | ^ | _ | ` | a | b | c |
| 100 | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w |
| 120 | x | y | z | { | | } | ~ | | € | | , | f | .. | ... | † | ‡ | ^ | %o | Š | ◀ |

Objetos (cont.)

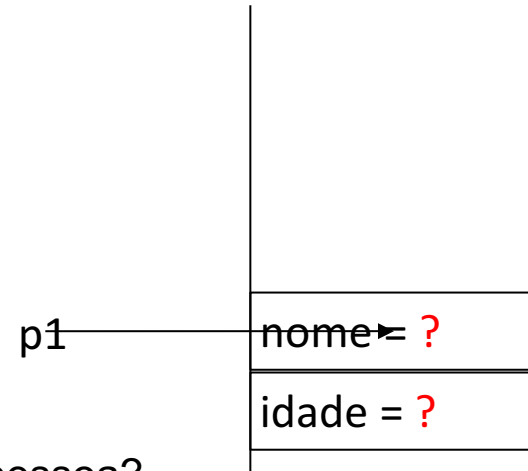
De maneira semelhante pode-se criar um objeto do tipo **Pessoa**:

```
Pessoa p1 = new Pessoa();
```

Aqui `p1` é uma **referência** para uma **Pessoa** criada dinamicamente (com o operador **new**).

Note que a lista de parâmetros está vazia.

```
public class Pessoa
{
    //dados que compõem o objeto
    private String nome;
    private int idade;
    ...
}
```



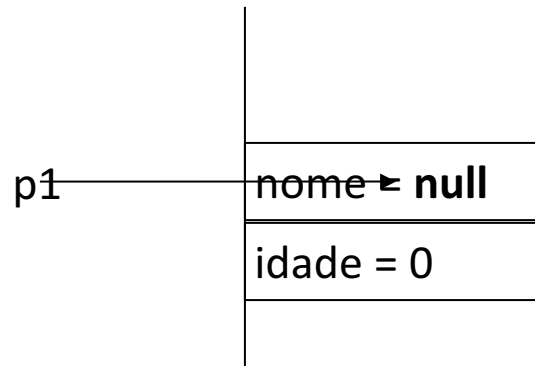
Que valores vão ter os atributos `nome` e `idade` desta pessoa?

Inicialização

Se não implementar nenhum método especial de inicialização (não temos), o compilador Java vai inicializar automaticamente todos os dados da classe com os seus **valores por omissão**.

O campo `nome` é uma referência (que deve referenciar uma **String**). O valor por omissão dum referência é **null** (i.e. não referencia nenhum objeto).

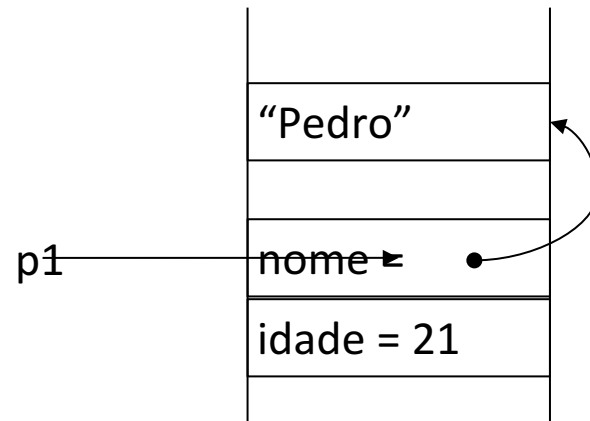
O dado `idade` é um inteiro (**int**). O valor por omissão dum inteiro é 0.



Envio de mensagens

Uma vez construído o objeto do tipo **Pessoa**, podemos enviar mensagens para ele, por exemplo, modificar o seu nome e idade:

```
Pessoa p1 = new Pessoa();  
p1.setName("Pedro");  
p1.setAge(21);
```

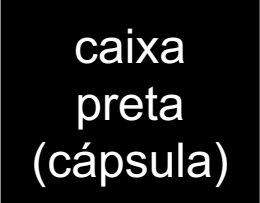


Encapsulamento

Propriedade fundamental da programação orientada a objetos.

Encapsulamento = interligação de dados e métodos dentro de classes em combinação com controlo de acesso.

O resultado é criação de tipos de dados que possuam **caraterísticas** e **comportamentos**.



caixa
preta
(cápsula)

objeto

Atributos estáticos

A palavra-chave **static** permite definir atributos que são **partilhados** por todos os objetos duma classe. No caso de atributos não estáticos, cada objeto tem a sua própria instância de cada atributo.

