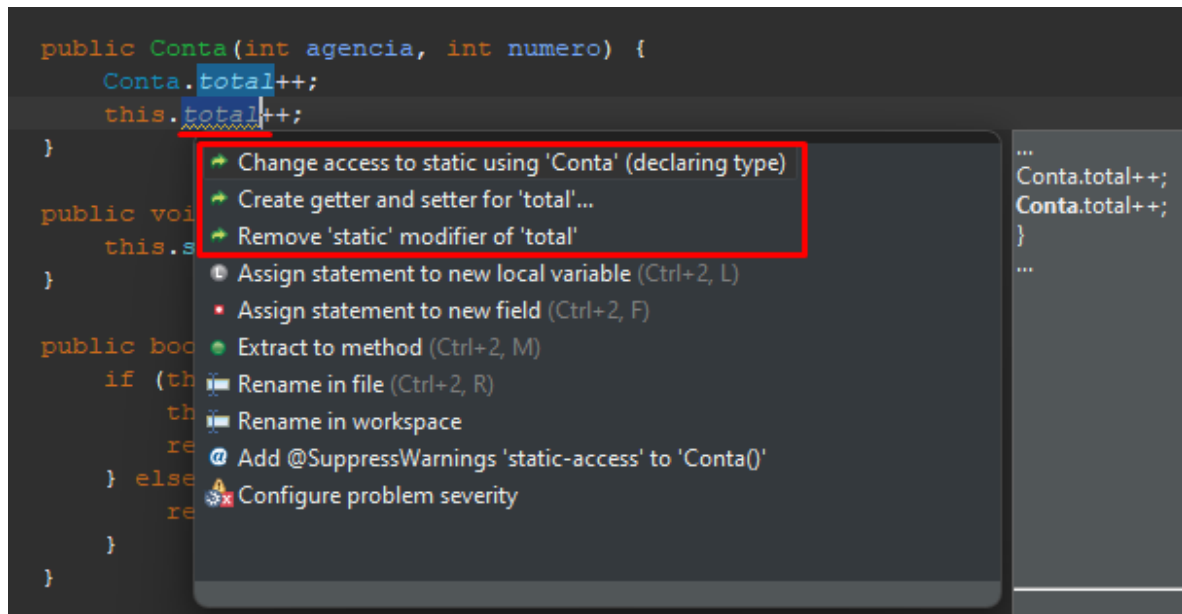


Herança

Anotações de coisas avulsas notadas:

Quando se coloca um atributo ou método como `static` a gente não pode chamá-lo com `this.atributo` ou com o `(objeto).atributo` (lembrando que objeto é uma classe já instanciada) e sim com `(classe).atributo`. Ex:



