Classes e Objetos

UA.DETI.POO

Conceito básico de classe

- Definição duma classe (ficheiro Exemplo.java):

```
public class Exemplo {
    // dados
    // métodos
}
```

- O ficheiro Exemplo.java deve conter uma classe pública denominada Exemplo.
  - Devemos usar uma nomenclatura do tipo Person, SomeClass, SomeLongNameForClass, …
  - Java é uma linguagem case-sensitive (i.e. Exemplo != exemplo)
- Esta classe deve ser declarada como public

Espaço de Nomes - Package

- Em Java a gestão do espaço de nomes (*namespace*) é efetuado 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 nas primeiras linhas 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<String> al = new ArrayList<>();
```

- De outra forma teríamos de escrever:

```
java.util.ArrayList<String> al = new java.util.ArrayList<>();
```

Criar um package

- Na primeira linha de código:

```
package poo;
```

  - garante que a classe pública dessa unidade de compilação fará parte do package poo.

- O espaço de nomes é baseado numa estrutura de sub-directórios
  - Este package vai corresponder a uma entrada de directório: **$CLASSPATH/poo**
  - Boa prática usar DNS invertido: **pt.ua.deti.poo**

- A sua utilização será na forma:

```
poo.Book sr = new poo.Book();
```

  - ou

```
import poo.*
Book sr = new Book();
```

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:

```
public class Book {
    String title;
    int pubYear;
}
```

Exemplo de classe

```
public class Book {
   String title;
   int pubYear;

   String getTitle() {
      return title;
   }
   int getPubYear() {
      return pubYear;
   }
   void setTitle(String atitle) {
      title = atitle;
   }
   void setPubYear(int apubYear) {
      pubYear = apubYear;
   }
}
```

Objetos

- Objetos são instâncias de classes

```
Book oneBook = new Book();
Book otherBook = new Book();
Book book3 = new Book();
```

- Todos os objetos são manipulados através de referências

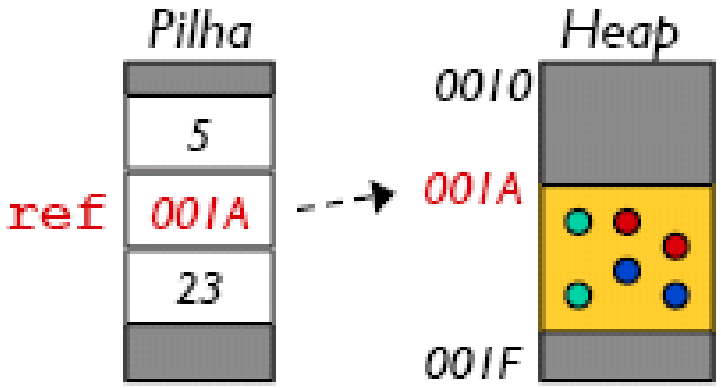
```
Pessoa nome1, nome2;
nome1 = new Pessoa("Manuel");
nome2 = nome1;
```

- Todos os objetos devem ser explicitamente criados.

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

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)



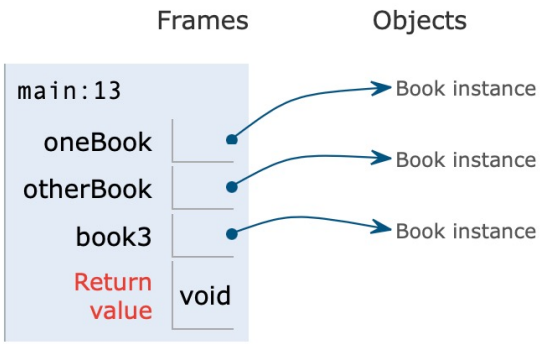
Objetos

Java 8
known limitations

```
1  public class Book {
2      public Book() {
3      }
4
5      public static void main(String[] args) {
6        Book oneBook = new Book();
7        Book otherBook = new Book();
8        Book book3 = new Book();
9
10       System.out.println(oneBook);
11       System.out.println(otherBook);
12       System.out.println(book3);
13     }
14 }
```

Print output (drag lower right corner to resize)

```
Book@504bae78
Book@3b764bce
Book@759ebb3d
```



https://tinyurl.com/yj7fubt7

Métodos

- Métodos, mensagens, funções, procedimentos
- A invocação é sempre efetuada através da notação de ponto.