Para um atributo estático só é criada uma **única instância** deste na memória.

O código de cada classe é compilado para o seu próprio ficheiro *.class que só é carregado no momento da primeira utilização desta classe. Isto ocorre quando criamos um objeto desta classe ou quando se tenta aceder a um atributo estático.

Os atributos estáticos são inicializados pela ordem da sua comparência na definição da classe.

Atributos estáticos podem ser acedidos quer através de objetos quer através do nome da classe (preferencialmente).

Exemplo de atributos estáticos

Um exemplo de atributo estático é **java.lang.Math.PI**. Para aceder o valor de **PI** não é preciso criar nenhum objeto de tipo **Math**:

```
System.out.println(Math.PI);
```

Exemplo:

```
public class StaticTest {  
    public static int i = 55;  
  
    public static void main(String[] args) {  
        System.out.println(StaticTest.i);  
        StaticTest obj1 = new StaticTest();  
        StaticTest obj2 = new StaticTest();  
        StaticTest.i++;  
        System.out.println(StaticTest.i); //obj1.i==obj2.i==56  
    }  
}
```

Métodos estáticos

Métodos estáticos também **não são** associados com qualquer objeto particular e **operam sobre toda a classe**.

Métodos estáticos podem ser invocados mesmo quando não foi criado nenhum objeto.

Métodos estáticos só têm acesso a atributos estáticos.

Exemplo:

```
public class StaticTest {  
    public static int i = 55;  
    private int j;  
    public static void modify ()  
    {  
        i++;  
        j; //erro  
    }  
  
    public static void main(String[] args) {  
        StaticTest.modify();  
        System.out.println(StaticTest.i);  
    }  
}
```

Métodos estáticos da classe **Math**

A classe **Math** inclui métodos estáticos que permitem realizar operações numéricas sem ter criado nenhum objeto de tipo **Math**.

Exemplo:

```
System.out.println(Math.abs(-5.3)); //valor absoluto
System.out.println(Math.cos(Math.PI)); // coseno
System.out.println(Math.log10(1000)); // logaritmo base 10
System.out.println(Math.max(45.6, 46.5)); // valor máximo
System.out.println(Math.pow(2, 10)); // potência
System.out.println(Math.random()); // aleatório [0.0,1.0)
System.out.println(Math.round(5.5)); // valor arredondado
```

Inicialização de objetos com construtores

Construtor é um método especial que é invocado sempre que um objeto é criado.

Se a classe tem um construtor, o compilador de Java invocará este construtor automaticamente para cada objeto criado, garantindo deste modo a inicialização deste objeto.

Construtor deve ter o **mesmo nome** que a classe e não retorna **nada**.

Exemplo:

```
public class Pessoa
{
    private String nome;
    private int idade;

    public Pessoa() //construtor
    {
        System.out.println("A criar uma pessoa");
    }

    public static void main(String[] args)
    {
        for (int i = 0; i < 5; i++)
            new Pessoa();
    }
}
```

```
A criar uma pessoa
A criar uma pessoa
A criar uma pessoa
A criar uma pessoa
A criar uma pessoa
```

Construtor por omissão

Um **construtor sem argumentos** é chamado **construtor por omissão** (*default constructor*).

Exemplo:

```
public class Pessoa
{
    private String nome;
    private int idade;

    public Pessoa() //construtor por omissão
    {
        System.out.println("A criar uma pessoa");
        nome = "Sem nome";
    }

    public static void main(String[] args)
    {
        for (int i = 0; i < 5; i++)
            System.out.println(new Pessoa().nome);
    }
}
```

A criar uma pessoa
Sem nome
A criar uma pessoa
Sem nome
A criar uma pessoa
Sem nome
A criar uma pessoa
Sem nome
A criar uma pessoa
Sem nome

Construtores com argumentos

Um construtor, assim como qualquer outro método, pode ter argumentos que especificam **como** criar um objeto.

Exemplo:

```
//construtor com dois argumentos
public Pessoa(String name, int age)
{
    nome = name;
    idade = age;
}

...
new Pessoa("Pedro Pinto", 19);
```

Sobreposição de nomes de métodos

Sobreposição de nomes permite definir **vários métodos com o mesmo nome** desde que tenham **argumentos diferentes**.

=> A classe pode ter **vários construtores**!

Exemplo:

```
public Pessoa ()
{
    System.out.println("A criar uma pessoa");
    nome = "Sem nome";
}

public Pessoa(String name, int age)
{
    nome = name;
    idade = age;
}

public Pessoa(int age)
{
    nome = "Sem nome";
    idade = age;
}
```


Sobreposição de nomes de métodos

Note que é **impossível** definir vários métodos com o mesmo nome que tenham a mesma lista de argumentos e devolvam tipos diferentes.

