02

Desenvolva código que mostra o uso de polimorfismo

Reescrita ou sobrescrita é a maneira como uma subclasse pode *redefinir* o comportamento de um método que foi herdado de uma das suas superclasses (direta ou indiretamente).

```
class Veiculo {
    public void liga() {
        System.out.println("Veiculo está sendo ligado!");
    }
}

class Carro extends Veiculo {
    public void liga() {
        System.out.println("Carro está sendo ligado!");
    }
}
```

COPIAR CÓDIGO

Agora considere:

```
public class Teste{
    public static void main(String [] args){
        Veiculo v = new Carro();
        v.liga();
    }
}
```

COPIAR CÓDIGO

O método chamado aqui será o da classe `Carro`, independente de a referência ser do tipo `Veiculo` (o que importa é o objeto).

Qual método será executado é descoberto em **tempo de execução** (a *assinatura* é decidida em tempo de compilação!), isso é a chamada virtual de método (*virtual method invocation*).

Para reescrever um método, é necessário:

- exatamente o mesmo nome;
- os parâmetros têm que ser iguais em tipo e ordem (nomes podem mudar);
- retorno do método deve ser igual ou mais específico que o da mãe;
- visibilidade deve ser igual ou maior que o da mãe;
- exceptions lançadas devem ser iguais ou menos que na mãe;
- método na mãe não pode ser `final`.

Se essas regras não forem respeitadas, pode haver um erro de compilação, ou o método declarado não será considerado uma reescrita do método.

A regra sobre visibilidade é: um método reescrito só pode ter visibilidade maior ou igual à do método que está sendo reescrito. (Essa não é uma regra mágica! Faz todo o sentido; pense um pouco sobre o que poderia acontecer se essa regra não existisse).

O código a seguir não compila, pois `ligar` é público na classe mãe, então só pode ser reescrito com visibilidade pública:

```
class Veiculo {
    public void liga() {
        System.out.println("Veiculo esta sendo ligado!");
```

```
    }
}

class Carro extends Veiculo {
    protected void liga() {
        System.out.println("Carro esta sendo ligado!");
    }
}
```

COPIAR CÓDIGO

Muito cuidado com interfaces, pois a definição de um método é, por padrão, `public` e o exercício pode apresentar uma pegadinha de compilação:

```
interface A {
    void a();
}
class B implements A {
    void a() {
        // não compila, o método deveria ser público
    }
}
class C implements A {
    public void a() {
        // compila
    }
}
```

COPIAR CÓDIGO

Estranhamente, um método sobrescrito pode ser abstrato, dizendo para o compilador que quem herdar dessa classe terá que sobrescrever o método original:

```
class A {
    void a() {
    }
}
abstract class B extends A {
    abstract void a(); // sobrescrevendo como abstrato
}

class C extends B{
    // não compila, não redefiniu a
}

class D extends B{
    void a() {
        // compila pois redefiniu a
    }
}
```

COPIAR CÓDIGO

Sobre o **retorno covariante**: permite que a classe filha tenha um retorno igual ou mais específico polimorficamente (um subtipo).

Cuidado! O retorno covariante não vale para tipos primitivos. Um exemplo de retorno covariante:

```
class A {
    List<String> metodo () {
        // devolve lista
    }
}
```

```
class B extends A {
    ArrayList<String> metodo() {
        // devolve array list
    }
}
```

COPIAR CÓDIGO

Outra regra importante sobre reescrita é a assinatura em relação ao *lançamento de exceções* (`throws`). Um método reescrito só pode lançar as mesmas exceções *checked* ou menos que o métodos que está sendo reescrito (quanto às *unchecked*, não há regras e sempre podemos lançar quantas quisermos).

```
import java.sql.SQLException;
import java.io.IOException;

class A {
    public void metodo () throws SQLException, IOException {
    }
}

class B extends A {
    public void metodo () throws IOException {
    }
}
```

COPIAR CÓDIGO

Esse código compila, pois o método na classe `B` lança menos exceções que na classe mãe, respeitando a regra. Já o código a seguir não compila:

```
import java.sql.SQLException;
import java.io.IOException;
```

```
class A {
    public void metodo () throws SQLException {
    }
}

class B extends A {
    public void metodo () throws IOException {
    }
}
```

COPIAR CÓDIGO

Apesar de ambos os métodos lançarem apenas uma exceção, não é isso que importa, pois elas são diferentes. Outro caso que não compila:

```
import java.io.IOException;

class A {
    public void metodo () throws IOException {
    }
}

class B extends A {
    public void metodo () throws Exception {
    }
}
```

COPIAR CÓDIGO

`Exception` é muito mais que `IOException` .

Repare que, quando dizemos *menos exceções que na mãe*, isso indica não apenas quantidade, mas também devemos considerar o polimorfismo. Se trocarmos o

exemplo anterior, compilamos:

```
import java.io.IOException;

class A {
    public void metodo () throws Exception {
    }
}

class B extends A {
    public void metodo () throws IOException {
    }
}
```

COPIAR CÓDIGO

Compila, pois `IOException` é mais específico que `Exception` na árvore de herança.

