

Classes e Objetos

UA.DETI.POO

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 a caracterizam e depois vão compor o objeto (i.e., o que podem armazenar)
 - Métodos que operam sobre (i) as características daquele objeto, (ii) características comuns a todos os objetos da classe, (iii) parâmetros de entrada dados na invocação do método (i.e., o que podem fazer)

O que é uma classe?

❖ Exemplo:

```
class Car {  
    String model;  
    int year;  
    int getYear() { return this.year; }  
  
    Car(String m, int yr) {model=m; year=yr;}  
  
    int getAge() {  
        int currentYear = java.util.Calendar.getInstance().get(java.util.Calendar.YEAR);  
        return getAge(currentYear);  
    }  
    int getAge(int currentYear) {  
        return currentYear - year;  
    }  
}
```

Classes são o esqueleto para construir objetos

❖ Podem ter:

- Valores (*i.e., as propriedades que a caracterizam*)
- Métodos

❖ Necessário “alocar” espaço e construir o objeto (*i.e., new*)

Java

```
1 class Car {
2   String model;
3   int year;
4   Car(String m, int yr) {model=m; year=yr;}
5 }
6
7 public class CarTest {
8   public static void main(String[] args) {
9     Car toyota = new Car("Toyota Camry",2009);
10    Car ford = new Car("Ford F-150",2023);
11    String desc = "This code is about cars!";
12  }
13 }
```

[Edit Code & Get AI Help](#)

→ line that just executed
→ next line to execute

Done running (7 steps)

Frames

main:12

toyota	
ford	
desc	"This code is about cars!"

Objects

Car instance

model	"Toyota Camry"
year	2009

Car instance

model	"Ford F-150"
year	2023

<https://pythontutor.com/articles/java-visualizer.html>

Classes são “blueprints” para objetos

```
class Car {  
    String model;  
    int year;  
    int getYear() { return this.year; }  
  
    Car(String m, int yr) {model=m; year=yr;}  
  
    int getAge() {  
        int currentYear = java.util.Calendar.getInstance().get(java.util.Calendar.YEAR);  
        return getAge(currentYear);  
    }  
    int getAge(int currentYear) {  
        return currentYear - year;  
    }  
}
```

Classes são “blueprints” para objetos

❖ Podem **ter valores / properties**

```
class Car {  
    String model;  
    int year;  
    int getYear() { return this.year; }  
  
    Car(String m, int yr) {model=m; year=yr;}  
  
    int getAge() {  
        int currentYear = java.util.Calendar.getInstance().get(java.util.Calendar.YEAR);  
        return getAge(currentYear);  
    }  
    int getAge(int currentYear) {  
        return currentYear - year;  
    }  
}
```

Classes são “blueprints” para objetos

❖ Podem **ter valores / properties**

```
class Car {  
    String model;  
    int year;  
    int getYear() { return this.year; }  
  
    Car(String m, int yr) {model=m; year=yr;}  
  
    int getAge() {  
        int currentYear = java.util.Calendar.getInstance().get(java.util.Calendar.YEAR);  
        return getAge(currentYear);  
    }  
    int getAge(int currentYear) {  
        return currentYear - year;  
    }  
}
```

Classes

- ❖ Devemos usar uma nomenclatura do tipo Person, SomeClass, SomeLongNameForClass, ...
- ❖ Java é uma linguagem case-sensitive (i.e. Exemplo != exemplo)
- ❖ Se classe for pública i.e., utilizável por outras classes
 - Deve ser declarada pública
 - O ficheiro Car.java deve conter uma classe pública denominada Car.

```
public class Car {  
  
    static int count = 0;  
  
    String model;  
    int year;  
    int matricula;  
    int getYear() { return this.year; }  
  
    Car(String m, int yr) {model=m; year=yr;  
        this.matricula = Car.count++;  
    }  
  
    int getAge() {  
        int currentYear =  
            java.util.Calendar.getInstance().get(java.util.Calendar.YEAR);  
        return getAge(currentYear);  
    }  
  
    int getAge(int currentYear) {  
        return currentYear - year;  
    }  
}
```


Classes

- ❖ Devemos usar uma nomenclatura do tipo Person, SomeClass, SomeLongNameForClass, ...
- ❖ Java é uma linguagem case-sensitive (i.e. Exemplo != exemplo)
- ❖ Se classe for **pública** i.e., utilizável por outras classes
 - Deve ser declarada pública
 - O ficheiro Car.java deve conter uma classe pública denominada Car.

```
public class Car {  
    static int count = 0;  
  
    String model;  
    int year;  
    int matricula;  
    int getYear() { return this.year; }  
  
    Car(String m, int yr) {model=m; year=yr;  
        this.matricula = Car.count++;  
    }  
  
    int getAge() {  
        int currentYear =  
            java.util.Calendar.getInstance().get(java.util.Calendar.YEAR);  
        return getAge(currentYear);  
    }  
    int getAge(int currentYear) {  
        return currentYear - year;  
    }  
}
```

