OVERRIDING

Overriding ocorre quando uma subclasse define um método com a mesma assinatura (nome, parâmetros e tipo de retorno) de um método já existente em sua superclasse. Isso permite que a subclasse modifique ou estenda o comportamento herdado da superclasse.

Exemplo:

```
 class Animal {
    public void fazerSom() {
        System.out.println("O animal faz um som");
    }
}

class Cachorro extends Animal {
    @Override
    public void fazerSom() {
        System.out.println("O cachorro late: Au Au!");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal animal = new Animal();
        animal.fazerSom(); // Saída: O animal faz um som

        Cachorro cachorro = new Cachorro();
        cachorro.fazerSom(); // Saída: O cachorro late: Au Au!

        Animal animalCachorro = new Cachorro();
        animalCachorro.fazerSom(); // Saída: O cachorro late: Au
Au!
    }
}
```

Neste exemplo, a classe `Cachorro` sobrescreve o método `fazerSom()` da classe `Animal`.

Nota:

- A variável `animalCachorro` é do tipo `Animal`, mas ela é instanciada com um objeto do tipo `Cachorro`.
- Mesmo que a variável tenha o tipo da classe pai (`Animal`), o método `fazerSom()` que será chamado é o da classe `Cachorro`, pois o método foi sobrescrito na subclasse.
- Em Java, o método que será executado é determinado em tempo de execução com base no tipo real do objeto referenciado pela variável, e não pelo tipo da variável. Como `animalCachorro` aponta para um objeto do tipo `Cachorro`, a implementação de `fazerSom()` na classe `Cachorro` será chamada, resultando na saída: "O cachorro late: Au Au!".

Quando usar *overriding*:

1. Especialização de comportamento: Quando você quer que uma subclasse tenha um comportamento mais específico para um método.

2. Polimorfismo: Para permitir que objetos de diferentes subclasses sejam tratados de maneira uniforme através da interface da superclasse.
3. Extensão de funcionalidade: Quando você quer adicionar funcionalidades ao método da superclasse na subclasse.
4. Implementação de métodos abstratos: Em classes abstratas ou interfaces, onde os métodos são declarados sem implementação.
5. Adaptação a contextos específicos: Quando o comportamento geral definido na superclasse não é apropriado para a subclasse.

Considerações importantes:

1. Use a anotação `@Override` para garantir que você está realmente sobrescrevendo um método e não criando um novo, além de melhorar a legibilidade do seu código.
2. Construtores não são herdados e, portanto, não podem ser sobrescritos.

Overriding é uma ferramenta poderosa para criar hierarquias de classes flexíveis e extensíveis, permitindo que você adapte o comportamento de classes herdadas às necessidades específicas de suas subclasses, mantendo uma interface consistente.

REGRAS PARA APLICAR O *OVERRIDING*

1. O método sobrescrito deve ter a mesma assinatura que o método da superclasse

Explicação:
Quando você sobrescreve um método em uma subclasse, o método deve ter exatamente a mesma assinatura que o método na superclasse. A assinatura inclui o nome do método, o número e o tipo dos parâmetros, além da ordem desses parâmetros.

Exemplo:
Suponha que você tenha uma superclasse `Animal` com um método `fazerSom()`:

```
class Animal {
    public void fazerSom() {
        System.out.println("Algum som genérico de animal");
    }
}
```

Se uma subclasse `Cachorro` quiser sobrescrever esse método, ela deve usar a mesma assinatura:

```
class Cachorro extends Animal {
    @Override
    public void fazerSom() {
        System.out.println("Latido");
    }
}
```

Se o método na subclasse tivesse uma assinatura diferente, como adicionar um parâmetro, ele não seria considerado uma sobrescrita, mas sim uma sobrecarga.

