

Aula 02

Classes, Objects e Pacotes

Como funcionam estes mecanismos em Java

Programação II, 2020-2021

v1.12, 20-02-2017

Classes

Novos Contextos de
Existência

Objects

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

1 Classes

Novos Contextos de Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

2 Pacotes (*Packages*)

Classes

Novos Contextos de Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

1 Classes

Novos Contextos de Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

2 Pacotes (*Packages*)

Classes

Novos Contextos de Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

Classe

Uma *classe* é uma entidade da linguagem que contém *métodos* e *atributos*, podendo também servir para definir novos tipos de dados (com os quais se podem instanciar *objectos*).

- Dentro da *classe* podemos definir *atributos* (ou *campos*) e *métodos* (ou *funções*).
- Os atributos permitem armazenar informação.
- Os métodos permitem implementar algoritmos.

```
public class Person {  
    String name;  
    static int personCount = 0;  
  
    String name() {  
        return name;  
    }  
  
    static void newPerson() {  
        personCount++;  
    }  
}
```

atributos

membros da classe

métodos

Classes

Novos Contextos de
Existência
Objectos
Encapsulamento
Sobreposição (*Overloading*)
Construtores
Resumo

Pacotes (*Packages*)

Classe

Uma *classe* é uma entidade da linguagem que contém *métodos* e *atributos*, podendo também servir para definir novos tipos de dados (com os quais se podem instanciar *objectos*).

- Dentro da *classe* podemos definir *atributos* (ou *campos*) e *métodos* (ou *funções*).
- Os atributos permitem armazenar informação.
- Os métodos permitem implementar algoritmos.

```
public class Person {  
    String name;  
    static int personCount = 0;  
  
    String name() {  
        return name;  
    }  
  
    static void newPerson() {  
        personCount++;  
    }  
}
```

atributos

membros da classe

métodos

Classes

Novos Contextos de Existência
Objectos
Encapsulamento
Sobreposição (*Overloading*)
Construtores
Resumo

Pacotes (*Packages*)

Classe

Uma *classe* é uma entidade da linguagem que contém *métodos* e *atributos*, podendo também servir para definir novos tipos de dados (com os quais se podem instanciar *objectos*).

- Dentro da *classe* podemos definir *atributos* (ou *campos*) e *métodos* (ou *funções*).
- Os atributos permitem armazenar informação.
- Os métodos permitem implementar algoritmos.

```
public class Person {  
    String name;  
    static int personCount = 0;  
  
    String name() {  
        return name;  
    }  
  
    static void newPerson() {  
        personCount++;  
    }  
}
```

atributos

membros da classe

métodos

Classes

Novos Contextos de Existência
Objectos
Encapsulamento
Sobreposição (*Overloading*)
Construtores
Resumo

Pacotes (*Packages*)

Classe

Uma *classe* é uma entidade da linguagem que contém *métodos* e *atributos*, podendo também servir para definir novos tipos de dados (com os quais se podem instanciar *objectos*).

- Dentro da *classe* podemos definir *atributos* (ou *campos*) e *métodos* (ou *funções*).
- Os atributos permitem armazenar informação.
- Os métodos permitem implementar algoritmos.

```
public class Person {  
    String name;  
    static int personCount = 0;  
  
    String name() {  
        return name;  
    }  
  
    static void newPerson() {  
        personCount++;  
    }  
}
```

atributos

membros da classe

métodos

Classes

Novos Contextos de Existência
Objectos
Encapsulamento
Sobreposição (*Overloading*)
Construtores
Resumo

Pacotes (*Packages*)

Classe

Uma *classe* é uma entidade da linguagem que contém *métodos* e *atributos*, podendo também servir para definir novos tipos de dados (com os quais se podem instanciar *objectos*).

- Dentro da *classe* podemos definir *atributos* (ou *campos*) e *métodos* (ou *funções*).
- Os atributos permitem armazenar informação.
- Os métodos permitem implementar algoritmos.

```
public class Person {  
    String name;  
    static int personCount = 0;  
  
    String name() {  
        return name;  
    }  
  
    static void newPerson() {  
        personCount++;  
    }  
}
```

Diagram illustrating the structure of the `Person` class:

- atributos**: `String name;` and `static int personCount = 0;`
- métodos**: `String name() { return name; }` and `static void newPerson() { personCount++; }`
- membros da classe**: The entire class body (attributes and methods).

Classes

Novos Contextos de Existência
Objectos
Encapsulamento
Sobreposição (*Overloading*)
Construtores
Resumo

Pacotes (*Packages*)

Classe

Uma *classe* é uma entidade da linguagem que contém *métodos* e *atributos*, podendo também servir para definir novos tipos de dados (com os quais se podem instanciar *objectos*).

- Dentro da *classe* podemos definir *atributos* (ou *campos*) e *métodos* (ou *funções*).
- Os atributos permitem armazenar informação.
- Os métodos permitem implementar algoritmos.

```
public class Person {  
    String name;  
    static int personCount = 0;  
  
    String name() {  
        return name;  
    }  
  
    static void newPerson() {  
        personCount++;  
    }  
}
```

Diagram illustrating the structure of the `Person` class:

- atributos**: `String name;` and `static int personCount = 0;`
- métodos**: `String name() { return name; }` and `static void newPerson() { personCount++; }`
- membros da classe**: The entire class body (attributes and methods).

Classes

Novos Contextos de Existência
Objectos
Encapsulamento
Sobreposição (*Overloading*)
Construtores
Resumo

Pacotes (*Packages*)

Classe

Uma *classe* é uma entidade da linguagem que contém *métodos* e *atributos*, podendo também servir para definir novos tipos de dados (com os quais se podem instanciar *objectos*).

- Dentro da *classe* podemos definir *atributos* (ou *campos*) e *métodos* (ou *funções*).
- Os atributos permitem armazenar informação.
- Os métodos permitem implementar algoritmos.

```
public class Person {  
    String name;  
    static int personCount = 0;  
  
    String name() {  
        return name;  
    }  
  
    static void newPerson() {  
        personCount++;  
    }  
}
```

atributos

membros da classe

métodos

Classes

Novos Contextos de
Existência
Objectos
Encapsulamento
Sobreposição (*Overloading*)
Construtores
Resumo

Pacotes (*Packages*)

Classe

Uma *classe* é uma entidade da linguagem que contém *métodos* e *atributos*, podendo também servir para definir novos tipos de dados (com os quais se podem instanciar *objectos*).

- Dentro da *classe* podemos definir *atributos* (ou *campos*) e *métodos* (ou *funções*).
- Os atributos permitem armazenar informação.
- Os métodos permitem implementar algoritmos.

```
public class Person {  
    String name;  
    static int personCount = 0;  
  
    String name() {  
        return name;  
    }  
  
    static void newPerson() {  
        personCount++;  
    }  
}
```

atributos

membros da classe

métodos

Classes

Novos Contextos de
Existência
Objectos
Encapsulamento
Sobreposição (*Overloading*)
Construtores
Resumo

Pacotes (*Packages*)

Classe

Uma *classe* é uma entidade da linguagem que contém *métodos* e *atributos*, podendo também servir para definir novos tipos de dados (com os quais se podem instanciar *objectos*).

- Dentro da *classe* podemos definir *atributos* (ou *campos*) e *métodos* (ou *funções*).
- Os atributos permitem armazenar informação.
- Os métodos permitem implementar algoritmos.

```
public class Person {  
    String name;  
    static int personCount = 0;  
  
    String name() {  
        return name;  
    }  
  
    static void newPerson() {  
        personCount++;  
    }  
}
```

atributos

membros da classe

métodos

Classes

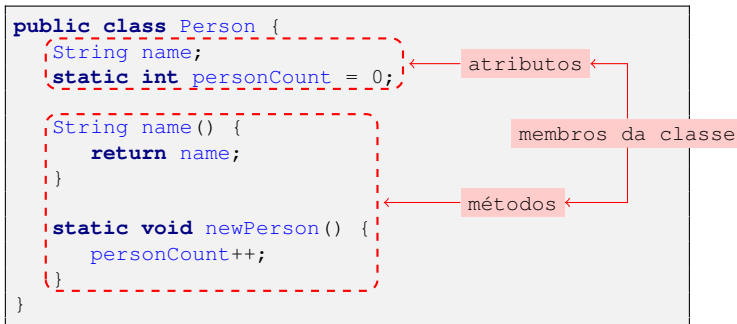
Novos Contextos de
Existência
Objects
Encapsulamento
Sobreposição (*Overloading*)
Construtores
Resumo

Pacotes (*Packages*)

Classe

Uma *classe* é uma entidade da linguagem que contém *métodos* e *atributos*, podendo também servir para definir novos tipos de dados (com os quais se podem instanciar *objectos*).

- Dentro da *classe* podemos definir *atributos* (ou *campos*) e *métodos* (ou *funções*).
- Os atributos permitem armazenar informação.
- Os métodos permitem implementar algoritmos.



Classes

Novos Contextos de Existência
Objectos
Encapsulamento
Sobreposição (*Overloading*)
Construtores
Resumo

Pacotes (*Packages*)

Classes

Novos Contextos de Existência

Objects

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

A classe define dois novos contextos de existência:

- 1 Contexto de classe (ou estático);
- 2 Contexto de objecto (ou de instância).

Contexto de classe:

```
public class C {  
    static int a;  
  
    static void p() {  
        a++; // ⇔ C.a++;  
    }  
  
    static boolean f() {  
        ...  
    }  
}
```

```
public class Test  
{  
    public static  
    void main(String[] args) {  
        C.a = 10;  
        C.p();  
        if (C.f()) {  
            ...  
        }  
    }  
}
```

Classes

Novos Contextos de Existência

Objects

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

A classe define dois novos contextos de existência:

- 1 Contexto de classe (ou estático);
- 2 Contexto de objecto (ou de instância).