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 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

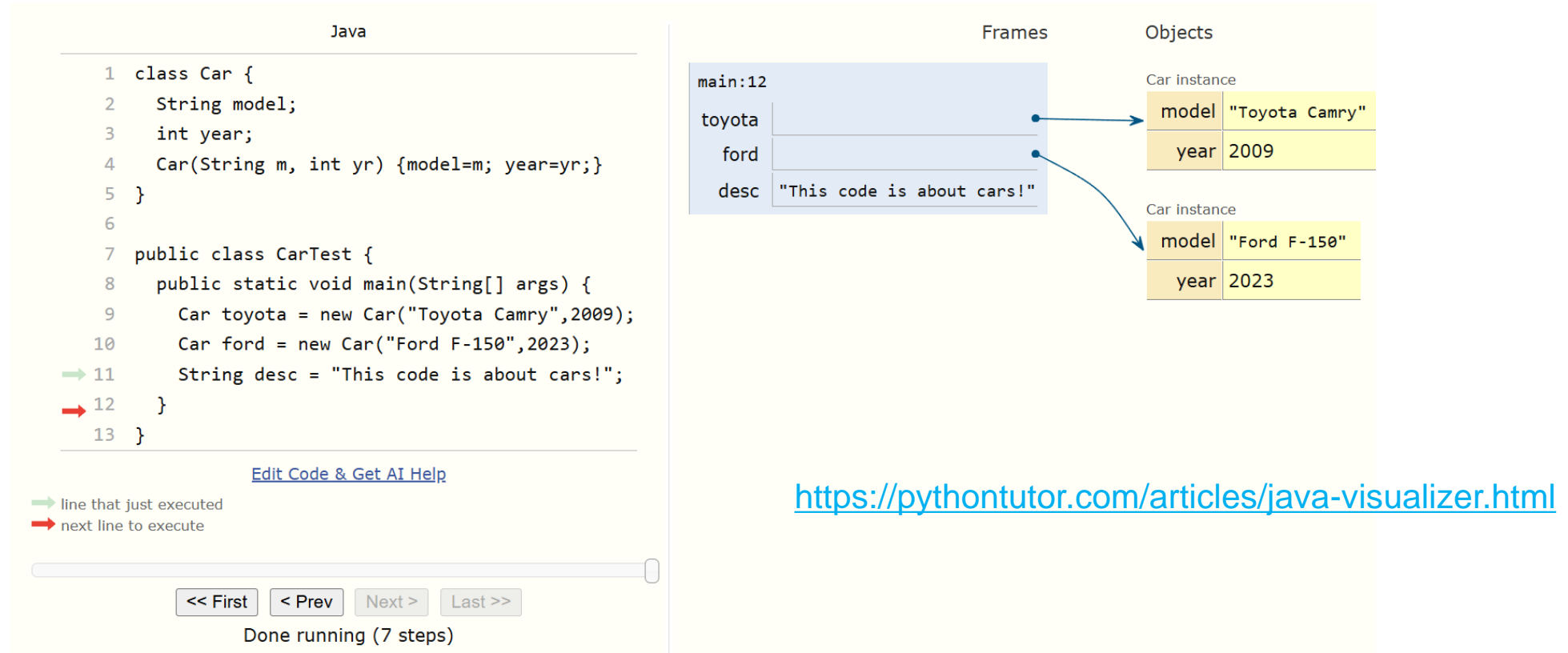
```
Pessoa nome1, nome2;  
nome1 = new Pessoa("Manuel");  
nome2 = nome1;
```

❖ Todos os objetos devem ser explicitamente criados.

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

Objetos são tipos de dados por referência

- ❖ Variáveis não têm “os dados”, são referências em memória para a localização “dos dados”.



Contrasta com Tipos primitivos

- ❖ Não são objetos.
- ❖ Variáveis armazenam diretamente os dados.
- ❖ Dimensão em memória depende do tipo de dados.

```
int myNum = 5;           // Integer (whole number)
float myFloatNum = 5.99f; // Floating point number
char myLetter = 'D';     // Character
boolean myBool = true;   // Boolean
String myText = "Hello"; // String
```

https://www.w3schools.com/java/java_data_types.asp

Tipos primitivos e classes adaptadoras (“wrappers”)

- ❖ As classes adaptadoras criam objetos para os tipos primitivos
- ❖ Contêm métodos especializados

Primitive Data Type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

https://www.w3schools.com/java/java_wrapper_classes.asp

Inicializar um objeto

usando construtores das classes

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* (sobrepuesto) 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;  
    }  
    // ...  
}
```

Objetos são tipos de dados por referência

❖ Necessário “alocar” espaço (i.e., new)

❖ Usar construtores

– Inicializar atributos