Os tipos primitivos podem ser **promovidos automaticamente** para outros de maior dimensão que estejam definidos.

Exemplo:

```
public class PrimitiveOverloading {  
  
    public void f1 (char x) { System.out.println("f1 (char)"); }  
    void f1 (byte x) { System.out.println("f1 (byte)"); }  
    void f1 (int x) { System.out.println("f1 (int)"); }  
    void f1 (double x) { System.out.println("f1 (double)"); }  
    double f1(double x) { return x * x; } // erro  
  
    public static void main(String[] args) {  
        PrimitiveOverloading p = new PrimitiveOverloading();  
        char c = 'x'; byte b = 1; int i = 3;  
        float f = 1.2f; double d = 2.3;  
        p.f1(c);                p.f1(b);  
        p.f1(i);                p.f1(f); //promove  
        p.f1(d);  
    }  
}
```

f1 (char)
f1 (byte)
f1 (int)
f1 (double)
f1 (double)

Construtores

Se uma classe não tiver **nenhum** construtor, o compilador vai automaticamente criar um **construtor por omissão**.

Se a classe tiver definido **pelo menos um construtor** (com ou sem argumentos), o compilador já **não vai criar nenhum construtor**.

Exemplos:

```
public class Bird {  
    //a classe não tem nenhum construtor  
    public static void main(String[] args) {  
        Bird b = new Bird();  
        //construtor por omissão criado pelo compilador  
    }  
}
```

```
public class Bird {  
    Bird (String nome) {}  
    public static void main(String[] args) {  
        Bird b = new Bird(); // erro, porquê?  
        Bird pelicano = new Bird("Pelicano");  
    }  
}
```

Palavra-chave **this**

Considere o código seguinte:

```
Pessoa p1 = new Pessoa();  
Pessoa p2 = new Pessoa();  
p1.setAge(19);  
p2.setAge(20);
```

Como o método **setAge** consegue saber se foi chamado para o objeto **p1** ou para o objeto **p2**?

Todos os métodos **não estáticos** duma classe recebem um argumento secreto – a referência **this** que aponta para o objeto que deve ser manipulado.

Esta referência é usada :

- para distinguir entre o atributo da classe e o argumento com o mesmo nome;
- para referenciar explicitamente o objeto corrente;
- para chamar um construtor a partir de outro.

Exemplos com this

Exemplo:

```
public void setAge(int idade) {  
    this.idade = idade;  
}  
  
public Pessoa incrementAge()  
{  
    idade++;  
    return this;  
}
```

...

```
Pessoa p = new Pessoa("Pedro Pinto", 19);  
p.setAge(25);  
//operações múltiplas aplicadas ao mesmo objeto:  
p.incrementAge().incrementAge();  
System.out.println(p.getAge());
```

Uso de **this** em construtores

Para evitar a duplicação de código, às vezes surge a necessidade de **chamar um construtor a partir de outro construtor** da mesma classe.

Nesta situação podemos usar **this** com argumentos que especificam que construtor a invocar.

Exemplo:

```
public Pessoa()
{
    System.out.println("A criar uma pessoa");
    nome = "Sem nome";
}

public Pessoa(int age)
{
    this(); //invocar construtor por omissão
    idade = age;
}
...
```

```
Pessoa p = new Pessoa(19);
System.out.println(p);
```

A criar uma pessoa
Sem nome 19

Uso de **this** em construtores

É **impossível** chamar mais que um construtor com **this**.

Se invocar um construtor com **this** esta deve ser a **primeira instrução** na função de construção.

Exemplo:

```
public Pessoa()  
{  
    System.out.println("A criar uma pessoa");  
    nome = "Sem nome";  
}  
  
public Pessoa(String nome, int age)  
{  
    this.nome = nome;  
    idade = age;  
}  
  
public Pessoa(int age)  
{  
    idade = 2; // errado  
    this("teste", 20);  
    this(); //erro  
}
```

Revisão de static

É possível chamar um método **estático** a partir dum método **não estático**?

É possível chamar um método **não estático** a partir dum método **estático**?

Porquê a função **main** é estática?

Exemplo:

```
public static void f2 () { f3 (); } //?  
private void f3 () { f2 (); } //?
```

Destruição de objetos

Em Java a limpeza de objetos é realizada pelo **coletor de lixo** (*garbage collector*).

Logo não há necessidade de destruir objetos explicitamente.

O coletor de lixo só desaloca memória reservada com o operador **new**.