```
oneBook.setTitle("Turismo em Aveiro");
otherBook.setPubYear(2000);
```

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

```
Math.sqrt(34);
otherBook.setPubYear(2000);
```

toString

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

```
Book oneBook = new Book();
oneBook.setTitle("Turismo em Aveiro");
System.out.println(oneBook); // oneBook.toString()
```

```
Book@33909752
```

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

```
@Override
public String toString() {
    return "Book: title=" + title + "; pubYear=" + pubYear;
}
```

```
Book: title=Turismo em Aveiro; pubYear=0
```

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

Inicialização de membros

- Dentro de uma classe, a inicialização de variáveis pode ser feita na sua declaração.

```
class Measurement {
     int i = 25;
     char c = 'K';
}
```

- Contudo, esta inicialização é igual para todos os objetos da classe
  - A solução mais comum é utilizar um construtor.

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

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 Book {
   String title;
   int pubYear;

   public Book(String t, int py) {
       title = t;
       pubYear = py;
    }
    // ...
}
```

Construtor por omissão

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

```
class Book {
    String title;
    int pubYear;
}
```

```
Book m = new Book(); // ok
```

  - Se houver pelo menos um construtor associado a uma dada classe, o compilador já não cria o de omissão.

```
class Book {
    String title;
    int pubYear;
    Book(int py) { pubYear = py; }
}
```

```
Book m = new Book(); // ok?
```

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?

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'
byte	(byte)0
short	(short)0
int	0
long	0L
float	0.0f
double	0.0
(outros tipos)	null

Sobreposição (Overloading)

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

```
void sort(int[] a);
void sort(Book[] b);
```

- A ligação estática verifica a assinatura da função (nome + argumentos)

Sobreposição (Overloading)

```
public class Test {

   public void someFunction(String[] s) {
       …
   }
   public int someFunction(String[] b) {
       …
   }

   // ...

   String[] someStrings = {“first string”, “another string”, “last”};
   someFunction(someStrings);
}
```

Problema ?

Sobreposição (Overloading)

```
public class Test {

    public void someFunction(String[] s) {
        …
    }
    public int someFunction(String[] b) {
        …
    }

    // ...

    String[] someStrings = {“first string”, “another string”, “last”};
    someFunction(someStrings);
}
```

Como distinguir?

Sobreposição (Overloading)

```
public class Test {

   public void someFunction(String[] s) {
       …
   }
   public int someFunction(String[] b) {
       …
   }

   // ...

   String[] someStrings = {“first string”, “another string”, “last”};
   someFunction(someStrings);
}
```

Como distinguir?

- 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 Book {
   public Book(String title, int pubYear)  {...}
   public Book(String title) {...}
   public Book() {...}
}

Book c1 = new Book("A jangada de pedra", 1986);
Book c2 = new Book("Galveias");
Book c3 = new Book();
```

A referência this

- A referência this pode ser utilizada dentro de cada objeto para referenciar esse mesmo objeto

```
public class Book {
    String title;
    int pubYear;
    public Book(String title, int pubYear) {
        this.title = title;
        this.pubYear = pubYear;
    }
}

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

A referência this

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

```
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 Book(String title, int pubYear) {
    this.title = title;
    this.pubYear = pubYear;
}
public Book(String title) {
    this(title, 2000);
}
```

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 têm associada 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).

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 sobre o identificador da classe

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

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

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[ ]
   }
}
```

Vetores de objetos

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

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

Book[] xC = new Book[10]; // 10 refs! Não são 10 Books!!
```

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 = 10;
    {  int i = k+1;;
    } // ‘i’ não é visível aqui
} // ‘k’ não é visível aqui
```

- Exemplo ilegal

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

Alcance de referências e objetos

- Exemplo com referências e objetos

```
{
    Book b1 = new Book("Memória de Elefante");
}
// 'b1' já não é visível aqui
```

- Neste caso a referência *b1* é libertada (removida) e o objeto deixa de poder ser usado
  - Será removido pelo **Garbage collector**

Sumário

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

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, protected, *default*, 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
  - **protected** – visível dentro do mesmo package e classes derivadas
  - "omissão" – visível dentro do mesmo package
  - **private** – apenas visível dentro da 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!

  - Todos os atributos deverão ser privados.

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

porquê?

Exemplo

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

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

Seletores/Modificadores (getters/setters)

- Seletor
  - Devolve o valor atual 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 radius(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')

Exemplo

```
class Point {
    private double x;
    private double y;

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

    public double getX() { return x; }
    public double getY() { return y; }

    public double distanceTo(Point p) {...}
}
```

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