```
public class CarTest {  
    public static void main(String[] args) {  
        Car toyota = new Car("Toyota Camry", 2009);  
        Car ford = new Car("Ford F-150", 2023);  
        String desc = "This code is about cars!";  
        System.out.println( "O Toyota é de "+toyota.getAge()+" anos.");  
        System.out.println( "O Ford é de "+ford.getAge()+" anos.");  
    }  
}
```

```
class Car {  
    String model;  
    int year;  
    int getYear() { return this.year; }  
  
    Car(String m, int yr) {model=m; year=yr;}  
  
    int getAge() {  
        int currentYear = java.util.Calendar.getInstance().get(java.util.Calendar.YEAR);  
        return getAge(currentYear);  
    }  
  
    int getAge(int currentYear) {  
        return currentYear - year;  
    }  
}
```



Frames

Objects

main:20

<init>:6

this	
m	"Toyota Camry"
yr	2009

Car instance

model	null
year	0

Objetos são tipos de dados por referência

❖ **Necessário “alocar” espaço (i.e., new)**

❖ **Usar construtores**

– **Inicializar atributos**

```
public class CarTest {  
    public static void main(String[] args) {  
        Car toyota = new Car("Toyota Camry", 2009);  
        Car ford = new Car("Ford F-150", 2023);  
        String desc = "This code is about cars!";  
        System.out.println("O Toyota é de "+toyota.getAge()+" anos.");  
        System.out.println("O Ford é de "+ford.getAge()+" anos.");  
    }  
}
```

↓

```
class Car {  
    String model;  
    int year;  
    int getYear() { return this.year; }  
  
    Car(String m, int yr) { model=m; year=yr; }  
  
    int getAge() {  
        int currentYear = java.util.Calendar.getInstance().get(java.util.Calendar.YEAR);  
        return getAge(currentYear);  
    }  
  
    int getAge(int currentYear) {  
        return currentYear - year;  
    }  
}
```

Frames

main:20

<init>:6

this	
m	"Toyota Camry"
yr	2009

Objects

Car instance

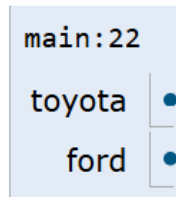
model	"Toyota Camry"
year	2009

Objetos são tipos de dados por referência

```
class Car {  
    String model;  
    int year;  
    int getYear() { return this.year; }  
  
    Car(String m, int yr) {model=m; year=yr;}  
  
    int getAge() {  
        int currentYear = java.util.Calendar.getInstance().get(java.util.Calendar.Y  
        return getAge(currentYear);  
    }  
    int getAge(int currentYear) {  
        return currentYear - year;  
    }  
}
```

```
public class CarTest {  
    public static void main(String[] args) {  
        Car toyota = new Car("Toyota Camry",2009);  
        Car ford = new Car("Ford F-150",2023);  
        String desc = "This code is about cars!";  
        System.out.println( "O Toyota é de "+toyota.getAge()+" anos.");  
        System.out.println( "O Ford é de "+ford.getAge()+" anos.");  
    }  
}
```

Frames



Objects

Car instance

model	"Toyota Camry"
year	2009

Car instance

model	"Ford F-150"
year	2023

Objetos são tipos de dados por referência

❖ Podem ter métodos

- Sobre objetos

```
public class CarTest {  
    public static void main(String[] args) {  
        Car toyota = new Car("Toyota Camry", 2009);  
        Car ford = new Car("Ford F-150", 2023);  
        String desc = "This code is about cars!";  
        System.out.println( "O Toyota é de " + toyota.getAge() + " anos.");  
        System.out.println( "O Ford é de " + ford.getAge() + " anos.");  
    }  
}
```

```
class Car {  
    String model;  
    int year;  
    int getYear() { return this.year; }
```

```
    Car(String m, int yr) {model=m; year=yr;}  
}
```

```
    int getAge() {  
        int currentYear = java.util.Calendar.getInstance().get(java.util.Calendar.YEAR);  
        return getAge(currentYear);  
    }  
    int getAge(int currentYear) {  
        return currentYear - year;  
    }  
}
```

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?
```