Caso tenha objetos que reservaram memória ou outros recursos de maneira diferente (do operador **new**), é da sua responsabilidade libertar estes recursos.

Para estes casos existe o método **finalize()** com o qual se consegue fazer limpeza na altura da coleção de lixo. **Não deve chamar este método explicitamente**. É o coletor de lixo que o invocará.

Não há garantia de execução deste método pois não há garantia que o coletor de lixo será invocado para o seu objeto. Se o programa nunca atingir limites de ocupação de memória, o coletor de lixo não é invocado para não comprometer o desempenho. Neste caso toda a memória ocupada pelo programa será devolvida ao sistema operativo **em massa** no fim de execução.

Pode forçar a execução do coletor de lixo com `System.gc()` ;

Condição de finalização

A função **finalize** pode ser usada para verificar a **condição de finalização** (*termination condition*) dum objeto.

Exemplo:

```
public class Tank {
    private double capacity;      private double level;
    private boolean full;
    public Tank(double cap) {      capacity = level = cap; full = true; }
    public void fill (double val) { level += val;
        if (level >= capacity) { full = true; level = capacity; } }

    public void spend (double val) { level -= val; full = false;
        if (level < 0.0) level = 0.0; }

    public void finalize() {
        if (!full)
            System.out.println("ERROR!!! Return with full tank!");
    }

    public static void rent()
    {
        Tank t = new Tank(60);
        t.spend(20);
        t.fill(5.6);
    }

    public static void main(String[] args) { rent(); System.gc(); }
}
```

Finalização

Em vez de invocar o coletor de lixo explicitamente, se for necessária a finalização, o código deve ser colocado num bloco **try** {...} **finally** {...}. O bloco **finally** deve invocar os seus métodos de finalização adequados.

Exemplo:

```
public class Tank {
    private double capacity; private double level;
    private boolean full;
    public Tank(double cap) { capacity = level = cap; full = true; }
    public void fill (double val) { level += val;
        if (level >= capacity) { full = true; level = capacity; } }

    public void spend (double val) { level -= val; full = false;
        if (level <= 0.0) level = 0.0; }

    public void cleanup() { if (!full) fill(capacity - level);
        System.out.println("Tank is full!"); }

    public static void main(String[] args) {
        Tank t = new Tank(60);
        try { t.spend(20); t.fill(5.6); }
        finally { t.cleanup(); }
    }
}
```

Inicialização de membros

Os atributos dum objeto são **garantidamente** inicializados com os seus valores por omissão **antes** que seja executado o construtor.

Pode também inicializar membros numa classe **na sua declaração**. Os valores fornecidos serão usados em vez dos por omissão.

Exemplo:

```
class Point { double x, y;  
public Point(double x, double y) { this.x = x; this.y = y; } }
```

```
public class Circle {  
    private String cor = "preto";  
    private Point centro = new Point (0.0, 0.0);  
    private double raio = 1.0;  
  
    public Circle(double r) { raio = r; }  
  
    public String toString() { return "Cor: " + cor + "; raio: " + raio +  
                                   "; centro: " + centro.x + "; " + centro.y;  
    }  
    public static void main(String[] args) {  
        Circle c1 = new Circle(5.5);  
        System.out.println(c1);  
        Circle c2 = new Circle(32);  
        System.out.println(c2);  
    }  
}
```

Cor: preto; raio: 5.5; centro: 0.0; 0.0

Cor: preto; raio: 32.0; centro: 0.0; 0.0

Ordem de inicialização

Dentro de uma classe a **ordem de inicialização é determinada pela ordem das definições dos atributos**.

Mesmo que os atributos apareçam misturados com os métodos serão sempre inicializados antes que seja executado qualquer método.

Inicialização dos atributos **estáticos** só ocorre uma única vez quando a classe é carregada.

Inicialização dos **membros estáticos** é realizada **antes** de todos os outros membros.

Em resumo a ordem é:

- atributos estáticos;
- os membros restantes inicializados com os seus valores por omissão;
- atribuição dos valores fornecidos na declaração (caso existam) por cima dos valores por omissão;
- execução do construtor.

