

Paradigmas de Programação



Aula 05

1. Palavras Reservadas
2. Enumerações (Enum)
3. Métodos
4. Sobrecarga de Métodos
5. Encapsulamento
6. Métodos de Acesso
7. Palavras Reservadas Usadas
8. Links Úteis

Palavras reservadas usadas

abstract	continue	for	new	switch
assert***	default	goto*	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum****	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp**	volatile
const*	float	native	super	while

* not used

** added in 1.2

*** added in 1.4

**** added in 5.0



Enumerações (Enums)

- Uma enumeração é um conjunto de valores fixos (constantes) que ajudam o programador a definir valores fixos que podem ser usados no sistema.
- A sintaxe de enumerações em java é a seguinte:

```
<modifier> enum <enum-name> {  
    <ENUM_1>, <ENUM_2>, ..<ENUM_N>  
}
```



- Um exemplo de enumeração poderão ser as estações do ano:

```
public enum Season {  
    SPRING, SUMMER, FALL, WINTER  
}
```

- Outro exemplo poderão ser os dias da semana:

```
public enum Day {  
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,  
    THURSDAY, FRIDAY, SATURDAY  
}
```

- De seguida vamos ver a demonstração dos dias da semana





```
public class EnumTest {
    Day day;

    public EnumTest(Day _day) {
        day = _day;
    }

    public void tellItLikeItIs() {
        switch (day) {
            case MONDAY: System.out.println("Mondays are bad.");
                        break;

            case FRIDAY: System.out.println("Fridays are better.");
                        break;

            case SATURDAY:
            case SUNDAY: System.out.println("Weekends are best.");
                        break;

            default:    System.out.println("Midweek days are so-so.");
                        break;
        }
    }
}

/*Esta classe continua no slide seguinte*/
```



`/*continuação*/`

```
public static void main(String[] args) {  
    EnumTest firstDay = new EnumTest(Day.MONDAY);  
    firstDay.tellItLikeItIs();  
    EnumTest thirdDay = new EnumTest(Day.WEDNESDAY);  
    thirdDay.tellItLikeItIs();  
    EnumTest fifthDay = new EnumTest(Day.FRIDAY);  
    fifthDay.tellItLikeItIs();  
    EnumTest sixthDay = new EnumTest(Day.SATURDAY);  
    sixthDay.tellItLikeItIs();  
    EnumTest seventhDay = new EnumTest(Day.SUNDAY);  
    seventhDay.tellItLikeItIs();  
  
    }  
}
```

`The output is:`

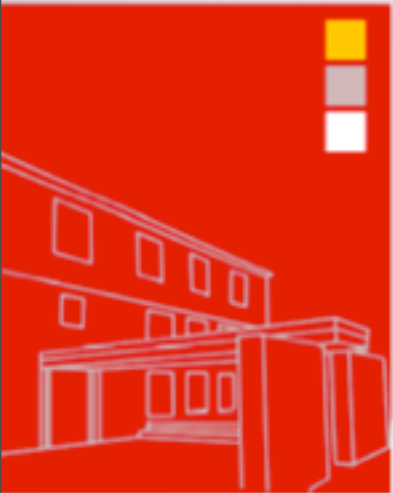
```
Mondays are bad.  
Midweek days are so-so.  
Fridays are better.  
Weekends are best.  
Weekends are best.
```





```
public class Main {  
  
    enum Professor { VITOR, CARLOS, BRUNO }  
  
    public static void main(String[] args) {  
  
        for (Professor professor : Professor.values()) {  
  
            System.out.print(professor.name());  
            System.out.println(" is a professor.");  
        }  
    }  
}
```





Métodos

- Um método numa linguagem orientada a objectos é o equivalente a um procedimento/função numa linguagem não orientada a objectos.

- Um método é a construção de um mecanismo (método) para realizar algum acto.

Definição Sintáctica de um método

```
void <method-name>(<arguments>...) {  
    <statements>...  
}
```

```
<return-type> <method-name>(<arguments>...) {  
    <statements>...  
}
```

- Para a classe `Dog` conseguimos pensar rapidamente num método que faz todo o sentido:

```
public class Dog {  
  
    String name;  
    String barkSound = "woof!";  
    int age = 6;  
  
    Dog(String nametmp, int agetmp) {  
        name = nametmp;  
        age = agetmp;  
    }  
}
```

- Esse método seria o `bark()`!



```
public class Dog {  
  
    String name;  
    String barkSound = "woof!";  
    int age = 6;  
  
    Dog(String nametmp, int agetmp) {  
        name = nametmp;  
        age = agetmp;  
    }  
  
    void bark() {  
        System.out.println(barkSound);  
    }  
}
```

Criámos o método bark ().