<https://tinyurl.com/4rbzxcd7>

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();
```

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?

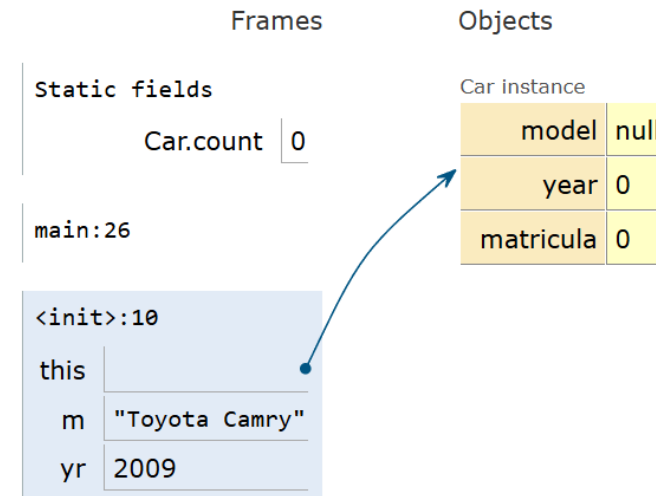
Classes podem ter membros estáticos

❖ manter informação da classe

- Partilhar valores ao nível da classe (similar variável global)
- e.g., constantes, contadores, contexto,

```
public class Car {  
  
    static int count = 0;  
  
    String model;  
    int year;  
    int matricula;  
    int getYear() { return this.year; }  
  
    Car(String m, int yr) {model=m; year=yr;  
        this.matricula = Car.count++;  
    }  
  
    int getAge() {  
        int currentYear =  
            java.util.Calendar.getInstance().get(java.util.Calendar.YEAR);  
        return getAge(currentYear);  
    }  
    int getAge(int currentYear) {  
        return currentYear - year;  
    }  
}
```

```
public class CarTest {  
    public static void main(String[] args) {  
        Car toyota = new Car("Toyota Camry",2009);  
        Car ford = new Car("Ford F-150",2023);  
        String desc = "This code is about cars!";  
        System.out.println( "O Toyota é de "+toyota.getAge()+" anos.");  
        System.out.println( "O Ford é de "+ford.getAge()+" anos.");  
        System.out.println("tenho "+Car.count+" carros");  
    }  
}
```



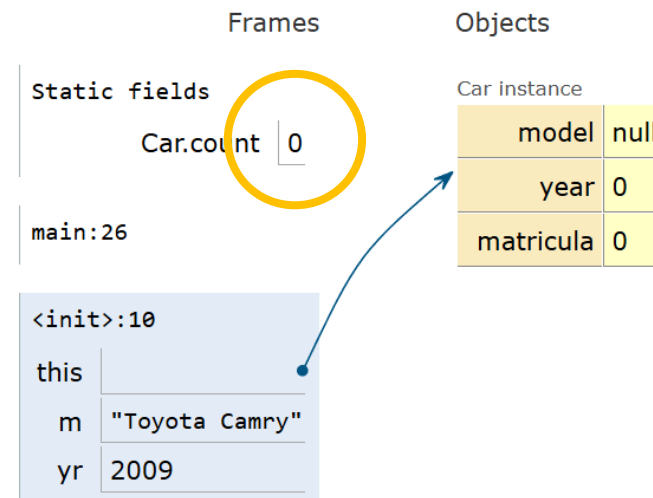
Classes podem membros estáticos

❖ manter informação da classe

- Partilhar valores ao nível da classe (similar variável global)
- e.g., constantes, contadores, contexto,

```
public class Car {  
    static int count = 0;  
    String model;  
    int year;  
    int matricula;  
    int getYear() { return this.year; }  
    Car(String model, int year) {  
        this.model = model;  
        this.year = year;  
        this.matricula = Car.count++;  
    }  
    int getAge() {  
        int currentYear =  
            java.util.Calendar.getInstance().get(java.util.Calendar.YEAR);  
        return getAge(currentYear);  
    }  
    int getAge(int currentYear) {  
        return currentYear - year;  
    }  
}
```

```
public class CarTest {  
    public static void main(String[] args) {  
        Car toyota = new Car("Toyota Camry", 2009);  
        Car ford = new Car("Ford F-150", 2023);  
        String desc = "This code is about cars!";  
        System.out.println("O Toyota é de "+toyota.getAge()+" anos.");  
        System.out.println("O Ford é de "+ford.getAge()+" anos.");  
        System.out.println("tenho "+Car.count+" carros");  
    }  
}
```



Classes podem membros estáticos

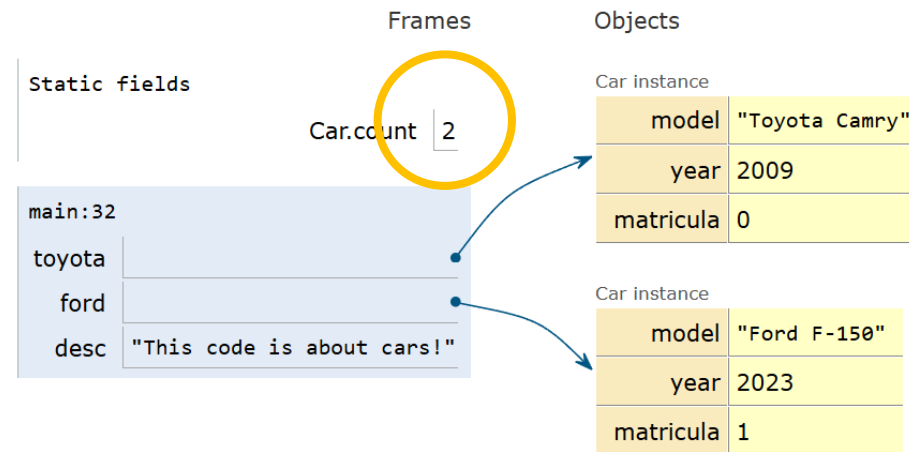
❖ manter informação da classe

- Partilhar valores ao nível da classe (similar variável global)
- e.g., constantes, contadores, contexto,