Exemplo de ordem de inicialização

Exemplo:

```
class Point {  
    double x, y;  
    public Point(double x, double y)  
    { System.out.println("A criar um ponto"); this.x = x; this.y = y; }  
}  
  
class Circle {  
    public Circle(double r)  
    { System.out.println("A criar um círculo"); raio = r; }  
    public String toString() {  
        return "raio: " + raio + "; centro: " + centro.x + "; " + centro.y; }  
    static private Point centro = new Point (0.0, 0.0);  
    private double raio = 1.0;
```

```
public class InitOrder {  
    static Point r = new Point(Math.random()*10, Math.random()*10);
```

```
    public static void main(String[] args) {  
        System.out.println("A criar c1");  
        Circle c1 = new Circle(5.5);  
        System.out.println(c1);  
        System.out.println("A criar c2");  
        Circle c2 = new Circle(32);  
        System.out.println(c2);  
    }
```

A criar um ponto
A criar c1
A criar um ponto
A criar um círculo
raio: 5.5; centro: 0.0; 0.0
A criar c2
A criar um círculo
raio: 32.0; centro: 0.0; 0.0

Inicialização estática explícita

Java permite agrupar inicializações estáticas numa cláusula especial **static**.

Este bloco, como todas as outras inicializações estáticas, é executado só uma vez quando a classe é carregada.

Exemplo:

```
class Circle {  
    public Circle(double r) {  
        System.out.println("A criar um círculo"); raio = r; }  
  
    static String cor;  
  
    public String toString() {  
        return "raio: " + raio + "; centro: " +  
            centro.x + "; " + centro.y;  
    }  
  
    static private Point centro;  
  
    static  
    {  
        centro = new Point (0.0, 0.0);  
        cor = "preto";  
    }  
    private double raio = 1.0;  
}
```

Inicialização explícita não estática

À semelhança da inicialização explícita estática, Java permite agrupar inicializações não estáticas num bloco (*non-static instance initialization*).

Deste modo consegue-se garantir que certas inicializações ocorrem sempre, independentemente do construtor que foi chamado.

Exemplo:

```
public class Circle {  
    private String cor;    private Point centro;    private double raio;  
    { cor = "preto"; centro = new Point (0.0, 0.0); raio = 1.0; }  
  
    public Circle(double r) { raio = r; }  
    public Circle(String c) { cor = c; }  
    public Circle(Point p) { centro = p; }  
  
    public String toString() { return "Cor: " + cor + ", raio: " + raio + "; centro: " + centro.x + " " + centro.y; }  
  
    public static void main(String[] args) {  
        Circle c1 = new Circle(5.5);  
        System.out.println(c1);  
        Circle c2 = new Circle("amarelo");  
        System.out.println(c2);  
        Circle c3 = new Circle(new Point (2.2, -3.3));  
        System.out.println(c3);  
    }  
}
```

Qual é a diferença
com o atributos
estáticos?

```
Cor: preto; raio: 5.5; centro: 0.0; 0.0  
Cor: amarelo; raio: 1.0; centro: 0.0; 0.0  
Cor: preto; raio: 1.0; centro: 2.2; -3.3
```

Tipo enum

A palavra reservada **enum** permite criar conjuntos de valores constantes.

Exemplo:

```
enum Mes
{ JANEIRO, FEVEREIRO, MARÇO, ABRIL, MAIO, JUNHO,
  JULHO, AGOSTO, SETEMBRO, OUTUBRO, NOVENBRO, DEZEMBRO
}
...
Mes m = Mes.OUTUBRO; int dias;
switch (m)
{
    case ABRIL:
    case JUNHO:
    case SETEMBRO:
    case NOVENBRO: dias = 30; break;
    case FEVEREIRO: dias = 28; break;
    default: dias = 31;
}
```


Exercícios

Identifique erros no programa seguinte. Qual é a saída deste programa (depois de corrigidos os erros)?

```
public class X{
    static String n;
    X (String nome) { n = nome; }
    public String toString() { return n; }

    public static void main(String[] args) {
        X um = new X();

        X outro = new X("xaxa");
        System.out.println(outro);

        X terceiro = new X("ohoh");
        System.out.println(terceiro);

        System.out.println(outro);
    }
}
```

Exercícios (cont.)

Identifique erros no programa seguinte:

```
class Point {  
    double x, y;  
    public Point(double x, double y){ x = x; y = y; }  
}  
  
class Circle {  
    public Circle(double r) { raio = r; }  
  
    public String toString()  
    { return "raio: " + raio + "; centro: " +  
      centro.x + "; " + centro.y; }  
  
    static private Point centro;  
    private double raio = 1.0;  
}  
  
public class InitOrder {  
    public static void main(String[] args) {  
        Circle c1 = new Circle(5.5);  
        System.out.println(c1);  
    }  
}
```

Bibliografia

Bruce Eckel, *Thinking in Java*, 4th edition, Prentice-Hall, 2006

=> Capítulos “Everything is an Object”, “**Initialization & Cleanup**”