Contexto de classe:

```
public class C {  
    static int a;  
  
    static void p() {  
        a++; // ⇔ C.a++;  
    }  
  
    static boolean f() {  
        ...  
    }  
}
```

```
public class Test  
{  
    public static  
    void main(String[] args) {  
        C.a = 10;  
        C.p();  
        if (C.f()) {  
            ...  
        }  
    }  
}
```

Classes

Novos Contextos de Existência

Objects

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

A classe define dois novos contextos de existência:

- 1 Contexto de classe (ou estático);
- 2 Contexto de objecto (ou de instância).

Contexto de classe:

```
public class C {  
    static int a;  
  
    static void p() {  
        a++; // ⇔ C.a++;  
    }  
  
    static boolean f() {  
        ...  
    }  
}
```

```
public class Test  
{  
    public static  
    void main(String[] args) {  
        C.a = 10;  
        C.p();  
        if (C.f()) {  
            ...  
        }  
    }  
}
```


Classes

Novos Contextos de Existência

Objects

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

A classe define dois novos contextos de existência:

- 1 Contexto de classe (ou estático);
- 2 Contexto de objecto (ou de instância).

Contexto de classe:

```
public class C {  
    static int a;  
  
    static void p() {  
        a++; // ⇔ C.a++;  
    }  
  
    static boolean f() {  
        ...  
    }  
}
```

```
public class Test  
{  
    public static  
    void main(String[] args) {  
        C.a = 10;  
        C.p();  
        if (C.f()) {  
            ...  
        }  
    }  
}
```

Classes

Novos Contextos de Existência

Objects

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

A classe define dois novos contextos de existência:

- 1 Contexto de classe (ou estático);
- 2 Contexto de objecto (ou de instância).

Contexto de classe:

```
public class C {  
    static int a;  
  
    static void p() {  
        a++; // ⇔ C.a++;  
    }  
  
    static boolean f() {  
        ...  
    }  
}
```

```
public class Test  
{  
    public static  
    void main(String[] args) {  
        C.a = 10;  
        C.p();  
        if (C.f()) {  
            ...  
        }  
    }  
}
```

Classes

Novos Contextos de Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

A classe define dois novos contextos de existência:

- 1 Contexto de classe (ou estático);
- 2 Contexto de objecto (ou de instância).

Contexto de classe:

```
public class C {  
    static int a;  
  
    static void p() {  
        a++; // ⇔ C.a++;  
    }  
  
    static boolean f() {  
        ...  
    }  
}
```

```
public class Test  
{  
    public static  
    void main(String[] args) {  
        C.a = 10;  
        C.p();  
        if (C.f()) {  
            ...  
        }  
    }  
}
```

Novos Contextos de Existência: Objectos

Contexto de objecto:

```
public class C {  
    int a;  
  
    void p() {  
        a++; // ⇔ this.a++;  
    }  
  
    boolean f() {  
        ...  
    }  
}
```

```
public class Test  
{  
    public static  
    void main(String[] args) {  
        // criar um objecto:  
        C x = new C();  
        x.a = 10;  
        x.p();  
        if (x.f()) {  
            ...  
        }  
        x = null;  
        // objecto x deixa de  
        // ser referenciável  
    }  
}
```

Classes

Novos Contextos de Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

Contexto de objecto:

```
public class C {  
    int a;  
  
    void p() {  
        a++; // ⇔ this.a++;  
    }  
  
    boolean f() {  
        ...  
    }  
}
```

```
public class Test  
{  
    public static  
    void main(String[] args) {  
        // criar um objecto:  
        C x = new C();  
        x.a = 10;  
        x.p();  
        if (x.f()) {  
            ...  
        }  
        x = null;  
        // objecto x deixa de  
        // ser referenciável  
    }  
}
```

Classes

Novos Contextos de Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

Contexto de objecto:

```
public class C {  
    int a;  
  
    void p() {  
        a++; // ⇔ this.a++;  
    }  
  
    boolean f() {  
        ...  
    }  
}
```

```
public class Test  
{  
    public static  
    void main(String[] args) {  
        // criar um objecto:  
        C x = new C();  
        x.a = 10;  
        x.p();  
        if (x.f()) {  
            ...  
        }  
        x = null;  
        // objecto x deixa de  
        // ser referenciável  
    }  
}
```

Classes

Novos Contextos de
Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

Novos Contextos de Existência: Classes e Objectos

- Contexto de classe (*static*):

- Partilhamos o mesmo espaço de memória com todos os objectos da JVM.
- Os atributos das classes são armazenados e partilhados por todos os objectos da classe.
- Os atributos da classe não possuem memória por objecto, apenas possuem uma memória associada à classe.
- Os métodos *static* têm acesso directo ao conteúdo *static* da sua classe.

- Contexto de objecto (*non static*):

- Partilhamos o mesmo espaço de memória com todos os objectos da JVM.
- Cada objecto tem um conjunto próprio de atributos *non static*.
- Os métodos *non static* não podem aceder às variáveis *static* sem um objecto determinado e têm acesso directo ao conteúdo desse objecto.

- Uma classe pode ter membros *static* e não *static*.

Classes

Novos Contextos de
Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

Classes

Novos Contextos de
Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

- Contexto de classe (*static*):
 - Atributos e métodos de classe existem sempre, haja objectos ou não.
 - Os atributos de classe são acessíveis e partilhados por todos os objetos da classe.
 - Os atributos de classe não ocupam memória nos objetos, apenas ocupam uma memória associada à classe.
 - Os métodos *static* têm acesso direto apenas ao contexto *static* da sua classe.
- Contexto de objecto (*non static*):
 - Atributos e métodos só existem enquanto o respectivo objecto existir.
 - Cada objeto tem um conjunto próprio de atributos *non static*.
 - Os métodos *non static* são necessariamente invocados sobre um objeto determinado e têm acesso direto a todo o contexto desse objeto.
- Uma classe pode ter membros *static* e não *static*.

Classes

Novos Contextos de
Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

- Contexto de classe (*static*):
 - Atributos e métodos de classe existem sempre, haja objectos ou não.
 - Os atributos de classe são acessíveis e partilhados por todos os objetos da classe.
 - Os atributos de classe não ocupam memória nos objetos, apenas ocupam uma memória associada à classe.
 - Os métodos *static* têm acesso direto apenas ao contexto *static* da sua classe.
- Contexto de objecto (*non static*):
 - Atributos e métodos só existem enquanto o respectivo objecto existir.
 - Cada objeto tem um conjunto próprio de atributos *non static*.
 - Os métodos *non static* são necessariamente invocados sobre um objeto determinado e têm acesso direto a todo o contexto desse objeto.
- Uma classe pode ter membros *static* e não *static*.

Classes

Novos Contextos de
Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

- Contexto de classe (*static*):
 - Atributos e métodos de classe existem sempre, haja objectos ou não.
 - Os atributos de classe são acessíveis e partilhados por todos os objetos da classe.
 - Os atributos de classe não ocupam memória nos objetos, apenas ocupam uma memória associada à classe.
 - Os métodos *static* têm acesso direto apenas ao contexto *static* da sua classe.
- Contexto de objecto (*non static*):
 - Atributos e métodos só existem enquanto o respectivo objecto existir.
 - Cada objeto tem um conjunto próprio de atributos *non static*.
 - Os métodos *non static* são necessariamente invocados sobre um objeto determinado e têm acesso direto a todo o contexto desse objeto.
- Uma classe pode ter membros *static* e não *static*.

Classes

Novos Contextos de
Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

- Contexto de classe (*static*):
 - Atributos e métodos de classe existem sempre, haja objectos ou não.
 - Os atributos de classe são acessíveis e partilhados por todos os objetos da classe.
 - Os atributos de classe não ocupam memória nos objetos, apenas ocupam uma memória associada à classe.
 - Os métodos *static* têm acesso direto apenas ao contexto *static* da sua classe.
- Contexto de objecto (*non static*):
 - Atributos e métodos só existem enquanto o respectivo objecto existir.
 - Cada objeto tem um conjunto próprio de atributos *non static*.
 - Os métodos *non static* são necessariamente invocados sobre um objeto determinado e têm acesso direto a todo o contexto desse objeto.
- Uma classe pode ter membros *static* e não *static*.

Classes

Novos Contextos de
Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

- Contexto de classe (*static*):
 - Atributos e métodos de classe existem sempre, haja objectos ou não.
 - Os atributos de classe são acessíveis e partilhados por todos os objetos da classe.
 - Os atributos de classe não ocupam memória nos objetos, apenas ocupam uma memória associada à classe.
 - Os métodos *static* têm acesso direto apenas ao contexto *static* da sua classe.
- Contexto de objecto (*non static*):
 - Atributos e métodos só existem enquanto o respectivo objecto existir.
 - Cada objeto tem um conjunto próprio de atributos *non static*.
 - Os métodos *non static* são necessariamente invocados sobre um objeto determinado e têm acesso direto a todo o contexto desse objeto.
- Uma classe pode ter membros *static* e não *static*.

Classes

Novos Contextos de
Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

- Contexto de classe (*static*):
 - Atributos e métodos de classe existem sempre, haja objectos ou não.
 - Os atributos de classe são acessíveis e partilhados por todos os objetos da classe.
 - Os atributos de classe não ocupam memória nos objetos, apenas ocupam uma memória associada à classe.
 - Os métodos *static* têm acesso direto apenas ao contexto *static* da sua classe.
- Contexto de objecto (*non static*):
 - Atributos e métodos só existem enquanto o respectivo objecto existir.
 - Cada objeto tem um conjunto próprio de atributos *non static*.
 - Os métodos *non static* são necessariamente invocados sobre um objeto determinado e têm acesso direto a todo o contexto desse objeto.
- Uma classe pode ter membros *static* e não *static*.