```
public class Car {  
    static int count = 0;  
  
    String model;  
    int year;  
    int matricula;  
    int getYear() { return this.year; }  
  
    Car(String m, int yr) {model=m; year=yr;  
        this.matricula = Car.count++;  
    }  
  
    int getAge() {  
        int currentYear =  
            java.util.Calendar.getInstance().get(java.util.Calendar.YEAR);  
        return getAge(currentYear);  
    }  
    int getAge(int currentYear) {  
        return currentYear - year;  
    }  
}  
  
public class CarTest {  
    public static void main(String[] args) {  
        Car toyota = new Car("Toyota Camry",2009);  
        Car ford = new Car("Ford F-150",2023);  
        String desc = "This code is about cars!";  
        System.out.println( "O Toyota é de "+toyota.getAge()+"  
        System.out.println( "O Ford é de "+ford.getAge()+" anc  
        System.out.println("tenho "+Car.count+" carros");  
    }  
}
```

Print output (drag lower right corner to resize)

```
O Toyota é de 16 anos.  
O Ford é de 2 anos.  
tenho 2 carros
```



Métodos

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

```
oneBook.setTitle("Turismo em Aveiro");  
otherBook.setPubYear(2025);
```

- ❖ 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(2025);
```

Chaining

- ❖ Chaining é a capacidade de encadear a invocação métodos, utilizando o estado/resultado anterior como entrada do próximo método.
- ❖ Outra utilização da referência `this` é para retornar, num dado método, a referência para esse objeto (permitindo este encadeamento).

<https://www.geeksforgeeks.org/method-chaining-in-java-with-examples/>

```
class A {  
  
    private int a;  
    private float b;  
  
    A() { System.out.println("Calling The Constructor"); }  
  
    public A setint(int a)  
    {  
        this.a = a;  
        return this;  
    }  
  
    public A setfloat(float b)  
    {  
        this.b = b;  
        return this;  
    }  
  
    void display()  
    {  
        System.out.println("Display=" + a + " " + b);  
    }  
}  
  
// Driver code  
public class Example {  
    public static void main(String[] args)  
    {  
        // This is the "method chaining".  
        new A().setint(10).setfloat(20).display();  
    }  
}
```

This: Auto referenciar objeto

- ❖ A referência **this** pode ser utilizada dentro de cada objeto para autorreferenciar esse mesmo objeto

```
public class Car {  
  
    static int count = 0;  
  
    String model;  
    int year;  
    int matricula;  
    int getYear() { return this.year; }  
  
    Car(String m, int yr) {model=m; year=yr;  
        this.matricula = Car.count++;  
    }  
  
    int getAge() {  
        int currentYear =  
            java.util.Calendar.getInstance().get(java.util.Calendar.YEAR);  
        return getAge(currentYear);  
    }  
  
    int getAge(int currentYear) {  
        return currentYear - year;  
    }  
}
```


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() */ }  
}
```

Chaining

- ❖ Outra utilização da referência `this` é para retornar, num dado método, a referência para esse objeto.

<https://www.geeksforgeeks.org/method-chaining-in-java-with-examples/>

```
class A {  
  
    private int a;  
    private float b;  
  
    A() { System.out.println("Calling The Constructor"); }  
  
    public A setint(int a)  
    {  
        this.a = a;  
        return this;  
    }  
  
    public A setfloat(float b)  
    {  
        this.b = b;  
        return this;  
    }  
  
    void display()  
    {  
        System.out.println("Display=" + a + " " + b);  
    }  
}  
  