- Dada a instância de uma entidade podemos invocar um comportamento com um "." que faz a associação de uma instância com um método.
- Sintaxe da invocação de um método:

```
<instance>.<behavior>( )  
<variable> = <instance>.<behavior>(<arguments>...)
```

- Para invocarmos o método do exemplo anterior temos de fazer o seguinte:

```
Dog fido = new Dog( );  
fido.bark( );
```





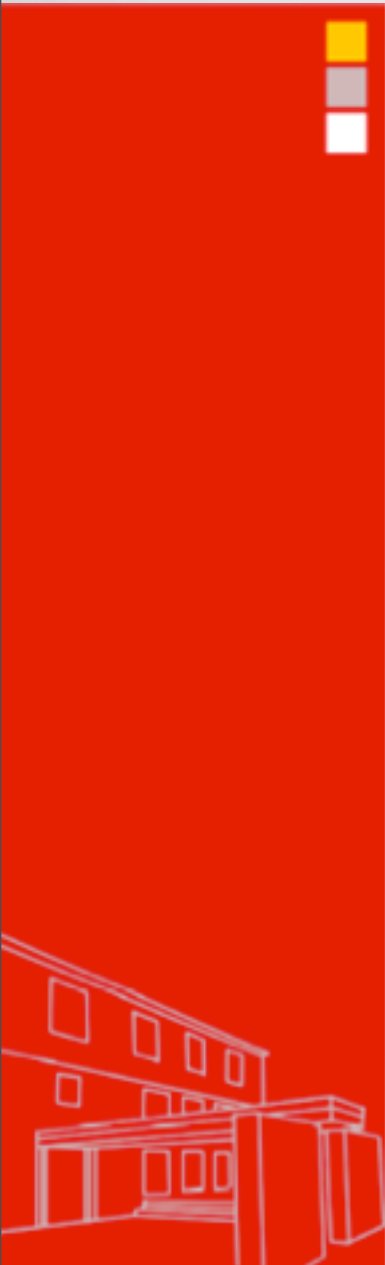
- ■ Em Java podemos definir múltiplos métodos com o mesmo nome desde que tenham assinaturas diferentes.
- ■ As assinaturas de um método são a combinação do nome do método, do tipo de retorno e da lista de argumentos.
- ■ A linguagem Java tem a restrição do tipo de retorno que não contribui para a assinatura do método.
- ■ Em Java não podemos ter dois métodos com o mesmo nome e lista de argumentos mas tipos de retorno diferentes.





Sobrecarga de métodos

- ■ Nem todos os cães soam da mesma forma.
- ■ Para implementarmos uma alteração ao ladrar do cão podemos definir um novo método alternativo ao `bark` que aceite uma `String` como parâmetro.



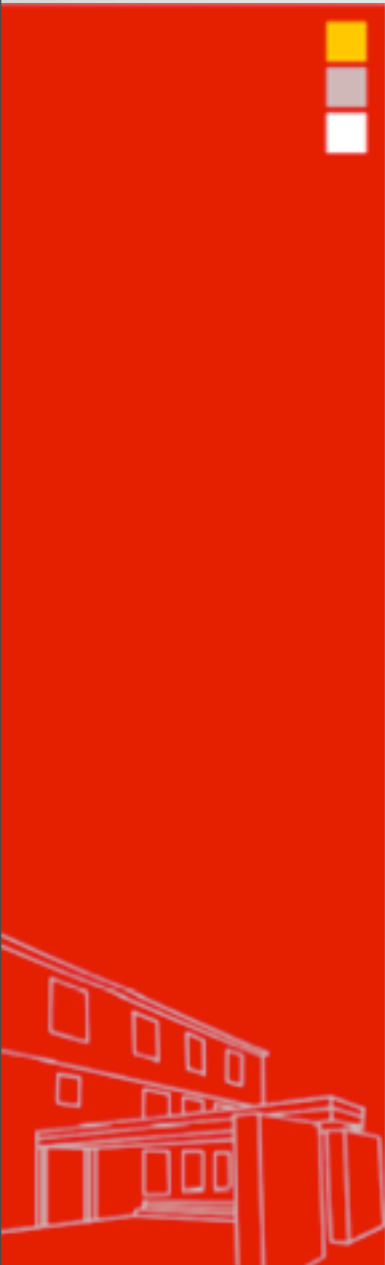
```
public class Dog {  
    (...)  
  
    public void bark() {  
        System.out.println("Woof");  
    }  
  
    public void bark(String barkSound) {  
        System.out.println(barkSound);  
    }  
}
```

- Esta versão de `Dog` é permitida porque apesar de existirem dois métodos `bark()` as diferenças das assinaturas permitem ao interpretador de Java escolher a invocação do método apropriado.



