02

Crie métodos sobrecarregados

Um método pode ter o mesmo nome que outro, desde que a chamada não fique ambígua: os **argumentos** que são recebidos têm de ser obrigatoriamente diferentes, seja em quantidade ou em tipos.

```
class Teste {
    public void metodo(int i) {
    }

    protected void metodo(double x) {
    }
}
```

COPIAR CÓDIGO

Já o código a seguir não compila:

```
class Teste {
    public int metodo() {}
    protected double metodo() {}
}
```

COPIAR CÓDIGO

Nesse exemplo, temos ambiguidade porque o tipo de retorno não é suficiente para distinguir os métodos durante a chamada.

O Java decide qual das assinaturas de método sobrecarregado (*overloaded*) será utilizada em **tempo de compilação**.

Métodos sobrecarregados podem ter ou não um retorno diferente e uma visibilidade diferente. Mas eles não podem ter exatamente os mesmos tipos e quantidade de parâmetros. Nesse caso, seria uma sobrescrita de método.

No caso de sobrecarga com tipos que possuem polimorfismo, como em `Object` ou `String`, o compilador sempre invoca o método com o tipo mais específico (menos genérico):

```
public class Teste {
    void metodo(Object o) {
        System.out.println("object");
    }
    void metodo(String s) {
        System.out.println("string");
    }

    public static void main(String[] args) {
        new Teste().metodo("string"); // imprime string
    }
}
```

COPIAR CÓDIGO

Se quisermos forçar a invocação ao método mais genérico, devemos fazer o casting forçado:

```
public class Teste {
    void metodo(Object o) {
```

```
        System.out.println("object");
    }
    void metodo(String s) {
        System.out.println("string");
    }

    public static void main(String[] args) {
        new Teste().metodo((Object)"string"); // imprime
object
    }
}
```

COPIAR CÓDIGO

Um exemplo clássico é a troca de ordem, que é vista como sobrecarga, afinal são dois métodos totalmente distintos:

```
void metodo(String i, double x) {
}
void metodo(double x, String i) {
}
```

COPIAR CÓDIGO

Porém, apesar de compiláveis, às vezes o compilador não sabe qual método deverá chamar. No caso a seguir, os números 2 e 3 podem ser considerados tanto `int` quanto `double`, portanto, o compilador fica perdido em qual dos dois métodos invocar, e decide não compilar:

```
public class Teste {
    void metodo(int i, double x) {
    }
    void metodo(double x, int i) {
```

```
    }

    public static void main(String[] args) {
        new Teste().metodo(2, 3);
    }
}
```

COPIAR CÓDIGO

Isso também ocorre com referências, que é diferente do caso com tipo mais específico. Aqui não há tipo mais específico, pois onde um é mais específico, o outro é mais genérico:

```
public class Xpto {
    void metodo(Object o, String s) {
        System.out.println("object");
    }
    void metodo(String s, Object o) {
        System.out.println("string");
    }

    public static void main(String[] args) {
        new Xpto().metodo("string", "string");
    }
}
```

COPIAR CÓDIGO

Diferente do caso em que o segundo método é mais específico:

```
class Xpto2 {
    void metodo(Object o, Object o2) {
        System.out.println("object");
```

```
    }
    void metodo(String s, String s2) {
        System.out.println("string");
    }

    public static void main(String[] args) {
        new Xpto2().metodo("string", "string"); // imprime
string
    }
}
```

COPIAR CÓDIGO