// Driver code  
public class Example {  
    public static void main(String[] args)  
    {  
        // This is the "method chaining".  
        new A().setint(10).setfloat(20).display();  
    }  
}
```

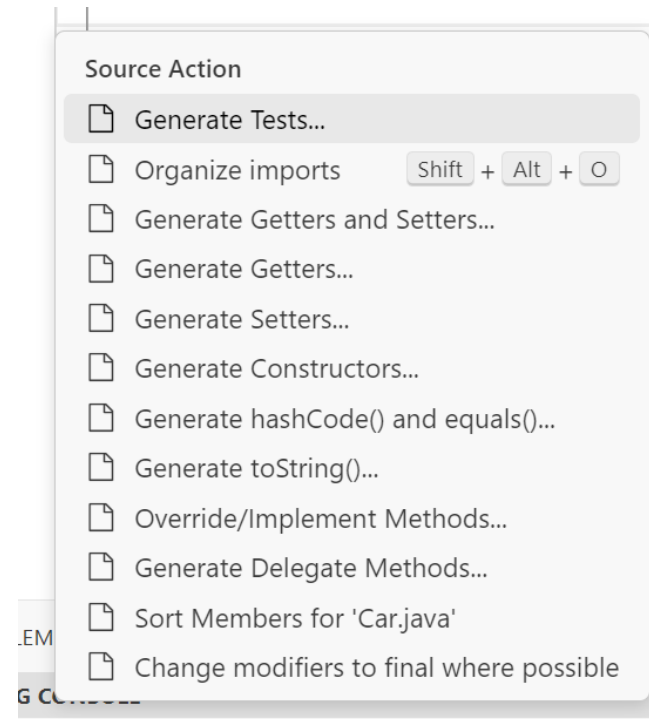


Métodos comuns a todos os objetos

- ❖ Em Java, todas as classes derivam da super classe `java.lang.Object`
- ❖ Métodos desta classe:
 - `toString()`
 - `equals()`
 - `hashCode()`
 - `getClass()`
 - `clone()`
 - `wait()`
 - `notify()`
 - `notifyAll()`
 - `finalize()`

Métodos comuns a todos os objetos

- ❖ Em Java, todas as classes derivam da super classe `java.lang.Object`
- ❖ Métodos desta classe:
 - `toString()`
 - `equals()`
 - `hashCode()`
 - `getClass()`
 - `clone()`
 - `wait()`
 - `notify()`
 - `notifyAll()`
 - `finalize()`



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

toString()

- ❖ Todos os objetos em Java entendem a mensagem toString()
- ❖ Geralmente é necessário redefinir este método de modo a fornecer um resultado mais adequado.

```
@Override
public String toString() {
    return "Car [model=\"" + model + "\", year=\"" + year + "\", matricula=\"" +
        matricula + "\", getAge()=\"" + getAge() + "\"]";
}
```

```
public class Car {

    static int count = 0;

    String model;
    int year;
    int matricula;
    int getYear() { return this.year; }

    Car(String m, int yr) {model=m; year=yr;
        this.matricula = Car.count++;
    }

    int getAge() {
        int currentYear =
            java.util.Calendar.getInstance().get(java.util.Calendar.YEAR);
        return getAge(currentYear);
    }

    int getAge(int currentYear) {
        return currentYear - year;
    }
}
```

toString()

```
Circulo c1 = new Circulo(1.5, 0, 0);  
System.out.println( c1 );
```

Circulo@1afa3

c1.toString() é invocado automaticamente

- O método toString() deve ser sempre redefinido para ter um comportamento de acordo com o objeto

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

Centro : (1.5, 0) Raio : 0

equals()

- ❖ A expressão `c1 == c2` verifica se as referências `c1` e `c2` apontam para o mesmo objeto
 - Caso `c1` e `c2` sejam variáveis automáticas a expressão anterior compara valores
- ❖ O método `equals()` testa se dois objetos são iguais

```
Circulo p1 = new Circulo(0, 0, 1);
Circulo p2 = new Circulo(0, 0, 1);
System.out.println(p1 == p2);           // false
System.out.println(p1.equals(p2)); // false (porquê?)
```
- `equals()` deve ser redefinido em concordância com os atributos dessa classe e é necessário para comparar objetos
 - Circulo, Ponto, Complexo ...

Problemas com equals()

❖ Propriedades da igualdade

- reflexiva: $x.equals(x) \rightarrow true$
- simétrica: $x.equals(y) \leftrightarrow y.equals(x)$
- transitiva: $x.equals(y) \text{ AND } y.equals(z) \rightarrow x.equals(z)$

❖ Devemos respeitar a assinatura `Object.equals(Object o)`

```
public class Circulo {  
    //...  
    @Override  
    public boolean equals(Object obj) { //...  
    }  
}
```

❖ Problemas

- E se 'obj' for null?
- E se referenciar um objeto diferente de Circulo?

equals()

```
@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Car other = (Car) obj;
    if (model == null) {
        if (other.model != null)
            return false;
    }
    if (matricula != other.matricula)
        return false;
    return true;
}
```

```
public class Car {

    static int count = 0;

    String model;
    int year;
    int matricula;
    int getYear() { return this.year; }