Classes

Novos Contextos de
Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

- Contexto de classe (*static*):
 - Atributos e métodos de classe existem sempre, haja objectos ou não.
 - Os atributos de classe são acessíveis e partilhados por todos os objetos da classe.
 - Os atributos de classe não ocupam memória nos objetos, apenas ocupam uma memória associada à classe.
 - Os métodos *static* têm acesso direto apenas ao contexto *static* da sua classe.
- Contexto de objecto (*non static*):
 - Atributos e métodos só existem enquanto o respectivo objecto existir.
 - Cada objeto tem um conjunto próprio de atributos *non static*.
 - Os métodos *non static* são necessariamente invocados sobre um objeto determinado e têm acesso direto a todo o contexto desse objeto.
- Uma classe pode ter membros *static* e não *static*.

Classes

Novos Contextos de
Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

- Contexto de classe (*static*):
 - Atributos e métodos de classe existem sempre, haja objectos ou não.
 - Os atributos de classe são acessíveis e partilhados por todos os objetos da classe.
 - Os atributos de classe não ocupam memória nos objetos, apenas ocupam uma memória associada à classe.
 - Os métodos *static* têm acesso direto apenas ao contexto *static* da sua classe.
- Contexto de objecto (*non static*):
 - Atributos e métodos só existem enquanto o respectivo objecto existir.
 - Cada objeto tem um conjunto próprio de atributos *non static*.
 - Os métodos *non static* são necessariamente invocados sobre um objeto determinado e têm acesso direto a todo o contexto desse objeto.
- Uma classe pode ter membros *static* e não *static*.

Classes

Novos Contextos de
Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

- Contexto de classe (*static*):
 - Atributos e métodos de classe existem sempre, haja objectos ou não.
 - Os atributos de classe são acessíveis e partilhados por todos os objetos da classe.
 - Os atributos de classe não ocupam memória nos objetos, apenas ocupam uma memória associada à classe.
 - Os métodos *static* têm acesso direto apenas ao contexto *static* da sua classe.
- Contexto de objecto (*non static*):
 - Atributos e métodos só existem enquanto o respectivo objecto existir.
 - Cada objeto tem um conjunto próprio de atributos *non static*.
 - Os métodos *non static* são necessariamente invocados sobre um objeto determinado e têm acesso direto a todo o contexto desse objeto.
- Uma classe pode ter membros *static* e não *static*.

Classes

Novos Contextos de
Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

- Contexto de classe (*static*):
 - Atributos e métodos de classe existem sempre, haja objectos ou não.
 - Os atributos de classe são acessíveis e partilhados por todos os objetos da classe.
 - Os atributos de classe não ocupam memória nos objetos, apenas ocupam uma memória associada à classe.
 - Os métodos *static* têm acesso direto apenas ao contexto *static* da sua classe.
- Contexto de objecto (*non static*):
 - Atributos e métodos só existem enquanto o respectivo objecto existir.
 - Cada objeto tem um conjunto próprio de atributos *non static*.
 - Os métodos *non static* são necessariamente invocados sobre um objeto determinado e têm acesso direto a todo o contexto desse objeto.
- Uma classe pode ter membros *static* e não *static*.

Exemplo de classe

Classes

Novos Contextos de Existência

Objects

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

```
public class Aluno {  
    Aluno(String nome) {  
        count++; nmec = count;  
        this.nome = nome  
    }  
  
    String nome() { return nome; }  
  
    String curso() { return curso; }  
  
    int nmec() { return nmec; }  
  
    void defineCurso(String curso) {  
        this.curso = curso;  
    }  
  
    String nome;  
    String curso;  
    int nmec;  
  
    static int count; // = 0;  
  
    static {  
        count = 0;  
    }  
}
```

construtor: procedimento de inicialização do objecto, executado aquando da sua criação.

métodos de objecto: só podem ser invocados através de um objecto.

atributos de objecto: definem o estado do objecto. Este estado não é partilhado com outros objectos.

atributo estático: não é preciso objectos para ser utilizado. É partilhado por todos os objectos da classe.

construtor da classe: código de inicialização do contexto estático da classe, executado uma única vez, quando a classe é carregada.

Exemplo de classe

Classes

Novos Contextos de Existência

Objects

Encapsulamento

Sobreposição (Overloading)

Construtores

Resumo

Pacotes (Packages)

```
public class Aluno {  
    Aluno(String nome) {  
        count++; nmec = count;  
        this.nome = nome  
    }  
  
    String nome() { return nome; }  
  
    String curso() { return curso; }  
  
    int nmec() { return nmec; }  
  
    void defineCurso(String curso) {  
        this.curso = curso;  
    }  
  
    String nome;  
    String curso;  
    int nmec;  
  
    static int count; // = 0;  
  
    static {  
        count = 0;  
    }  
}
```

construtor: procedimento de inicialização do objecto, executado aquando da sua criação.

métodos de objecto: só podem ser invocados através de um objecto.

atributos de objecto: definem o estado do objecto. Este estado não é partilhado com outros objectos.

atributo estático: não é preciso objectos para ser utilizado. É partilhado por todos os objectos da classe.

construtor da classe: código de inicialização do contexto estático da classe, executado uma única vez, quando a classe é carregada.

Exemplo de classe

Classes

Novos Contextos de Existência

Objects

Encapsulamento

Sobreposição (Overloading)

Construtores

Resumo

Pacotes (Packages)

```
public class Aluno {  
    Aluno(String nome) {  
        count++; nmec = count;  
        this.nome = nome  
    }  
  
    String nome() { return nome; }  
  
    String curso() { return curso; }  
  
    int nmec() { return nmec; }  
  
    void defineCurso(String curso) {  
        this.curso = curso;  
    }  
  
    String nome;  
    String curso;  
    int nmec;  
  
    static int count; // = 0;  
  
    static {  
        count = 0;  
    }  
}
```

construtor: procedimento de inicialização do objecto, executado aquando da sua criação.

métodos de objecto: só podem ser invocados através de um objecto.

atributos de objecto: definem o estado do objecto. Este estado não é partilhado com outros objectos.

atributo estático: não é preciso objectos para ser utilizado. É partilhado por todos os objectos da classe.

construtor da classe: código de inicialização do contexto estático da classe, executado uma única vez, quando a classe é carregada.

Exemplo de classe

Classes

Novos Contextos de Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

```
public class Aluno {  
    Aluno(String nome) {  
        count++; nmec = count;  
        this.nome = nome  
    }  
  
    String nome() { return nome; }  
  
    String curso() { return curso; }  
  
    int nmec() { return nmec; }  
  
    void defineCurso(String curso) {  
        this.curso = curso;  
    }  
  
    String nome;  
    String curso;  
    int nmec;  
  
    static int count; // = 0;  
  
    static {  
        count = 0;  
    }  
}
```

construtor: procedimento de inicialização do objecto, executado aquando da sua criação.

métodos de objecto: só podem ser invocados através de um objecto.

atributos de objecto: definem o estado do objecto. Este estado não é partilhado com outros objectos.

atributo estático: não é preciso objectos para ser utilizado. É partilhado por todos os objectos da classe.

construtor da classe: código de inicialização do contexto estático da classe, executado uma única vez, quando a classe é carregada.

Exemplo de classe

Classes

Novos Contextos de Existência

Objects

Encapsulamento

Sobreposição (Overloading)

Construtores

Resumo

Pacotes (Packages)

```
public class Aluno {  
    Aluno(String nome) {  
        count++; nmec = count;  
        this.nome = nome  
    }  
  
    String nome() { return nome; }  
  
    String curso() { return curso; }  
  
    int nmec() { return nmec; }  
  
    void defineCurso(String curso) {  
        this.curso = curso;  
    }  
  
    String nome;  
    String curso;  
    int nmec;  
  
    static int count; // = 0;  
  
    static {  
        count = 0;  
    }  
}
```

construtor: procedimento de inicialização do objecto, executado aquando da sua criação.

métodos de objecto: só podem ser invocados através de um objecto.

atributos de objecto: definem o estado do objecto. Este estado não é partilhado com outros objectos.

atributo estático: não é preciso objectos para ser utilizado. É partilhado por todos os objectos da classe.

construtor da classe: código de inicialização do contexto estático da classe, executado uma única vez, quando a classe é carregada.

Exemplo de classe

Classes

Novos Contextos de Existência

Objects

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

```
public class Aluno {  
    Aluno(String nome) {  
        count++; nmec = count;  
        this.nome = nome  
    }  
  
    String nome() { return nome; }  
  
    String curso() { return curso; }  
  
    int nmec() { return nmec; }  
  
    void defineCurso(String curso) {  
        this.curso = curso;  
    }  
  
    String nome;  
    String curso;  
    int nmec;  
  
    static int count; // = 0;  
  
    static {  
        count = 0;  
    }  
}
```

construtor: procedimento de inicialização do objecto, executado aquando da sua criação.

métodos de objecto: só podem ser invocados através de um objecto.

atributos de objecto: definem o estado do objecto. Este estado não é partilhado com outros objectos.

atributo estático: não é preciso objectos para ser utilizado. É partilhado por todos os objectos da classe.

construtor da classe: código de inicialização do contexto estático da classe, executado uma única vez, quando a classe é carregada.