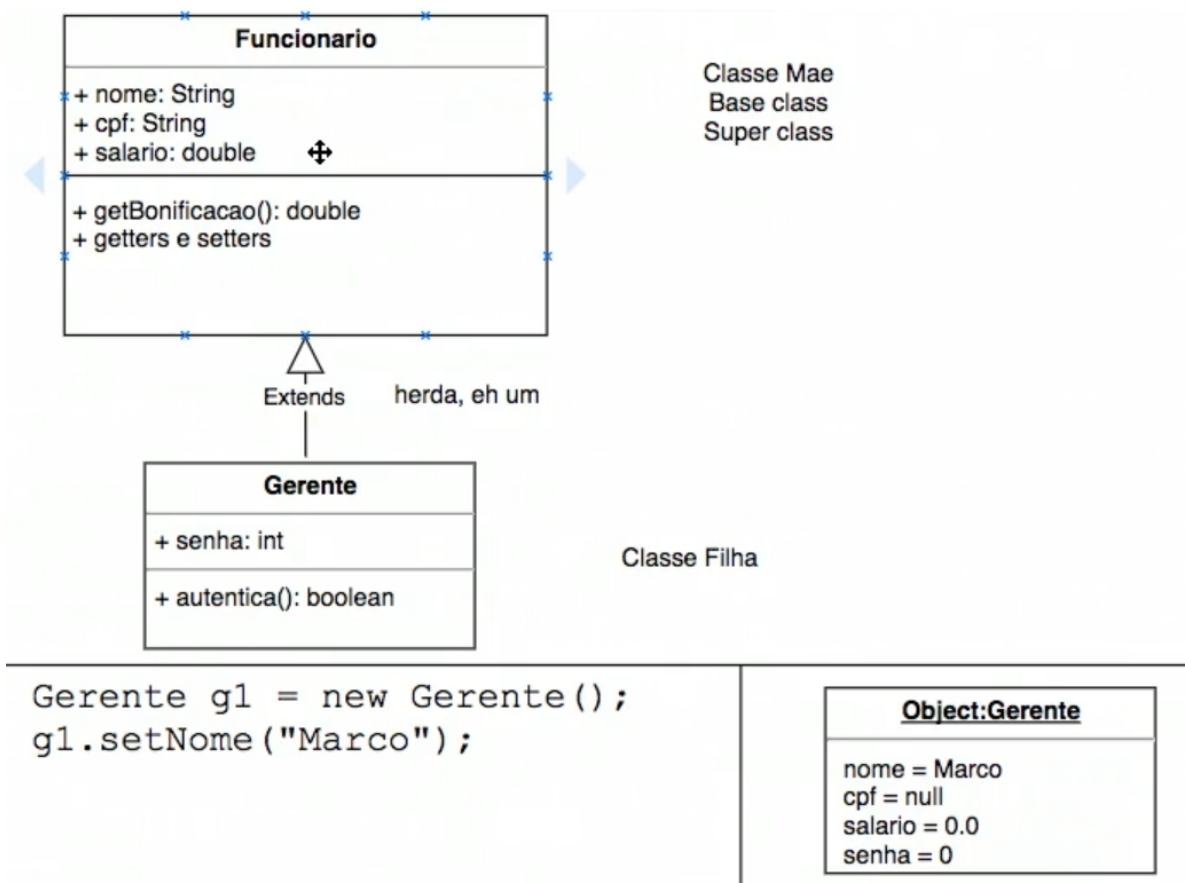
Herança também é um relacionamento entre classes, composição por exemplo é um relacionamento entre classes que é quando damos a um atributo o tipo de outra classe nossa, fugindo do convencional que é `String`, `int`, `boolean`, etc. Ex:

- ```
private int numero;
private Cliente titular;
```

Se não criamos nenhum construtor o compilador automaticamente insere um construtor padrão que é o vazio.

Ao criamos uma classe Funcionário ela deve conter atributos que tem na classe gerente e diretor, para não precisarmos em cada uma repetir métodos e atributos a gente na gerente e diretor estende a funcionário que aí conseguimos usar os métodos e atributos dela nessas classes.

Assim segue o nosso projeto até o momento:



Temos como usar o **protected** no atributo **salario** na classe **Funcionário** para escrever que o método **getBonificacao()** do gerente seja sem a comissão e apenas o salário.

```
3 usages 1 inheritor
public class Funcionario {

 2 usages
 private String nome;
 2 usages
 private String cpf;
 4 usages
 protected double salario;
```

O **protected** diz que tal atributo ou método ou até mesmo classe pode somente ser acessado por classes que estendem daquela classe ou seja **classes filhas**. **(Mas não é uma boa prática, pois é melhor usarmos os métodos get and setters).**

Ao puxarmos algum atributo ou método da classe mãe em classes filhas é uma boa prática usarmos o **super** ao invés de **this**, pois **this** é usado geralmente para nos referirmos ao que se tem na **classe** em questão e o **super** é para falar que estamos pegando aquilo da classe mãe.

A **assinatura** de um **método** é exatamente o seu tipo de acesso, o tipo de retorno que ele vai retornar, o nome e os parâmetros, exemplo abaixo:

```
public double getBonificacao()
```

Essa assinatura por boas práticas não alteramos ela quando reescrevemos o método, somente em algumas regrinhas que podemos alterar.

```
1 usage
public double getBonificacao() {
 return super.salario;
 // return this.salario;
}
```

### Super com métodos:

Podemos também chamar um método da **classe mãe (super class)** com o **super** e usar ele dentro deste mesmo método só que **reescrito na classe filha**. Ex:

```
public double getBonificacao() {
 return (super.getBonificacao()) + super.salario;
}
```

### Polimorfismo:

Objeto não troca o tipo o que pode variar é a referência.