    Car(String m, int yr) {model=m; year=yr;
        this.matricula = Car.count++;
    }

    int getAge() {
        int currentYear =
            java.util.Calendar.getInstance().get(java.util.Calendar.YEAR);
        return getAge(currentYear);
    }

    int getAge(int currentYear) {
        return currentYear - year;
    }
}
```

Circulo.equals()

@Override

```
public boolean equals(Object obj) {  
    if (this == obj)  
        return true;  
    if (obj == null)  
        return false;  
    if (getClass() != obj.getClass())  
        return false;  
    Circulo other = (Circulo) obj;  
    // verify if the object's attributes are equals  
    if (centro == null) {  
        if (other.centro != null)  
            return false;  
    } else if (!centro.equals(other.centro))  
        return false;  
    if (raio != other.raio)  
        return false;  
    return true;  
}
```

hashCode()

- ❖ Sempre que o método `equals()` for reescrito, `hashCode()` também deve ser
 - Objetos iguais devem retornar códigos de hash iguais
- ❖ O objetivo do hash é ajudar a identificar qualquer objeto através de um número inteiro

// Circulo.hashCode() – Exemplo muito simples !!!

```
public int hashCode() {  
    return raio * centro.x() * centro.y();  
}
```

//..

Circulo c1 = new Circulo(10,15,27);

Circulo c2 = new Circulo(10,15,27);

Circulo c3 = new Circulo(10,15,28);

4050

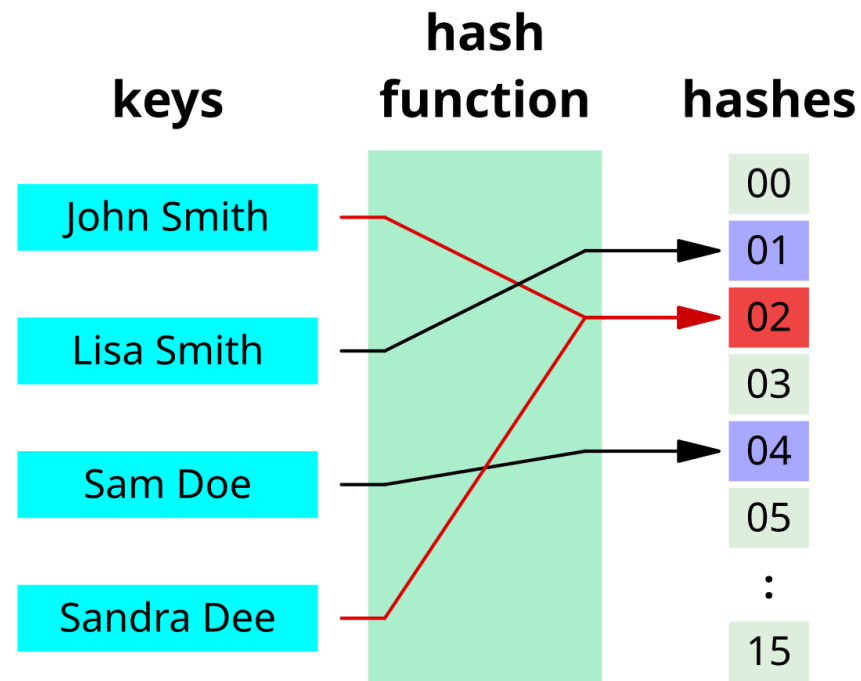
4050

4200

- **A construção de uma boa função de hash não é trivial. Para a sua construção recomendam-se outras fontes**

O que é uma função de Hash?

- ❖ Uma função de hash (ou de dispersão) serve para transformar dados de tamanho arbitrário num valor de tamanho mais pequeno, mais rápido de processar e fácil de armazenar, que preserve algumas características de identidade no domínio do nosso problema.



O que é uma função de Hash?

- ❖ **Fundamental**: Objetos iguais têm de ter um valor de hash igual.
(não funciona caso não se verifique)
- ❖ Objetos diferentes podem ter o mesmo valor de hash (i.e., uma colisão).
 - **Indicador de performance**: Uma função de hash é tão melhor quanto a sua capacidade de evitar estas colisões *(no domínio do nosso problema)*.
- ❖ Chama-se “função de dispersão” porque esta função de hash é tão boa quanto a sua capacidade de dispersar os dados de entrada pelo conjunto de destino *(mantendo sempre a restrição de entradas iguais terem o mesmo valor de hash)*.

Funções de hash: data-science vs. segurança

- ❖ As funções de hash de cibersegurança cumprem com os objetivos das hashes de computação geral (i.e., data-science), mais alguns requisitos de resistência a pré-imagem, segunda pré-imagem, e colisão determinística.
- ❖ Esta resistência adicional consegue-se em troca de um custo computacional acrescido (*que normalmente cresce consoante os requisitos de resistência*).
- ❖ Quando o modelo de ameaças não obriga a resistir a atacantes naquele ponto específico, usam-se funções de hash não criptográficas por serem mais rápidas.

hashCode()