Exemplo de classe

Classes

Novos Contextos de Existência

Objects

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

```
public class Aluno {  
    Aluno(String nome) {  
        count++; nmec = count;  
        this.nome = nome  
    }  
  
    String nome() { return nome; }  
  
    String curso() { return curso; }  
  
    int nmec() { return nmec; }  
  
    void defineCurso(String curso) {  
        this.curso = curso;  
    }  
  
    String nome;  
    String curso;  
    int nmec;  
  
    static int count; // = 0;  
  
    static {  
        count = 0;  
    }  
}
```

construtor: procedimento de inicialização do objecto, executado aquando da sua criação.

métodos de objecto: só podem ser invocados através de um objecto.

atributos de objecto: definem o estado do objecto. Este estado não é partilhado com outros objectos.

atributo estático: não é preciso objectos para ser utilizado. É partilhado por todos os objectos da classe.

construtor da classe: código de inicialização do contexto estático da classe, executado uma única vez, quando a classe é carregada.

Invocação de Métodos (Mensagens)

- A invocação de um método pode ser *interna* ou *externa*;
- A invocação externa é sempre efectuada através da *notação de ponto*:

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

- A invocação de um método de um objecto pode ser vista como o envio de uma mensagem (pedido de um serviço) ao objecto: *"DETI, abre a tua porta!"*
 - 1. O receptor da mensagem é o indicado à esquerda do ponto;
 - 2. O tipo de mensagem é o nome do método;
 - 3. Outros dados eventualmente necessários serão argumentos;
 - 4. Dentro do método, o objecto receptor funciona como um *parâmetro implícito* (*this*);
 - 5. *this* é um identificador reservado, que tem uma referência para o objecto receptor da invocação e que se pode usar apenas no corpo de um método de instância.
- No caso de métodos de classe (*static*), o receptor pode ser o nome da classe, e.g.: `Math.sqrt()`.
- O acesso a atributos segue regras idênticas.

Classes

Novos Contextos de
Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

Invocação de Métodos (Mensagens)

- A invocação de um método pode ser *interna* ou *externa*;
- A invocação externa é sempre efectuada através da *notação de ponto*:

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

- A invocação de um método de um objecto pode ser vista como o envio de uma mensagem (pedido de um serviço) ao objecto: *“DETI, abre a tua porta!”*
 - O receptor da mensagem é o indicado à esquerda do ponto.
 - O tipo de mensagem é o nome do método.
 - Outros dados eventualmente necessários serão argumentos.
 - Dentro do método, o objecto receptor funciona como um parâmetro implícito (*this*).
 - *this* é um identificador reservado, que tem uma referência para o objecto receptor da invocação e que se pode usar apenas no corpo de um método de instância.
- No caso de métodos de classe (*static*), o receptor pode ser o nome da classe, e.g.: `Math.sqrt()`.
- O acesso a atributos segue regras idênticas.

Classes

Novos Contextos de
Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

Invocação de Métodos (Mensagens)

- A invocação de um método pode ser *interna* ou *externa*;
- A invocação externa é sempre efectuada através da *notação de ponto*:

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

- A invocação de um método de um objecto pode ser vista como o envio de uma mensagem (pedido de um serviço) ao objecto: “*DETI, abre a tua porta!*”
 - O receptor da mensagem é o indicado à esquerda do ponto.
 - O tipo de mensagem é o nome do método.
 - Outros dados eventualmente necessários serão argumentos.
 - Dentro do método, o objecto receptor funciona como um parâmetro implícito (*this*).
 - *this* é um identificador reservado, que tem uma referência para o objecto receptor da invocação e que se pode usar apenas no corpo de um método de instância.
- No caso de métodos de classe (*static*), o receptor pode ser o nome da classe, e.g.: `Math.sqrt()`.
- O acesso a atributos segue regras idênticas.

Classes

Novos Contextos de
Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

Invocação de Métodos (Mensagens)

- A invocação de um método pode ser *interna* ou *externa*;
- A invocação externa é sempre efectuada através da *notação de ponto*:

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

- A invocação de um método de um objecto pode ser vista como o envio de uma mensagem (pedido de um serviço) ao objecto: “*DETI, abre a tua porta!*”
 - O receptor da mensagem é o indicado à esquerda do ponto.
 - O tipo de mensagem é o nome do método.
 - Outros dados eventualmente necessários serão argumentos.
 - Dentro do método, o objecto receptor funciona como um parâmetro implícito (`this`).
 - `this` é um identificador reservado, que tem uma referência para o objecto receptor da invocação e que se pode usar apenas no corpo de um método de instância.
- No caso de métodos de classe (`static`), o receptor pode ser o nome da classe, e.g.: `Math.sqrt()`.
- O acesso a atributos segue regras idênticas.

Classes

Novos Contextos de
Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

Invocação de Métodos (Mensagens)

- A invocação de um método pode ser *interna* ou *externa*;
- A invocação externa é sempre efectuada através da *notação de ponto*:

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

- A invocação de um método de um objecto pode ser vista como o envio de uma mensagem (pedido de um serviço) ao objecto: “*DETI, abre a tua porta!*”
 - O receptor da mensagem é o indicado à esquerda do ponto.
 - O tipo de mensagem é o nome do método.
 - Outros dados eventualmente necessários serão argumentos.
 - Dentro do método, o objecto receptor funciona como um parâmetro implícito (`this`).
 - `this` é um identificador reservado, que tem uma referência para o objecto receptor da invocação e que se pode usar apenas no corpo de um método de instância.
- No caso de métodos de classe (`static`), o receptor pode ser o nome da classe, e.g.: `Math.sqrt()`.
- O acesso a atributos segue regras idênticas.

Classes

Novos Contextos de
Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

Invocação de Métodos (Mensagens)

- A invocação de um método pode ser *interna* ou *externa*;
- A invocação externa é sempre efectuada através da *notação de ponto*:

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

- A invocação de um método de um objecto pode ser vista como o envio de uma mensagem (pedido de um serviço) ao objecto: “*DETI, abre a tua porta!*”
 - O receptor da mensagem é o indicado à esquerda do ponto.
 - O tipo de mensagem é o nome do método.
 - Outros dados eventualmente necessários serão argumentos.
 - Dentro do método, o objecto receptor funciona como um parâmetro implícito (`this`).
 - `this` é um identificador reservado, que tem uma referência para o objecto receptor da invocação e que se pode usar apenas no corpo de um método de instância.
- No caso de métodos de classe (`static`), o receptor pode ser o nome da classe, e.g.: `Math.sqrt()`.
- O acesso a atributos segue regras idênticas.

Classes

Novos Contextos de
Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

Invocação de Métodos (Mensagens)

- A invocação de um método pode ser *interna* ou *externa*;
- A invocação externa é sempre efectuada através da *notação de ponto*:

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

- A invocação de um método de um objecto pode ser vista como o envio de uma mensagem (pedido de um serviço) ao objecto: “*DETI, abre a tua porta!*”
 - O receptor da mensagem é o indicado à esquerda do ponto.
 - O tipo de mensagem é o nome do método.
 - Outros dados eventualmente necessários serão argumentos.
 - Dentro do método, o objecto receptor funciona como um parâmetro implícito (`this`).
 - `this` é um identificador reservado, que tem uma referência para o objecto receptor da invocação e que se pode usar apenas no corpo de um método de instância.
- No caso de métodos de classe (`static`), o receptor pode ser o nome da classe, e.g.: `Math.sqrt()`.
- O acesso a atributos segue regras idênticas.

Classes

Novos Contextos de
Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

Invocação de Métodos (Mensagens)

- A invocação de um método pode ser *interna* ou *externa*;
- A invocação externa é sempre efectuada através da *notação de ponto*:

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

- A invocação de um método de um objecto pode ser vista como o envio de uma mensagem (pedido de um serviço) ao objecto: “*DETI, abre a tua porta!*”
 - O receptor da mensagem é o indicado à esquerda do ponto.
 - O tipo de mensagem é o nome do método.
 - Outros dados eventualmente necessários serão argumentos.
 - Dentro do método, o objecto receptor funciona como um parâmetro implícito (`this`).
 - `this` é um identificador reservado, que tem uma referência para o objecto receptor da invocação e que se pode usar apenas no corpo de um método de instância.
- No caso de métodos de classe (`static`), o receptor pode ser o nome da classe, e.g.: `Math.sqrt()`.
- O acesso a atributos segue regras idênticas.

Classes

Novos Contextos de
Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

Invocação de Métodos (Mensagens)

- A invocação de um método pode ser *interna* ou *externa*;
- A invocação externa é sempre efectuada através da *notação de ponto*:

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

- A invocação de um método de um objecto pode ser vista como o envio de uma mensagem (pedido de um serviço) ao objecto: “*DETI, abre a tua porta!*”
 - O receptor da mensagem é o indicado à esquerda do ponto.
 - O tipo de mensagem é o nome do método.
 - Outros dados eventualmente necessários serão argumentos.
 - Dentro do método, o objecto receptor funciona como um parâmetro implícito (`this`).
 - `this` é um identificador reservado, que tem uma referência para o objecto receptor da invocação e que se pode usar apenas no corpo de um método de instância.
- No caso de métodos de classe (`static`), o receptor pode ser o nome da classe, e.g.: `Math.sqrt()`.
- O acesso a atributos segue regras idênticas.

Classes

Novos Contextos de
Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

Invocação de Métodos (Mensagens)

- A invocação de um método pode ser *interna* ou *externa*;
- A invocação externa é sempre efectuada através da *notação de ponto*:

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

- A invocação de um método de um objecto pode ser vista como o envio de uma mensagem (pedido de um serviço) ao objecto: “*DETI, abre a tua porta!*”
 - O receptor da mensagem é o indicado à esquerda do ponto.
 - O tipo de mensagem é o nome do método.
 - Outros dados eventualmente necessários serão argumentos.
 - Dentro do método, o objecto receptor funciona como um parâmetro implícito (`this`).
 - `this` é um identificador reservado, que tem uma referência para o objecto receptor da invocação e que se pode usar apenas no corpo de um método de instância.
- No caso de métodos de classe (`static`), o receptor pode ser o nome da classe, e.g.: `Math.sqrt()`.
- O acesso a atributos segue regras idênticas.

