02

Diferencie tipo de uma referência e tipo de um objeto

Sempre que estendemos alguma classe ou implementamos alguma interface, estamos relacionando nossa classe com a classe mãe ou interface usando um relacionamento chamado de **é um**.

Se `Carro extends Veiculo`, dizemos que *Carro* ***é um*** *Veiculo*. Ou se `ArrayList implements List` dizemos que *ArrayList* ***é um*** *List*.

O relacionamento de **é um** é um dos recursos mais poderosos da orientação a objetos. E é chamado formalmente de **polimorfismo**.

Polimorfismo é a capacidade que temos de referenciar um objeto de formas diferentes, segundo seus relacionamentos de *é um*.

Em especial, usamos polimorfismo quando escrevemos:

```
Veiculo v = new Carro();
List l = new ArrayList();
```

COPIAR CÓDIGO

As heranças e implementações vão formando uma árvore que terá sempre como raiz a classe `Object`. Assim, direta ou indiretamente, todo objeto **é um** `Object`.

O polimorfismo pode ser aplicado à passagem de parâmetros (e é aí que está seu grande poder). Imagine as classes:

```
class Veiculo {}
class Carro extends Veiculo {}
class Moto extends Veiculo {}
class Onibus extends Veiculo {}
class Conversivel extends Carro {}
```

COPIAR CÓDIGO

Se temos um método que recebe `Veiculo`, podemos passar qualquer um daqueles objetos:

```
void metodo (Veiculo v) {

}

// .....

metodo(new Carro());
metodo(new Moto());
metodo(new Onibus());
metodo(new Veiculo());
metodo(new Conversivel());
```

COPIAR CÓDIGO

Dessa forma, conseguimos obter um forte reaproveitamento de código.

Repare que, quando usamos polimorfismo, estamos mudando o tipo da referência, mas nunca o tipo do objeto. Em Java, objetos nunca mudam seu tipo, que é aquele onde demos `new`. O que fazemos é chamar (referenciar) o objeto de várias formas diferentes. Chamar de várias formas.... é o polimorfismo.

Podemos referenciar um objeto pelo seu próprio tipo, por uma de suas classes pai, ou por qualquer interface implementada por ele, direta ou indiretamente:

```
interface A {}
interface B {}
class C implements A {}
class D extends C implements B {}
public class Teste {
    public static void main(String[] args) {

        // mesmo tipo, compila
        D d = new D();

        // D extends C, todo D é um C, compila
        C c = new D();
        C c2 = d;

        // D implements B, todo D implementa B, compila
        B b = new D();
        B b2 = d;

        // D implements A indiretamente, compila
        A a = new D();
        A a2 = a;

        D d2 = new C(); // não, C não é D, não compila

        D d3 = new D();
        C c3 = d3; // compila
        D d4 = c3; // não compila, por mais que o ser humano
                   // saiba, em execução, nem todo C é um D.
    }
}
```

COPIAR CÓDIGO

E como funciona o acesso às variáveis membro e aos métodos? Se temos uma referência para a classe mãe, não importa o que o valor seja em tempo de execução, o compilador não conhece o tempo de execução, então ele só compila chamadas aos métodos definidos na classe mãe:

```
class Veiculo {
    public void liga() { }
}
class Carro extends Veiculo{
    public void mudaMarcha() {}
}

// teste
Veiculo v = new Veiculo();
v.liga(); // compila

Carro c = new Carro(); // ok
c.mudaMarcha(); // compila

Veiculo v2 = c;
v2.liga(); // todo veiculo tem método liga, compila
v2.mudaMarcha(); // não compila, nem todo veiculo tem
```

COPIAR CÓDIGO

Mesmo em casos em que "achamos" que todo veículo tem, se o método não foi definido na classe de referência, o codigo não compila:

```
abstract class Veiculo {
    public void liga() { }
```

```
}
class Carro extends Veiculo{
    public void desliga() { }
}
class Moto extends Veiculo{
    public void desliga() { }
}

Carro c = new Carro(); // ok
c.desliga(); // compila

Veiculo v2 = c;
v2.desliga(); // não compila, Veiculo não tem o método desliga
              // definido
```

COPIAR CÓDIGO

O mesmo valerá para variáveis membro:

```
class Veiculo {
    int velocidade;
}
class Carro extends Veiculo{
    int marcha;
}

// teste
Veiculo v = new Veiculo();
v.velocidade = 3; // compila

Carro c = new Carro(); // ok
c.marcha = 1; // compila

Veiculo v2 = c;
```

```
v2.velocidade = 5; // compila
v2.marcha = 7; // não compila
```

COPIAR CÓDIGO

E temos que cuidar de mais um caso específico: o que acontece se estamos trabalhando com pacotes distintos?

Se o método da classe pai que está sendo sobrescrito é `public`, os métodos que sobrescrevem devem ser `public`, então não tem muita graça.

Já se o método da classe pai é `protected`, os filhos são `protected` ou `public`, que também não tem graça, pois o filho - mesmo em outro pacote - já tinha acesso ao método do pai.

Mas o que acontece se o método no pai é `private` e eu tento sobrescrevê-lo? Ou se o método é `default` e tento sobrescrevê-lo em outro pacote? O mesmo valerá tanto para `private` quando para modificador de escopo padrão:

```
package financeiro;
public class ContaFinanceira extends modelo.Conta {
    void fecha() {
        System.out.println("fechando financeiro");
    }
}
```

COPIAR CÓDIGO

```
package modelo;
public class Conta {
    void fecha() {
        System.out.println("fechando conta normal");
```

```
    }
}
```

COPIAR CÓDIGO

Ao invocar o método `fecha` através de uma referência para `Conta` ou `ContaFinanceira`, o resultado será totalmente diferente:

```
ContaFinanceira c = new ContaFinanceira();
c.fecha();
Conta d = c;
d.fecha();
```

COPIAR CÓDIGO

O código não compila, dependendo do pacote onde ele está. Como assim? Acontece que o método não foi sobrescrito, a classe filha nem sabe da existência do método (privado ou default) do pai, portanto o que ela fez foi criar um método totalmente novo.

Nesse caso, ao invocarmos o método durante compilação, o *binding* é feito para o método específico de cada uma delas, uma vez que são métodos totalmente diferentes. Se o código está no pacote de modelo, a chamada ao método `fecha` de `Conta` compila e imprimiria `normal`. Se estivermos no pacote `financeiro`, a chamada ao `ContaFinanceira` compila e imprime `financeiro`.

Lembre-se que os métodos privados terão um efeito equivalente: eles só são vistos internamente à classe onde foram definidos.