```
@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((model == null) ? 0 : model.hashCode());
    result = prime * result + year;
    result = prime * result + matricula;
    return result;
}
```

```
public class Car {

    static int count = 0;

    String model;
    int year;
    int matricula;
    int getYear() { return this.year; }

    Car(String m, int yr) {model=m; year=yr;
        this.matricula = Car.count++;
    }

    int getAge() {
        int currentYear =
            java.util.Calendar.getInstance().get(java.util.Calendar.YEAR);
        return getAge(currentYear);
    }

    int getAge(int currentYear) {
        return currentYear - year;
    }
}
```


Circulo.hashCode()

@Override

```
public int hashCode() {  
    final int prime = 31;  
    int result = prime  
        + ((centro == null) ? 0 : centro.hashCode());  
    long temp = Double.doubleToLongBits(raio);  
    result = prime * result + (int) (temp ^ (temp >>> 32));  
    // ^ Bitwise exclusive OR  
    // >>> Unsigned right shift  
    return result;  
}
```

Sobreposição (Overloading)

Sobreposição (Overloading)

```
class Car {  
    String model;  
    int year;  
    int getYear() { return this.year; }  
  
    Car(String m, int yr) {model=m; year=yr;}  
  
    int getAge() {  
        int currentYear = java.util.Calendar.getInstance().get(java.util.Calendar.YEAR);  
        return getAge(currentYear);  
    }  
  
    int getAge(int currentYear) {  
        return currentYear - year;  
    }  
}
```

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)

- ❖ 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)
- ❖ 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

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 ?

Problema com overloading

- ❖ 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

```
public class Test {  
    public void someFunction(String[] s) {  
        ...  
    }  
    public int someFunction(String[] b) {  
        ...  
    }  
  
    // ...  
    String[] someStrings = {"first string", "another string", "last"};  
    someFunction(someStrings);  
}
```

NÃO É POSSÍVEL

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?

Todos os atributos deverão ser privados.

```
public class Car {  
    static int count = 0;  
    String model;  
    int year;  
    int matricula;  
    int getYear() { return this.year; }  
  
    Car(String m, int yr) {model=m; year=yr;  
        this.matricula = Car.count++;  
    }  
  
    int getAge() {  
        int currentYear =  
            java.util.Calendar.getInstance().get(java.util.Calendar.  
                return getAge(currentYear);  
    }  
    int getAge(int currentYear) {  
        return currentYear - year;  
    }  
}
```

```
public class Car {  
    static int count = 0;  
    private String model;  
    private int year;  
    private int matricula;  
    int getYear() { return this.year; }  
  
    Car(String m, int yr) {model=m; year=yr;  
        this.matricula = Car.count++;  
    }  
}
```

Private ou
protected

```
public String getModel() {  
    return model;  
}  
  
public int getMatricula() {  
    return matricula;  
}  
  
public int getYear() {  
    return year;  
}
```

getters

```
public int getAge() {  
    int currentYear = java.util.Calendar.getInstance().get(java.util.Calendar.YEAR);  
    return getAge(currentYear);  
}  
  
public int getAge(int currentYear) {  
    return currentYear - year;  
}
```

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();  
    // ...  
};
```

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

Java tem pacotes

- ❖ Java fornece classes existentes em pacotes
- ❖ Podes definir packages

<https://docs.oracle.com/en/java/javase/11/docs/api/>

All Modules	Java SE	JDK	Other Modules
Module	Description		
java.base	Defines the foundational APIs of the Java SE Platform.		
java.compiler	Defines the Language Model, Annotation Processors, and Compiler.		
java.datatransfer	Defines the API for transferring data between applications.		
java.desktop	Defines the AWT and Swing user interface toolkit.		
java.instrument	Defines services that allow agents to instrument classes.		
java.logging	Defines the Java Logging API.		
java.management	Defines the Java Management Extensions (JMX) API.		
java.management.rmi	Defines the RMI connector for the Java Management Extensions.		
java.naming	Defines the Java Naming and Directory Interface.		
java.net.http	Defines the HTTP Client and WebSocket APIs.		
java.prefs	Defines the Preferences API.		
java.rmi	Defines the Remote Method Invocation (RMI) API.		
java.scripting	Defines the Scripting API.		
java.se	Defines the API of the Java SE Platform.		
java.security.jgss	Defines the Java binding of the IETF Generic Security Service API.		
java.security.sasl	Defines Java support for the IETF Simple Authentication and Security Layer (SASL).		
java.smartcardio	Defines the Java Smart Card I/O API.		
java.sql	Defines the JDBC API.		
java.sql.rowset	Defines the JDBC RowSet API.		

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();
```

Memória

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

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

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 pode ser utilizada desde o momento 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**