Classes

Novos Contextos de
Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

Invocação de Métodos (Mensagens)

- A invocação de um método pode ser *interna* ou *externa*;
- A invocação externa é sempre efectuada através da *notação de ponto*:

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

- A invocação de um método de um objecto pode ser vista como o envio de uma mensagem (pedido de um serviço) ao objecto: “*DETI, abre a tua porta!*”
 - O receptor da mensagem é o indicado à esquerda do ponto.
 - O tipo de mensagem é o nome do método.
 - Outros dados eventualmente necessários serão argumentos.
 - Dentro do método, o objecto receptor funciona como um parâmetro implícito (`this`).
 - `this` é um identificador reservado, que tem uma referência para o objecto receptor da invocação e que se pode usar apenas no corpo de um método de instância.
- No caso de métodos de classe (`static`), o receptor pode ser o nome da classe, e.g.: `Math.sqrt()`.
- O acesso a atributos segue regras idênticas.

Classes

Novos Contextos de
Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

Classes

Novos Contextos de

Existência

Objects

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

- Permite que a classe defina a política de acesso exterior aos seus membros autorizando, ou proibindo, esse acesso;

- Em Java, os modificadores de controlo de acesso que podemos usar são os seguintes:

`public` - indica que o membro pode ser usado em qualquer classe;

`protected` - o membro só pode ser usado por classes derivadas (conceito estudado noutra disciplina) ou do mesmo pacote;

`private` - o membro só pode ser usado em classes do mesmo pacote;

`default` - o membro só pode ser usado na própria classe.

- Mais informação sobre controlo de acesso no Java Tutorial.

Classes

Novos Contextos de

Existência

Objects

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

- Permite que a classe defina a política de acesso exterior aos seus membros autorizando, ou proibindo, esse acesso;
- Em Java, os modificadores de controlo de acesso que podemos usar são os seguintes:

`public` - indica que o membro pode ser usado em qualquer classe;

`protected` - o membro só pode ser usado por *classes derivadas* (conceito estudado noutra disciplina) ou do mesmo *package*;

(nada) - o membro só pode ser usado em classes do mesmo *package*;

`private` - o membro só pode ser usado na própria classe.

- Mais informação sobre [controlo de acesso no Java Tutorial](#).

Classes

Novos Contextos de
Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

- Permite que a classe defina a política de acesso exterior aos seus membros autorizando, ou proibindo, esse acesso;
- Em Java, os modificadores de controlo de acesso que podemos usar são os seguintes:

`public` - indica que o membro pode ser usado em qualquer classe;

`protected` - o membro só pode ser usado por *classes derivadas* (conceito estudado noutra disciplina) ou do mesmo *package*;

(nada) - o membro só pode ser usado em classes do mesmo *package*;

`private` - o membro só pode ser usado na própria classe.

- Mais informação sobre [controlo de acesso no Java Tutorial](#).

Classes

Novos Contextos de

Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

- Permite que a classe defina a política de acesso exterior aos seus membros autorizando, ou proibindo, esse acesso;
- Em Java, os modificadores de controlo de acesso que podemos usar são os seguintes:

`public` - indica que o membro pode ser usado em qualquer classe;

`protected` - o membro só pode ser usado por *classes derivadas* (conceito estudado noutra disciplina) ou do mesmo *package*;

(nada) - o membro só pode ser usado em classes do mesmo *package*;

`private` - o membro só pode ser usado na própria classe.

- Mais informação sobre [controlo de acesso no Java Tutorial](#).

Classes

Novos Contextos de

Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

- Permite que a classe defina a política de acesso exterior aos seus membros autorizando, ou proibindo, esse acesso;
- Em `Java`, os modificadores de controlo de acesso que podemos usar são os seguintes:

`public` - indica que o membro pode ser usado em qualquer classe;

`protected` - o membro só pode ser usado por *classes derivadas* (conceito estudado noutra disciplina) ou do mesmo *package*;

(nada) - o membro só pode ser usado em classes do mesmo *package*;

`private` - o membro só pode ser usado na própria classe.

- Mais informação sobre [controlo de acesso no Java Tutorial](#).

Classes

Novos Contextos de

Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

- Permite que a classe defina a política de acesso exterior aos seus membros autorizando, ou proibindo, esse acesso;
- Em `Java`, os modificadores de controlo de acesso que podemos usar são os seguintes:

`public` - indica que o membro pode ser usado em qualquer classe;

`protected` - o membro só pode ser usado por *classes derivadas* (conceito estudado noutra disciplina) ou do mesmo *package*;

(nada) - o membro só pode ser usado em classes do mesmo *package*;

`private` - o membro só pode ser usado na própria classe.

- Mais informação sobre [controlo de acesso no Java Tutorial](#).

Classes

Novos Contextos de

Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

- Permite que a classe defina a política de acesso exterior aos seus membros autorizando, ou proibindo, esse acesso;
- Em Java, os modificadores de controlo de acesso que podemos usar são os seguintes:

`public` - indica que o membro pode ser usado em qualquer classe;

`protected` - o membro só pode ser usado por *classes derivadas* (conceito estudado noutra disciplina) ou do mesmo *package*;

(nada) - o membro só pode ser usado em classes do mesmo *package*;

`private` - o membro só pode ser usado na própria classe.

- Mais informação sobre [controlo de acesso no Java Tutorial](#).

Classes

Novos Contextos de

Existência

Objects

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

- Permite que a classe defina a política de acesso exterior aos seus membros autorizando, ou proibindo, esse acesso;
- Em `Java`, os modificadores de controlo de acesso que podemos usar são os seguintes:

`public` - indica que o membro pode ser usado em qualquer classe;

`protected` - o membro só pode ser usado por *classes derivadas* (conceito estudado noutra disciplina) ou do mesmo *package*;

(nada) - o membro só pode ser usado em classes do mesmo *package*;

`private` - o membro só pode ser usado na própria classe.

- Mais informação sobre [controlo de acesso no Java Tutorial](#).

Classes

Novos Contextos de

Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

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

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

- Um método tem acesso aos atributos e métodos da própria classe, mesmo que sejam `private`.

Classes

Novos Contextos de

Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

```
public class X {  
    public void pub1( ) { /* . . . */ }  
    public void pub2( ) { /* . . . */ }  
    private void priv1( ) { /* . . . */ }  
    private void priv2( ) { /* . . . */ }  
    private int i;  
    ...  
}  
  
public class XUser {  
    private X myX = new X();  
    public void teste() {  
        myX.pub1(); // OK!  
        // myX.priv1(); Errado!  
    }  
}
```

- Um método tem acesso aos atributos e métodos da própria classe, mesmo que sejam `private`.

Classes

Novos Contextos de

Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

```
public class X {  
    public void publ ( ) { /* . . . */ }  
    public void pub2 ( ) { /* . . . */ }  
    private void priv1 ( ) { /* . . . */ }  
    private void priv2 ( ) { /* . . . */ }  
    private int i;  
    ...  
}  
  
public class XUser {  
    private X myX = new X();  
    public void teste() {  
        myX.publ(); // OK!  
        // myX.priv1(); Errado!  
    }  
}
```

- Um método tem acesso aos atributos e métodos da própria classe, mesmo que sejam `private`.

Classes

Novos Contextos de

Existência

Objects

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

- Uma classe pode dispor de diversos métodos privados que só podem ser utilizados internamente por outros métodos da classe;

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

Classes

Novos Contextos de

Existência

Objects

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

- Uma classe pode dispor de diversos métodos privados que só podem ser utilizados internamente por outros métodos da classe;

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


Sobreposição (*Overloading*)

Classes

Novos Contextos de
Existência

Objects

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

- Muitas linguagens requerem que funções diferentes tenham nomes diferentes – mesmo que executem essencialmente a mesma acção:

```
void sortArray(Array a);  
void sortLista(Lista l);  
void sortSet(Set s);
```

- Em Java, é possível ter várias funções com o mesmo nome:

```
void sort(Array a);  
void sort(Lista l);  
void sort(Set s);
```

- A distinção faz-se pela *assinatura* completa da função (assinatura = nome + parâmetros);
- Não é possível distinguir funções pelo tipo de valor devolvido (porque poderia gerar situações ambíguas).

Classes

Novos Contextos de
Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

- Muitas linguagens requerem que funções diferentes tenham nomes diferentes – mesmo que executem essencialmente a mesma acção:

```
void sortArray(Array a);  
void sortLista(Lista l);  
void sortSet(Set s);
```

- Em Java, é possível ter várias funções com o mesmo nome:

```
void sort(Array a);  
void sort(Lista l);  
void sort(Set s);
```

- A distinção faz-se pela *assinatura* completa da função (assinatura = nome + parâmetros);
- Não é possível distinguir funções pelo tipo de valor devolvido (porque poderia gerar situações ambíguas).

Classes

Novos Contextos de
Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

- Muitas linguagens requerem que funções diferentes tenham nomes diferentes – mesmo que executem essencialmente a mesma acção:

```
void sortArray(Array a);  
void sortLista(Lista l);  
void sortSet(Set s);
```

- Em Java, é possível ter várias funções com o mesmo nome:

```
void sort(Array a);  
void sort(Lista l);  
void sort(Set s);
```

- A distinção faz-se pela *assinatura* completa da função (assinatura = nome + parâmetros);
- Não é possível distinguir funções pelo tipo de valor devolvido (porque poderia gerar situações ambíguas).