Polimorfismo e chamadas de métodos

Imagine as classes:

```
class Veiculo {
    void liga() {
        System.out.println("ligando o veiculo");
    }
}
class Carro extends Veiculo {
    void liga() {
        System.out.println("ligando o carro");
    }
```

```
    void desliga() {
    }
}
```

COPIAR CÓDIGO

Se tivermos um objeto do tipo `Carro` com uma referência do tipo `Carro`, ou seja, sem usar polimorfismo, podemos fazer:

```
Carro c = new Carro();
c.liga(); // ligando o carro
c.desliga();
```

COPIAR CÓDIGO

Conseguimos chamar os dois métodos. E, como estamos trabalhando com sobrescrita, o método `liga` chamado é o da classe filha `Carro`.

Mas e se usarmos polimorfismo e a referência para `Veiculo`?

```
Veiculo v = new Carro();
v.liga(); // ligando o carro?
v.desliga();
```

COPIAR CÓDIGO

Vamos linha por linha: primeiro, podemos chamar um `Carro` de `Veiculo` porque ele *é um* (compila sem problemas). Podemos também chamar o método `liga` pois ambas as classes o possuem. Mas o método que será invocado será o da classe filha, o sobrescrito.

Já a chamada ao método `desliga` não compilará, porque ele não está definido na classe `Veiculo`. Como a referência é desse tipo, o método (que existe no objeto) não é visível.

A regra é: para saber se um método de um objeto pode ser chamado, olhamos para o tipo da referência em tempo de compilação. Para realmente chamar o método em tempo de execução, devemos olhar para o objeto ao que demos `new`.

Essa regra faz sentido quando pensamos em um método polimórfico como o seguinte:

```
void metodo (Veiculo v) {
    v.liga(); // compila
    v.desliga(); // não compila
}
```

COPIAR CÓDIGO

Se passarmos um objeto `Carro` para o método, teoricamente ambas as chamadas funcionariam, já que a classe possui tanto o `liga` quanto o `desliga`.

Mas imagine uma classe `Moto` que não tem o método `desliga`. Como `Moto` *é um* `Veiculo`, podemos passar como argumento. O que aconteceria se pudéssemos ter chamado o `desliga` na referência `Veiculo`? Alguma coisa estaria errada.

Portanto, a regra geral é que somente podemos acessar os métodos de acordo com o tipo da referência, pois a verificação da existência do método é feita em compilação. Mas qual o método que será invocado, isso será conferido dinamicamente, em execução.

Um ponto muito importante é que o compilador **nunca** sabe o valor das variáveis depois da linha que as cria. Ou seja, o compilador não sabe se estamos passando

um `Carro` ou uma `Moto`. O que ele sabe é apenas o tipo da variável; no caso, `Veiculo`. E como `Veiculo` não tem o método `desliga`, o código não pode compilar.

this, super e sobrescrita de métodos

Na ocasião em que um método foi sobrescrito, podemos utilizar as palavras-chave `super` e `this` para deixar explícito qual método desejamos invocar:

```
class A {
    public void metodo() {
        System.out.println("a");
    }
}
class B extends A {
    public void metodo() {
        System.out.println("b");
        super.metodo(); // imprime a
    }

    public void metodo2() {
        metodo(); // imprime b, a
        super.metodo(); // imprime a
    }
}
```

COPIAR CÓDIGO

E se eu invocar o segundo método na primeira classe? Sem o `this`?

```
class A{
```

```
    public void metodo() {
        System.out.println("a");
        metodo2();
    }
    public void metodo2() {
        System.out.println("metodo 2 do pai");
    }
}
class B extends A {
    public void metodo() {
        System.out.println("b");
        super.metodo();
    }
    public void metodo2() {
        System.out.println("c");
        metodo();
        super.metodo();
    }
    public static void main(String[] args) {
        new B().metodo2();
    }
}
```

COPIAR CÓDIGO

O Java entra em loop infinito, uma vez que o método será invocado no objeto. Então faremos o *lookup* do `metodo2` dinamicamente, encontrando o `metodo2` que chama `metodo`, que chama `metodo` do pai, que chama novamente `metodo2`. Note que o *lookup* dos métodos, o *binding* dos métodos, é feito em execução mesmo se invocarmos dentro de um próprio objeto. Até mesmo o uso da palavra-chave `this` não evitaria isso, causando o loop:

```
class A{
    public void metodo() {
```

```
        System.out.println("a");
        this.metodo2();
    }
    public void metodo2() {
        System.out.println("metodo 2 do pai");
    }
}
class B extends A {
    public void metodo() {
        System.out.println("b");
        super.metodo();
    }
    public void metodo2() {
        System.out.println("c");
        metodo();
        super.metodo();
    }
    public static void main(String[] args) {
        new B().metodo2();
    }
}
```

COPIAR CÓDIGO