02

Crie métodos com argumentos e valores de retorno

Classes, *enums* e interfaces podem ter métodos definidos em seus corpos.

Todo método tem uma *assinatura* (também chamada de *interface*) e um *corpo* (somente no caso de métodos não abstratos).

A assinatura do método sempre tem:

- um nome seguindo as regras de identificadores;
- um tipo de retorno;
- um conjunto de parâmetros (pode ser vazio), cada um com seu nome e seu tipo;
- um modificador de visibilidade (nem que seja implícito, *package-private*).

E, ainda na assinatura, podemos ter:

- `final` - em caso de herança, o método não pode ser sobrescrito nas classes filhas;
- `abstract` - obriga as classes filhas a implementarem o método. O método abstrato **não** pode ter corpo definido;
- `static` - atributos acessados direto na classe, sem instâncias;
- `synchronized` - *lock* da instância;
- `native` - não cai nesta prova. Permite a implementação do método em código nativo (*JNI*);
- `strictfp` - não cai nesta prova. Ativa o modo de portabilidade matemática para contas de ponto flutuante.

- `throws <EXCEPTIONS>` - após a lista de parâmetros, podemos indicar quantas exceptions quisermos para o `throws` .

A ordem dos elementos na assinatura dos métodos é sempre a seguinte, sendo que os modificadores podem aparecer em qualquer ordem: `<MODIFICADORES> <TIPO_RETORNO> <NOME> (<PARÂMETROS>) <THROWS_EXCEPTIONS>`

Parâmetros

Em Java, usamos parâmetros em métodos e construtores. Definimos uma lista de parâmetros sempre declarando seus tipos e nomes e separando por vírgula:

```
class Param {
    void teste(int a, int b) {

    }
}

// chamada
p.teste(1, 2);
```

COPIAR CÓDIGO

A declaração das variáveis é feita na declaração dos métodos. A inicialização dos valores é feita por quem chama o método. (Note que, em Java, não é possível ter valores `default` para parâmetros e todos são obrigatórios, não podemos deixar de passar nenhum).

O único modificador possível de ser marcado em parâmetros é `final` , para indicar que aquele parâmetro não pode ter seu valor modificado depois da chamada do método (considerado boa prática):

```
class Param {
    void teste (final int a) {
        a = 10; // não compila
    }
}
```

COPIAR CÓDIGO

Promoção em parâmetros

Temos que saber que nossos parâmetros também estão sujeitos à promoção de primitivos e ao polimorfismo. Por exemplo, a classe a seguir ilustra as duas situações:

```
class Param {
    void primitivo (double d) {

    }

    void referencia (Object o) {

    }
}
```

COPIAR CÓDIGO

O primeiro método espera um `double`. Mas se chamarmos passando um `int`, um `float` ou qualquer outro tipo compatível, este será promovido a `double` e a chamada funciona:

```
Param p = new Param();
p.primitivo(10);
p.primitivo(10L);
p.primitivo(10F);
p.primitivo((short) 10);
p.primitivo((byte) 10);
p.primitivo('Z');
```

COPIAR CÓDIGO

A mesma coisa ocorre com o método que recebe `Object` : podemos passar qualquer um que **é um** `Object` , ou seja, qualquer objeto:

```
Param p = new Param();
p.referencia(new Carro());
p.referencia(new Moto());
```

COPIAR CÓDIGO

Retornando valores

Todo método pode retornar um valor ou ser definido como `void` , quando não devolve nada:

```
class A {
    int numero() {
        return 5;
    }
    void nada() {
        return;
```

```
    }
}
```

COPIAR CÓDIGO

No caso de métodos de tipo de retorno `void` (nada), podemos omitir a última instrução:

```
class A {
    void nada() {
        // return; // pois esta linha é opcional
    }
}
```

COPIAR CÓDIGO

Um método desse tipo também pode ter um retorno antecipado:

```
class A {
    void nada(int i) {
        if(i >= 0) return;
        System.out.println("negativo");
    }
}
```

COPIAR CÓDIGO

Não podemos ter nenhum código que seria executado após um retorno:

```
class A {
    void nada(int i) {
        if(i >= 0) {
```

```
            return;

            // não compila, pois nunca chegará aqui
            System.out.println("era positivo ou zero");
        }
        System.out.println("negativo");
    }
}
```

COPIAR CÓDIGO

Todo método que possui um tipo de retorno definido (isto é, diferente de `void`), deve retornar algo ou jogar uma `Exception` em cada um dos caminhos de saída possíveis do método, caso contrário o código não compila:

```
String metodo(int a) {
    if(a > 0) {
        return "positivo";
    } else if(a < 0) {
        return "negativo";
    }
    //não compila, o que acontece se não for nem if nem else
if?
}
```

COPIAR CÓDIGO

Lembre-se que isso é feito pelo compilador, então ele não sabe os valores da variável `a` e se todos os casos foram cobertos:

```
String metodo(int a) {
    if(a > 0) {
        return "positivo";
```

```
    } else if(a <= 0) {
        return "negativo ou zero";
    }
    //não compila, o que acontece se não for nem if nem else
if?
    //o compilador não consegue analisar os dois casos
}
```

COPIAR CÓDIGO

Podemos jogar uma `exception` ou colocar um `return` :

```
String metodo(int a) {
    if(a > 0) {
        return "positivo";
    } else if(a < 0) {
        return "negativo";
    }
    return "zero";
}

String metodo2(int a) {
    if(a > 0) {
        return "positivo";
    } else if(a < 0) {
        return "negativo";
    }
    throw new RuntimeException("não quero zero!");
}
```

COPIAR CÓDIGO

Métodos que não retornam nada não podem ter seu resultado atribuído a uma variável:

```
void metodo() {
    System.out.println("oi");
}
void metodo2() {
    // não compila, o método acima não retorna nada
    int i = metodo();
}
```

COPIAR CÓDIGO

Pelo outro lado, mesmo que um método retorne algo, seu retorno pode ser ignorado:

```
int metodo() {
    System.out.println("oi");
    return 5;
}
void metodo2() {
    int i = metodo(); // i = 5
    // chamei novamente e não retornei nada, sem problemas
    metodo();
}
```

COPIAR CÓDIGO