Sobreposição (*Overloading*)

Classes

Novos Contextos de
Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

- Muitas linguagens requerem que funções diferentes tenham nomes diferentes – mesmo que executem essencialmente a mesma acção:

```
void sortArray(Array a);  
void sortLista(Lista l);  
void sortSet(Set s);
```

- Em Java, é possível ter várias funções com o mesmo nome:

```
void sort(Array a);  
void sort(Lista l);  
void sort(Set s);
```

- A distinção faz-se pela *assinatura* completa da função (assinatura = nome + parâmetros);
- Não é possível distinguir funções pelo tipo de valor devolvido (porque poderia gerar situações ambíguas).

Classes

Novos Contextos de
Existência
Objectos
Encapsulamento

Sobreposição (*Overloading*)

Construtores
Resumo

Pacotes (*Packages*)

- Muitas linguagens requerem que funções diferentes tenham nomes diferentes – mesmo que executem essencialmente a mesma acção:

```
void sortArray(Array a);  
void sortLista(Lista l);  
void sortSet(Set s);
```

- Em Java, é possível ter várias funções com o mesmo nome:

```
void sort(Array a);  
void sort(Lista l);  
void sort(Set s);
```

- A distinção faz-se pela *assinatura* completa da função (assinatura = nome + parâmetros);
- Não é possível distinguir funções pelo tipo de valor devolvido (porque poderia gerar situações ambíguas).

Classes

Novos Contextos de
Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

- A inicialização de um objecto pode implicar a inicialização simultânea de diversos atributos.
- Um **construtor** é um método especial que é invocado sempre que um novo objecto é criado.
- Os objectos são criados por instanciação através do operador `new`:

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

- O construtor distingue-se por ter o nome igual ao da classe e por não ter resultado (nem sequer `void`).
- Pode haver vários construtores sobrepostos (com assinaturas distintas) de modo a permitir diferentes formas de inicialização:

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

Classes

Novos Contextos de
Existência
Objectos
Encapsulamento
Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

- A inicialização de um objecto pode implicar a inicialização simultânea de diversos atributos.
- Um **construtor** é um método especial que é invocado sempre que um novo objecto é criado.
- Os objectos são criados por instanciação através do operador `new`:

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

- O construtor distingue-se por ter o nome igual ao da classe e por não ter resultado (nem sequer `void`).
- Pode haver vários construtores sobrepostos (com assinaturas distintas) de modo a permitir diferentes formas de inicialização:

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

Classes

Novos Contextos de
Existência
Objectos
Encapsulamento
Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

- A inicialização de um objecto pode implicar a inicialização simultânea de diversos atributos.
- Um **construtor** é um método especial que é invocado sempre que um novo objecto é criado.
- Os objectos são criados por instanciação através do operador `new`:

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

- O construtor distingue-se por ter o nome igual ao da classe e por não ter resultado (nem sequer `void`).
- Pode haver vários construtores sobrepostos (com assinaturas distintas) de modo a permitir diferentes formas de inicialização:

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


Classes

Novos Contextos de
Existência
Objectos
Encapsulamento
Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

- A inicialização de um objecto pode implicar a inicialização simultânea de diversos atributos.
- Um **construtor** é um método especial que é invocado sempre que um novo objecto é criado.
- Os objectos são criados por instanciação através do operador `new`:

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

- O construtor distingue-se por ter o nome igual ao da classe e por não ter resultado (nem sequer `void`).
- Pode haver vários construtores sobrepostos (com assinaturas distintas) de modo a permitir diferentes formas de inicialização:

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

Classes

Novos Contextos de
Existência
Objectos
Encapsulamento
Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

- A inicialização de um objecto pode implicar a inicialização simultânea de diversos atributos.
- Um **construtor** é um método especial que é invocado sempre que um novo objecto é criado.
- Os objectos são criados por instanciação através do operador `new`:

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

- O construtor distingue-se por ter o nome igual ao da classe e por não ter resultado (nem sequer `void`).
- Pode haver vários construtores sobrepostos (com assinaturas distintas) de modo a permitir diferentes formas de inicialização:

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

Classes

Novos Contextos de
Existência
Objectos
Encapsulamento
Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

- A inicialização de um objecto pode implicar a inicialização simultânea de diversos atributos.
- Um **construtor** é um método especial que é invocado sempre que um novo objecto é criado.
- Os objectos são criados por instanciação através do operador `new`:

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

- O construtor distingue-se por ter o nome igual ao da classe e por não ter resultado (nem sequer `void`).
- Pode haver vários construtores sobrepostos (com assinaturas distintas) de modo a permitir diferentes formas de inicialização:

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

Construtores (2)

- O construtor é executado apenas no momento da criação do objecto.
- É usado para inicializar os atributos do novo objecto, de forma a deixá-lo num estado coerente.
- Pode ter parâmetros.
- Não devolve qualquer resultado.
- Tem sempre o nome da classe.

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

Classes

Novos Contextos de
Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

- O construtor é executado apenas no momento da criação do objecto.
- É usado para inicializar os atributos do novo objecto, de forma a deixá-lo num estado coerente.
- Pode ter parâmetros.
- Não devolve qualquer resultado.
- Tem sempre o nome da classe.

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

Classes

Novos Contextos de

Existência

Objects

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

Construtores (2)

- O construtor é executado apenas no momento da criação do objecto.
- É usado para inicializar os atributos do novo objecto, de forma a deixá-lo num estado coerente.
- Pode ter parâmetros.
- Não devolve qualquer resultado.
- Tem sempre o nome da classe.

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

Classes

Novos Contextos de

Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

- O construtor é executado apenas no momento da criação do objecto.
- É usado para inicializar os atributos do novo objecto, de forma a deixá-lo num estado coerente.
- Pode ter parâmetros.
- Não devolve qualquer resultado.
- Tem sempre o nome da classe.

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

Classes

Novos Contextos de

Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

- O construtor é executado apenas no momento da criação do objecto.
- É usado para inicializar os atributos do novo objecto, de forma a deixá-lo num estado coerente.
- Pode ter parâmetros.
- Não devolve qualquer resultado.
- Tem sempre o nome da classe.

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

Classes

Novos Contextos de

Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

- O construtor é executado apenas no momento da criação do objecto.
- É usado para inicializar os atributos do novo objecto, de forma a deixá-lo num estado coerente.
- Pode ter parâmetros.
- Não devolve qualquer resultado.
- Tem sempre o nome da classe.

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

Classes

Novos Contextos de

Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

- O construtor é executado apenas no momento da criação do objecto.
- É usado para inicializar os atributos do novo objecto, de forma a deixá-lo num estado coerente.
- Pode ter parâmetros.
- Não devolve qualquer resultado.
- Tem sempre o nome da classe.

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

Classes

Novos Contextos de

Existência

Objectos

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

- O construtor é executado apenas no momento da criação do objecto.
- É usado para inicializar os atributos do novo objecto, de forma a deixá-lo num estado coerente.
- Pode ter parâmetros.
- Não devolve qualquer resultado.
- Tem sempre o nome da classe.

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

Classes

Novos Contextos de

Existência

Objects

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

Construtor “por omissão”

- Se a classe não definir nenhum construtor, o compilador cria automaticamente um *construtor por omissão* (default constructor).
- O construtor por omissão não tem parâmetros.

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

- No entanto, se a classe definir um construtor ou mais, o compilador já não cria o de omissão (nem este pode ser utilizado):

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

- Mesmo antes de executar o construtor, a linguagem Java inicializa todos os atributos com valores nulos ou com os valores dados nas suas declarações.

Classes

Novos Contextos de
Existência

Objects

Encapsulamento

Sobreposição (Overloading)

Construtores

Resumo

Pacotes (Packages)

Construtor “por omissão”

- Se a classe não definir nenhum construtor, o compilador cria automaticamente um *construtor por omissão* (default constructor).
- O construtor por omissão não tem parâmetros.

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

- No entanto, se a classe definir um construtor ou mais, o compilador já não cria o de omissão (nem este pode ser utilizado):

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

- Mesmo antes de executar o construtor, a linguagem Java inicializa todos os atributos com valores nulos ou com os valores dados nas suas declarações.

Construtor “por omissão”

- Se a classe não definir nenhum construtor, o compilador cria automaticamente um *construtor por omissão* (default constructor).
- O construtor por omissão não tem parâmetros.

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

- No entanto, se a classe definir um construtor ou mais, o compilador já não cria o de omissão (nem este pode ser utilizado):

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

- Mesmo antes de executar o construtor, a linguagem Java inicializa todos os atributos com valores nulos ou com os valores dados nas suas declarações.

Classes

Novos Contextos de

Existência

Objects

Encapsulamento

Sobreposição (Overloading)

Construtores

Resumo

Pacotes (Packages)

Construtor “por omissão”

- Se a classe não definir nenhum construtor, o compilador cria automaticamente um *construtor por omissão* (default constructor).
- O construtor por omissão não tem parâmetros.

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

- No entanto, se a classe definir um construtor ou mais, o compilador já não cria o de omissão (nem este pode ser utilizado):

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

- Mesmo antes de executar o construtor, a linguagem Java inicializa todos os atributos com valores nulos ou com os valores dados nas suas declarações.

Classes

Novos Contextos de

Existência

Objects

Encapsulamento

Sobreposição (Overloading)

Construtores

Resumo

Pacotes (Packages)

Construtor “por omissão”

- Se a classe não definir nenhum construtor, o compilador cria automaticamente um *construtor por omissão* (*default constructor*).
- O construtor por omissão não tem parâmetros.