2. O método sobrescrito deve ser pelo menos tão acessível quanto o método original

Explicação:
A visibilidade (nível de acesso) do método sobrescrito na subclasse não pode ser mais restritiva do que a do método na superclasse. Se o método da superclasse for `public`, o método da subclasse também deve ser `public` e não pode ser `protected` ou `private`.

Exemplo:
Se o método da superclasse for `public`, o método sobrescrito não pode ser `protected` ou `private`:

```
class Animal {
    public void fazerSom() {
        System.out.println("Algum som genérico de animal");
    }
}

class Cachorro extends Animal {
    // Correto: Mesmo nível de acesso ou menos restritivo
    @Override
    public void fazerSom() {
        System.out.println("Latido");
    }
}
```

Se você tentasse fazer o método sobrescrito `private`, isso resultaria em um erro de compilação.

3. O método sobrescrito não pode declarar uma exceção verificada (*Checked exception*) que seja nova ou mais ampla do que a do método original

Explicação:
Se o método original na superclasse declarar uma exceção verificada (por exemplo, `IOException`), o método sobrescrito na subclasse não pode declarar uma nova exceção verificada ou uma exceção mais ampla (por exemplo, `Exception`). No entanto, ele pode declarar a mesma exceção, uma subclasse dela ou nenhuma exceção.

Exemplo:
Considere a seguinte superclasse:

```
class Animal {
    public void fazerSom() throws IOException {
        System.out.println("Algum som genérico de animal");
    }
}
```

O método sobrescrito na subclasse não deve declarar uma exceção mais ampla como `Exception`:

```
class Cachorro extends Animal {
    // Correto: Declara a mesma exceção ou uma mais específica
    @Override
    public void fazerSom() throws FileNotFoundException { //
Subclasse de IOException
        System.out.println("Latido");
    }
}
```

Se você tentasse declarar uma exceção mais ampla como `Exception`, isso causaria um erro de compilação.

4. O tipo de retorno do método sobrescrito deve ser o mesmo ou um subtipo do tipo de retorno do método original (tipos de retorno covariantes)

Explicação:
O tipo de retorno do método sobrescrito pode ser o mesmo que o tipo de retorno do método na superclasse, ou pode ser uma subclasse (tipo covariante) desse tipo de retorno.

Exemplo:
Se o método da superclasse retorna uma `List`, o método da subclasse pode retornar uma `List` ou qualquer subclasse de `List`, como `ArrayList`:

```
class Animal {
    public List<String> obterAlimento() {
        return new ArrayList<>();
    }
}

class Cachorro extends Animal {
    // Correto: Tipo de retorno covariante
    @Override
    public ArrayList<String> obterAlimento() {
        return new ArrayList<>();
    }
}
```

O exemplo acima é válido porque `ArrayList` é uma subclasse de `List`. Se o método da subclasse tentasse retornar um tipo que não fosse um subtipo, isso causaria um erro de compilação.

5. Se o método é privado, não é visivel para outras classes.

Explicação: Métodos privados em Java são acessíveis apenas dentro da própria classe onde foram definidos. Eles não podem ser acessados ou invocados por outras classes, nem mesmo por subclasses. Isso significa que um método `private` não pode ser sobrescrito por uma subclasse porque ele não é visível para ela. Ao se criar o mesmo método na classe filha o compilador vai encarar como um método completamente independente do método da classe pai.

Exemplo:

```
class ClasseBase {
    private void metodoPrivado() {
        System.out.println("Método privado na ClasseBase");
    }
}

class SubClasse extends ClasseBase {
    // Este método não está sobrescrevendo o método privado da
ClasseBase
    private void metodoPrivado() {
        System.out.println("Método privado na SubClasse");
    }
}
```

Neste exemplo, o método `metodoPrivado()` da `SubClasse` é um método completamente diferente do método `metodoPrivado()` da `ClasseBase`, pois o método na `ClasseBase` é privado e, portanto, não pode ser sobrescrito.