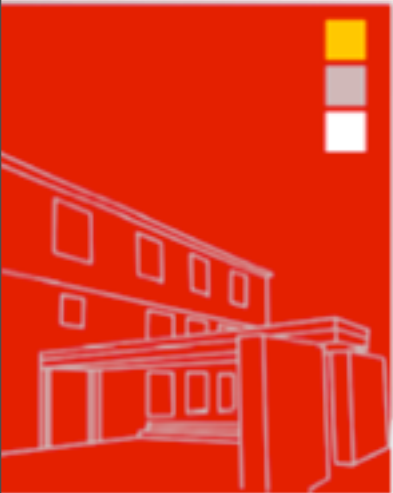
- A sintaxe da definição do método `bark(String barksound)` indica que aceita um argumento do tipo `String`.
- Como exemplo de sobrecarga de métodos considere o programa `DogChorus` onde são criados dois cães cada um com um comportamento diferente para o método `bark`.





```
public class DogChorus {  
    public static void main(String[] args) {  
        Dog fido = new Dog();  
        Dog spot = new Dog();  
        fido.bark();  
        spot.bark("Arf      Arf");  
        fido.bark("Ruff     Ruff");  
    }  
}
```

- Como **Dog** suporta dois tipos de comportamento diferentes para o método **bark** foi invocado no método anterior um ladrar diferente para os cães **fido** e **spot**.
- Note-se que as alterações do ladrar do **fido** surgiram depois do ladrar do **spot**.



Encapsulamento

- Um objecto deve ser visto como uma cápsula de forma a conseguirmos obter uma certa independência do contexto.
- Podemos com isto ter uma maior facilidade na reutilização, detecção de erros e modularidade.

Objecto





- Dentro de um destes módulos (cápsula), os dados, os procedimentos ou ambos podem ser privados ou públicos.
- Os dados e os procedimentos privados são conhecidos e acessíveis apenas pelo próprio objecto e não por qualquer programa externo ao objecto.





- Quando os dados ou os procedimentos de um módulo são públicos, podem ser acedidos por um qualquer programa.
- Tipicamente os procedimentos públicos de um módulo são usados para fornecer um interface controlado para os elementos privados do módulo.





```
public class Dog {  
  
    private String name;  
    private String barkSound = "woof!";  
    private int age = 6;  
  
    public Dog(String nametmp, int agetmp) {  
        name = nametmp;  
        age = agetmp;  
    }  
  
    public void bark() {  
        System.out.println(barkSound);  
    }  
  
    public void bark(String barkSound) {  
        System.out.println(barkSound);  
    }  
}
```





Métodos de Acesso

- Para podermos alterar o valor de uma variável de instância ao longo do tempo temos que ter um método para alterar o seu valor.
- Esse método é tipicamente referido como método de acesso.



- Tipicamente se uma classe tem variáveis de instância que são suportadas por operações "**set**" (de atribuição) também têm operações "**get**" (obter).
- Por convenção um método que afecte ou altere o valor de uma variável de instância deve começar com a palavra "**set**".

```
public void setBarkSound(String barkSound) {  
    this.barkSound = barkSound;  
}
```

- Este método é interessante porque usa variáveis com o mesmo nome, **barkSound**.






- O `barkSound` definido como um parâmetro é o novo ladrar.
- Temos também mais um `barkSound` que é uma variável de instância de `Dog`.
- Com a linguagem Java podemos nos referir a esta variável de instância com o "`this`".

```
this.barkSound = barkSound;
```

- Para cada método "`set`" devemos ter o correspondente "`get`".





```
public String getBarkSound() {  
    return this.barkSound;  
}
```

- No caso das variáveis de instância booleanas alguns programadores gostam de usar o "is" em vez do "get".

- Na versão anterior de `DogChorus` crie um objecto `fido` e altere as características do ladrar de `Woof` para `Ruff` e depois invoque o método `bark`.

```
public class DogChorus {  
  
    public static void main(String[] args) {  
        Dog fido = new Dog();  
        fido.setBarkSound("Ruff.");  
        fido.bark();  
    }  
}
```



```
public class Dog {  
    private String name;  
    private String barkSound = "woof!";  
    private int age=6;  
  
    public Dog(String name, int age) {  
        this.name=name;  
        this.age=age;  
    }  
  
    public void setBarkSound(String barkSound) {  
        this.barkSound = barkSound;  
    }  
  
    public String getBarkSound() {  
        return this.barkSound;  
    }  
  
    (...)  
}
```

Fazer o mesmo para as outras
variáveis de instância





Métodos de Instância

- Os métodos criados até agora são denominados de métodos de instância porque são invocados relativamente a uma instância de uma classe.

- ❖ É por esta razão que um método de instância pode referenciar uma variável directamente sem o qualificador `this` desde que não haja conflito com mais nenhuma variável.

```
public void bark() {  
    System.out.println(barkSound);  
}
```



Palavras reservadas usadas

abstract	continue	for	new	switch
assert***	default	goto*	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum****	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp**	volatile
const*	float	native	super	while

* not used

** added in 1.2

*** added in 1.4

**** added in 5.0



Links Úteis

- ■ <http://docs.oracle.com/javase/tutorial/java/javaOO/enum.html>
- ■ <http://docs.oracle.com/javase/tutorial/java/concepts/object.html>
- ■ <http://docs.oracle.com/javase/tutorial/java/javaOO/methods.html>