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

- No entanto, se a classe definir um construtor ou mais, o compilador já não cria o de omissão (nem este pode ser utilizado):

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

- Mesmo antes de executar o construtor, a linguagem Java inicializa todos os atributos com valores nulos ou com os valores dados nas suas declarações.

Classes

Novos Contextos de

Existência

Objects

Encapsulamento

Sobreposição (*Overloading*)

Construtores

Resumo

Pacotes (*Packages*)

O que uma classe pode conter

Classes

Novos Contextos de
Existência
Objects
Encapsulamento
Sobreposição (*Overloading*)
Construtores

Resumo

Pacotes (*Packages*)

- A definição de uma classe pode incluir:
 - zero ou mais declarações de *atributos*;
 - zero ou mais definições de *métodos*;
 - zero ou mais *constantes*;
 - zero ou mais *classes internas* (*inner*);
 - zero ou mais declarações de *classes internas* (*name*).
- Esses elementos só podem ocorrer dentro do bloco:

```
class NomeDaClasse { ... }
```

O que uma classe pode conter

Classes

Novos Contextos de
Existência
Objects
Encapsulamento
Sobreposição (*Overloading*)
Construtores

Resumo

Pacotes (*Packages*)

- A definição de uma classe pode incluir:
 - zero ou mais declarações de **atributos**;
 - zero ou mais definições de **métodos**;
 - zero ou mais **construtores**;
 - zero ou mais **blocos static** (raro);
 - zero ou mais declarações de **classes internas** (raro).
- Esses elementos só podem ocorrer dentro do bloco:

```
class NomeDaClasse { ... }
```

O que uma classe pode conter

Classes

Novos Contextos de
Existência
Objects
Encapsulamento
Sobreposição (*Overloading*)
Construtores

Resumo

Pacotes (*Packages*)

- A definição de uma classe pode incluir:
 - zero ou mais declarações de **atributos**;
 - zero ou mais definições de **métodos**;
 - zero ou mais **construtores**;
 - zero ou mais **blocos static** (raro);
 - zero ou mais declarações de **classes internas** (raro).
- Esses elementos só podem ocorrer dentro do bloco:

```
class NomeDaClasse { ... }
```

O que uma classe pode conter

Classes

Novos Contextos de
Existência
Objects
Encapsulamento
Sobreposição (*Overloading*)
Construtores

Resumo

Pacotes (*Packages*)

- A definição de uma classe pode incluir:
 - zero ou mais declarações de **atributos**;
 - zero ou mais definições de **métodos**;
 - zero ou mais **construtores**;
 - zero ou mais **blocos static** (raro);
 - zero ou mais declarações de **classes internas** (raro).
- Esses elementos só podem ocorrer dentro do bloco:

```
class NomeDaClasse { ... }
```

O que uma classe pode conter

Classes

Novos Contextos de
Existência
Objectos
Encapsulamento
Sobreposição (*Overloading*)
Construtores

Resumo

Pacotes (*Packages*)

- A definição de uma classe pode incluir:
 - zero ou mais declarações de **atributos**;
 - zero ou mais definições de **métodos**;
 - zero ou mais **construtores**;
 - zero ou mais **blocos static** (raro);
 - zero ou mais declarações de **classes internas** (raro).
- Esses elementos só podem ocorrer dentro do bloco:

```
class NomeDaClasse { ... }
```

O que uma classe pode conter

Classes

Novos Contextos de
Existência
Objectos
Encapsulamento
Sobreposição (*Overloading*)
Construtores

Resumo

Pacotes (*Packages*)

- A definição de uma classe pode incluir:
 - zero ou mais declarações de **atributos**;
 - zero ou mais definições de **métodos**;
 - zero ou mais **construtores**;
 - zero ou mais **blocos static** (raro);
 - zero ou mais declarações de **classes internas** (raro).
- Esses elementos só podem ocorrer dentro do bloco:

```
class NomeDaClasse { ... }
```

O que uma classe pode conter

Classes

Novos Contextos de
Existência
Objectos
Encapsulamento
Sobreposição (*Overloading*)
Construtores

Resumo

Pacotes (*Packages*)

- A definição de uma classe pode incluir:
 - zero ou mais declarações de **atributos**;
 - zero ou mais definições de **métodos**;
 - zero ou mais **construtores**;
 - zero ou mais **blocos static** (raro);
 - zero ou mais declarações de **classes internas** (raro).
- Esses elementos só podem ocorrer dentro do bloco:

```
class NomeDaClasse { ... }
```

O que uma classe pode conter

Classes

Novos Contextos de
Existência
Objectos
Encapsulamento
Sobreposição (*Overloading*)
Construtores

Resumo

Pacotes (*Packages*)

- A definição de uma classe pode incluir:
 - zero ou mais declarações de **atributos**;
 - zero ou mais definições de **métodos**;
 - zero ou mais **construtores**;
 - zero ou mais **blocos static** (raro);
 - zero ou mais declarações de **classes internas** (raro).
- Esses elementos só podem ocorrer dentro do bloco:

```
class NomeDaClasse { ... }
```


Classes

- Novos Contextos de Existência
- Objects
- Encapsulamento
- Sobreposição (*Overloading*)
- Construtores

Resumo

Pacotes (*Packages*)

```
public class Point {  
  
    public Point() {...}  
    public Point(double x, double y) {...}  
  
    public void set(double newX, double newY) {...}  
    public void move(double deltaX, double deltaY) {...}  
  
    public double getX() {...}  
    public double getY() {...}  
    public double distanceTo(Point p) {...}  
    public void display() {...}  
  
    private double x;  
    private double y;  
  
}
```

Classes

- Novos Contextos de Existência
- Objectos
- Encapsulamento
- Sobreposição (*Overloading*)
- Construtores

Resumo

Pacotes (*Packages*)

```
public class Point {  
  
    public Point() {...}  
    public Point(double x, double y) {...}  
  
    public void set(double newX, double newY) {...}  
    public void move(double deltaX, double deltaY) {...}  
  
    public double getX() {...}  
    public double getY() {...}  
    public double distanceTo(Point p) {...}  
    public void display() {...}  
  
    private double x;  
    private double y;  
  
}
```

Classes

Novos Contextos de
Existência
Objects
Encapsulamento
Sobreposição (*Overloading*)
Construtores
Resumo

Pacotes (*Packages*)

- Em Java o espaço de nomes é gerido através do conceito de *package*;
- Porque é preciso gerir o espaço de nomes?
- Para evitar conflitos de nomes de classes!
 - Não temos geralmente problemas por distinguir os nomes das classes que implementamos.
 - Mas como garantimos que a nossa classe `Point` não colide com outra que eventualmente possa já existir?
- É um problema análogo ao dos nomes de ficheiros num disco.

Classes

Novos Contextos de
Existência
Objects
Encapsulamento
Sobreposição (*Overloading*)
Construtores
Resumo

Pacotes (*Packages*)

- Em Java o espaço de nomes é gerido através do conceito de *package*;
- Porque é preciso gerir o espaço de nomes?
- Para evitar conflitos de nomes de classes!
 - Não temos geralmente problemas em distinguir os nomes das classes que implementamos.
 - Mas como garantimos que a nossa classe *Point* não colide com outra que eventualmente possa já existir?
- É um problema análogo ao dos nomes de ficheiros num disco.

Classes

Novos Contextos de
Existência
Objects
Encapsulamento
Sobreposição (*Overloading*)
Construtores
Resumo

Pacotes (*Packages*)

- Em `Java` o espaço de nomes é gerido através do conceito de *package*;
- Porque é preciso gerir o espaço de nomes?
- Para evitar conflitos de nomes de classes!
 - Não temos geralmente problemas em distinguir os nomes das classes que implementamos.
 - Mas como garantimos que a nossa classe `Point` não colide com outra que eventualmente possa já existir?
- É um problema análogo ao dos nomes de ficheiros num disco.

Classes

Novos Contextos de
Existência
Objects
Encapsulamento
Sobreposição (*Overloading*)
Construtores
Resumo

Pacotes (*Packages*)

- Em `Java` o espaço de nomes é gerido através do conceito de *package*;
- Porque é preciso gerir o espaço de nomes?
- Para evitar conflitos de nomes de classes!
 - Não temos geralmente problemas em distinguir os nomes das classes que implementamos.
 - Mas como garantimos que a nossa classe `Point` não colide com outra que eventualmente possa já existir?
- É um problema análogo ao dos nomes de ficheiros num disco.

Classes

Novos Contextos de
Existência
Objects
Encapsulamento
Sobreposição (*Overloading*)
Construtores
Resumo

Pacotes (*Packages*)

- Em `Java` o espaço de nomes é gerido através do conceito de *package*;
- Porque é preciso gerir o espaço de nomes?
- Para evitar conflitos de nomes de classes!
 - Não temos geralmente problemas em distinguir os nomes das classes que implementamos.
 - Mas como garantimos que a nossa classe `Point` não colide com outra que eventualmente possa já existir?
- É um problema análogo ao dos nomes de ficheiros num disco.

Classes

Novos Contextos de
Existência
Objects
Encapsulamento
Sobreposição (*Overloading*)
Construtores
Resumo

Pacotes (*Packages*)

- Em `Java` o espaço de nomes é gerido através do conceito de *package*;
- Porque é preciso gerir o espaço de nomes?
- Para evitar conflitos de nomes de classes!
 - Não temos geralmente problemas em distinguir os nomes das classes que implementamos.
 - Mas como garantimos que a nossa classe `Point` não colide com outra que eventualmente possa já existir?
- É um problema análogo ao dos nomes de ficheiros num disco.