6. Se o método é estático, o método sobreescrito na classe filha também deve ser estático.

Explicação: Métodos estáticos pertencem à classe, não a instâncias de classes. Se um método estático em uma classe base for "sobrescrito" em uma subclasse, o método na subclasse também deve ser declarado como `static`. Isso não é uma verdadeira sobrescrita, mas sim um processo chamado de "ocultação de método" (method hiding).
Exemplo:

```
class ClasseBase {
    static void metodoEstatico() {
        System.out.println("Método estático na ClasseBase");
    }
}

class SubClasse extends ClasseBase {
    static void metodoEstatico() {
        System.out.println("Método estático na SubClasse");
    }
}
```

Neste exemplo, o método `metodoEstatico()` na `SubClasse` esconde o método `metodoEstatico()` da `ClasseBase`. Ambos são métodos estáticos e pertencem às suas respectivas classes. Se você chamar `ClasseBase.metodoEstatico()`, ele vai imprimir "Método estático na ClasseBase". Se você chamar `SubClasse.metodoEstatico()`, ele vai imprimir "Método estático na SubClasse".

7. Métodos marcados como “final” não podem ser sobreescritos nem escondidos (overriden/hiden).

Explicação: Em Java, se um método é marcado com a palavra-chave `final`, ele não pode ser sobrescrito em uma subclasse. Isso é aplicado tanto a métodos de instância quanto a métodos estáticos.

Exemplo:

```
class ClasseBase {
          public final void metodoFinal() {
              System.out.println("Método final na ClasseBase");
          }
      }

      class SubClasse extends ClasseBase {
          // Isso causará um erro de compilação
          // public void metodoFinal() {
          //       System.out.println("Tentativa de sobrescrever método
      final na SubClasse");
          // }
      }
```

No exemplo acima, qualquer tentativa de sobrescrever o método `metodoFinal()` na `SubClasse` resultará em um erro de compilação porque o método é final na `ClasseBase`.

ATENÇÃO!!

Veja este exemplo em Java:

```java
package com.exemplo.hiding;

public class Cachorro extends Mamifero {

    public String nome = "Faisca";

    @Override
    public void fazSom() {
        System.out.println("Cachorro latindo au au!");
    }

    public static void main(String[] args) {
        Cachorro c = new Cachorro();
        Mamifero m = c;
        System.out.println(c.nome);
        System.out.println(m.nome);

        c.fazSom();
        m.fazSom();
    }

}
```

```java
package com.exemplo.hiding;

public class Mamifero {

    public String nome = "Desconhecido";

    public void fazSom() {
        System.out.println("Mamífero fazendo algum som");
    }
}
```

```
Problems  Javadoc  Declaration  Console
<terminated> Cachorro (1) [Java Application] C:\Program Files\eclipse-java-2022-09-R-win32-x86_64\eclipse\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17
Faisca
Desconhecido
Cachorro latindo au au!
Cachorro latindo au au!
```

Em Java, **os campos não são polimórficos**. Este comportamento ocorre devido ao *field hiding* . O acesso a um campo é determinado pelo tipo declarado da variável, não pelo tipo real do objeto.

- Quando usamos `c.nome`, Java acessa o campo `nome` da classe `Cachorro`.
- Quando usamos `m.nome`, mesmo que `m` referencie um objeto `Cachorro`, Java acessa o campo `nome` da classe `Mamifero`, porque `m` é declarado como tipo `Mamifero`.

Para **métodos, Java usa polimorfismo dinâmico**. A versão do método que é chamada depende do tipo real do objeto em tempo de execução, não do tipo declarado da variável.

- Quando chamamos `c.fazSom()`, naturalmente o método da classe `Cachorro` é invocado.
- Quando chamamos `m.fazSom()`, mesmo que `m` seja declarado como `Mamifero`, Java sabe que o objeto real é um `Cachorro`, então ele chama o método `fazSom()` sobrescrito na classe `Cachorro`.