Classes

Novos Contextos de
Existência
Objects
Encapsulamento
Sobreposição (*Overloading*)
Construtores
Resumo

Pacotes (*Packages*)

- Em `Java` o espaço de nomes é gerido através do conceito de *package*;
- Porque é preciso gerir o espaço de nomes?
- Para evitar conflitos de nomes de classes!
 - Não temos geralmente problemas em distinguir os nomes das classes que implementamos.
 - Mas como garantimos que a nossa classe `Point` não colide com outra que eventualmente possa já existir?
- É um problema análogo ao dos nomes de ficheiros num disco.

Instrução *import*

- Utilização:

- As classes e as referências simples das mesmas podem ser usadas ou utilizando a primitiva `import`.

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

- As cláusulas `import` devem aparecer sempre antes das declarações de classes.

- Quando escrevemos:

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

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

```
Scanner in = new Scanner(System.in);
```

- De outra forma teríamos de escrever:

```
java.util.Scanner in = new java.util.Scanner(System.in);
```

Classes

Novos Contextos de
Existência
Objects
Encapsulamento
Sobreposição (*Overloading*)
Construtores
Resumo

Pacotes (*Packages*)

- Utilização:

- As classes são referenciadas através dos seus nomes absolutos ou utilizando a primitiva `import`;

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

- As cláusulas `import` devem aparecer sempre antes das declarações de classes;

- Quando escrevemos:

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

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

```
Scanner in = new Scanner(System.in);
```

- De outra forma teríamos de escrever:

```
java.util.Scanner in = new java.util.Scanner(System.in);
```

Classes

Novos Contextos de
Existência
Objects
Encapsulamento
Sobreposição (*Overloading*)
Construtores
Resumo

Pacotes (*Packages*)

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

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

- As cláusulas `import` devem aparecer sempre antes das declarações de classes;
- Quando escrevemos:

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

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

```
Scanner in = new Scanner(System.in);
```

- De outra forma teríamos de escrever:

```
java.util.Scanner in = new java.util.Scanner(System.in);
```

Classes

Novos Contextos de
Existência
Objectos
Encapsulamento
Sobreposição (*Overloading*)
Construtores
Resumo

Pacotes (*Packages*)

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

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

- As cláusulas `import` devem aparecer sempre antes das declarações de classes;
- Quando escrevemos:

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

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

```
Scanner in = new Scanner(System.in);
```

- De outra forma teríamos de escrever:

```
java.util.Scanner in = new java.util.Scanner(System.in);
```

Classes

Novos Contextos de
Existência
Objects
Encapsulamento
Sobreposição (*Overloading*)
Construtores
Resumo

Pacotes (*Packages*)

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

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

- As cláusulas `import` devem aparecer sempre antes das declarações de classes;
- Quando escrevemos:

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

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

```
Scanner in = new Scanner(System.in);
```

- De outra forma teríamos de escrever:

```
java.util.Scanner in = new java.util.Scanner(System.in);
```

Classes

Novos Contextos de
Existência
Objectos
Encapsulamento
Sobreposição (*Overloading*)
Construtores
Resumo

Pacotes (*Packages*)

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

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

- As cláusulas `import` devem aparecer sempre antes das declarações de classes;
- Quando escrevemos:

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

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

```
Scanner in = new Scanner(System.in);
```

- De outra forma teríamos de escrever:

```
java.util.Scanner in = new java.util.Scanner(System.in);
```

Classes

Novos Contextos de
Existência
Objects
Encapsulamento
Sobreposição (*Overloading*)
Construtores
Resumo

Pacotes (*Packages*)

Criar um novo pacote

Classes

Novos Contextos de
Existência
Objects
Encapsulamento
Sobreposição (*Overloading*)
Construtores
Resumo

Pacotes (*Packages*)

- Podemos organizar as nossas classes em pacotes.
- Para isso, o ficheiro que define a classe (`MyClass.java`, por exemplo) deve declarar na primeira linha de código:

```
package pt.ua.prog;
```

- Isto garante que a classe (`MyClass`) terá parte do pacote `pt.ua.prog`.
- Além disso, o ficheiro tem de corresponder a uma entrada de directório que reflita o nome do pacote:
`pt/ua/prog/MyClass.java`
 - É recomendado usar uma espécie de análogo de `import` para o pacote.

Classes

Novos Contextos de
Existência
Objects
Encapsulamento
Sobreposição (*Overloading*)
Construtores
Resumo

Pacotes (*Packages*)

- Podemos organizar as nossas classes em pacotes.
- Para isso, o ficheiro que define a classe (`MyClass.java`, por exemplo) deve declarar na primeira linha de código:

```
package pt.ua.prog;
```

- Isto garante que a classe (`MyClass`) fará parte do pacote `pt.ua.prog`.
- Além disso, o ficheiro tem de corresponder a uma entrada de directório que reflita o nome do pacote:
`pt/ua/prog/MyClass.java`
 - É recomendado usar uma espécie de endereço de Internet invertido.

Classes

Novos Contextos de
Existência
Objectos
Encapsulamento
Sobreposição (*Overloading*)
Construtores
Resumo

Pacotes (*Packages*)

- Podemos organizar as nossas classes em pacotes.
- Para isso, o ficheiro que define a classe (`MyClass.java`, por exemplo) deve declarar na primeira linha de código:

```
package pt.ua.prog;
```

- Isto garante que a classe (`MyClass`) fará parte do pacote `pt.ua.prog`.
- Além disso, o ficheiro tem de corresponder a uma entrada de directório que reflita o nome do pacote:
`pt/ua/prog/MyClass.java`
 - É recomendado usar uma espécie de endereço de Internet invertido.

Classes

Novos Contextos de
Existência
Objectos
Encapsulamento
Sobreposição (*Overloading*)
Construtores
Resumo

Pacotes (*Packages*)

- Podemos organizar as nossas classes em pacotes.
- Para isso, o ficheiro que define a classe (`MyClass.java`, por exemplo) deve declarar na primeira linha de código:

```
package pt.ua.prog;
```

- Isto garante que a classe (`MyClass`) fará parte do pacote `pt.ua.prog`.
- Além disso, o ficheiro tem de corresponder a uma entrada de directório que reflita o nome do pacote:
`pt/ua/prog/MyClass.java`
 - É recomendado usar uma espécie de endereço de Internet invertido.

Classes

Novos Contextos de
Existência
Objectos
Encapsulamento
Sobreposição (*Overloading*)
Construtores
Resumo

Pacotes (*Packages*)

- Podemos organizar as nossas classes em pacotes.
- Para isso, o ficheiro que define a classe (`MyClass.java`, por exemplo) deve declarar na primeira linha de código:

```
package pt.ua.prog;
```

- Isto garante que a classe (`MyClass`) fará parte do pacote `pt.ua.prog`.
- Além disso, o ficheiro tem de corresponder a uma entrada de directório que reflita o nome do pacote:
`pt/ua/prog/MyClass.java`
 - É recomendado usar uma espécie de endereço de Internet invertido.

Classes

Novos Contextos de
Existência
Objectos
Encapsulamento
Sobreposição (*Overloading*)
Construtores
Resumo

Pacotes (*Packages*)

- Podemos organizar as nossas classes em pacotes.
- Para isso, o ficheiro que define a classe (`MyClass.java`, por exemplo) deve declarar na primeira linha de código:

```
package pt.ua.prog;
```

- Isto garante que a classe (`MyClass`) fará parte do pacote `pt.ua.prog`.
- Além disso, o ficheiro tem de corresponder a uma entrada de directório que reflita o nome do pacote:
`pt/ua/prog/MyClass.java`
 - É recomendado usar uma espécie de endereço de Internet invertido.

Classes

Novos Contextos de
Existência
Objectos
Encapsulamento
Sobreposição (*Overloading*)
Construtores
Resumo

Pacotes (*Packages*)

- A sua utilização será na forma:

```
pt.ua.prog.MyClass.someMethod(...);
```

- Ou, recorrendo a um `import`:

```
import pt.ua.prog.MyClass;  
...  
MyClass.someMethod(...);
```

- Ou, para ter acesso direto a todos os membros estáticos:

```
import static pt.ua.prog.MyClass.*;  
...  
someMethod(...);
```

Classes

Novos Contextos de
Existência
Objectos
Encapsulamento
Sobreposição (*Overloading*)
Construtores
Resumo

Pacotes (*Packages*)

- A sua utilização será na forma:

```
pt.ua.prog.MyClass.someMethod(...);
```

- Ou, recorrendo a um `import`:

```
import pt.ua.prog.MyClass;  
...  
MyClass.someMethod(...);
```

- Ou, para ter acesso direto a todos os membros estáticos:

```
import static pt.ua.prog.MyClass.*;  
...  
someMethod(...);
```

Classes

Novos Contextos de
Existência
Objectos
Encapsulamento
Sobreposição (*Overloading*)
Construtores
Resumo

Pacotes (*Packages*)

- A sua utilização será na forma:

```
pt.ua.prog.MyClass.someMethod(...);
```

- Ou, recorrendo a um `import`:

```
import pt.ua.prog.MyClass;  
...  
MyClass.someMethod(...);
```

- Ou, para ter acesso direto a todos os membros estáticos:

```
import static pt.ua.prog.MyClass.*;  
...  
someMethod(...);
```


Classes

Novos Contextos de
Existência
Objectos
Encapsulamento
Sobreposição (*Overloading*)
Construtores
Resumo

Pacotes (*Packages*)

- A sua utilização será na forma:

```
pt.ua.prog.MyClass.someMethod(...);
```

- Ou, recorrendo a um `import`:

```
import pt.ua.prog.MyClass;  
...  
MyClass.someMethod(...);
```

- Ou, para ter acesso direto a todos os membros estáticos:

```
import static pt.ua.prog.MyClass.*;  
...  
someMethod(...);
```