Ou seja o que fica do lado esquerdo (**Funcionario**) pode variar já o que fica no direito (**Gerente()**) não.

```
Funcionario g1 = new Gerente();
```

Porém se a classe mãe for a que está sendo instanciada (lado direito) e uma filha quiser instancia-la (lado esquerdo) não vai dar certo, pois a mesma não é do tipo da filha e sim a filha estende o tipo da mãe.

Com isso só conseguimos utilizar métodos e atributos da classe mãe **Funcionario**.

Polimorfismo nada mais é que um objeto que pode ser referenciado através de uma referência do mesmo tipo ou de uma referência mais genérica (classe mãe).

Para deixarmos claro essa ideia segue o exemplo abaixo:

Imagina que temos uma classe que cuida de todas as bonificações somando o gasto somente de bonificações que a empresa terá, sem o polimorfismo nós teríamos que criar um método para cada classe que representa um tipo de funcionário:

```

public class ControleBonificacao {

 private double soma;

 public void registra(Gerente g) {
 double boni = g.getBonificacao();
 this.soma = this.soma + boni;
 }

 public void registra(Funcionario f) {
 double boni = f.getBonificacao();
 this.soma = this.soma + boni;
 }

 public void registra(EditorVideo ev) {
 double boni = ev.getBonificacao();
 this.soma = this.soma + boni;
 }

 public double getSoma() {
 return soma;
 }
}

```

Já com o polimorfismo podemos facilmente retirar todo esse código duplicado, a classe funcionário é a classe mãe das classes (gerente, programador, diretor, etc) com isso todas são do tipo funcionário por trás dos panos, então só precisamos de um método que registra recebendo como parâmetro um objeto do tipo Funcionario.

```

public class ControleBonificacao {

 private double soma;

 public void registra(Funcionario f) {
 double boni = f.getBonificacao();
 this.soma = this.soma + boni;
 }

 public double getSoma() {
 return soma;
 }
}

```

Sempre o método da classe especificada (instanciada) que vai ser executado.

```

public static void main(String[] args) {

 Gerente g1 = new Gerente();
 g1.setNome("Marcos");
 g1.setSalario(5000.0);

 Funcionario f = new Funcionario();
 f.setSalario(2000.0);

 EditorVideo ev = new EditorVideo();
 f.setSalario(2500.0);

 ControleBonificacao controle = new ControleBonificacao();
 controle.registra(g1);
 controle.registra(f);
 controle.registra(ev);

 System.out.println(controle.getSoma());
}

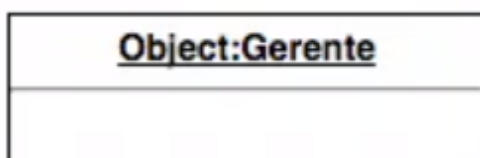
```

Nesse caso o método de bonificação da classe **gerente** que vai ser usada na classe de **ControleBonificacao**.

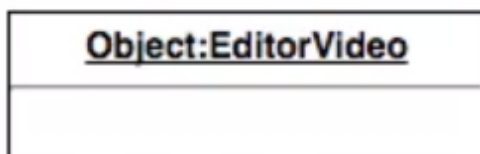
#### RECAPITULANDO:

Podemos ter um objeto do tipo gerente que tem uma referencia do tipo Funcionario. **(MAS NÃO PRECISAMOS COLOCAR FUNCIONARIO PODE SER APENAS O GERENTE DE REFERÊNCIA JÁ QUE O MESMO É UM FUNCIONARIO, POIS É A CLASSE FILHA, FAZEMOS ISSO SOMENTE PARA VERMOS).**

**Funcionario gerente = new Gerente();**

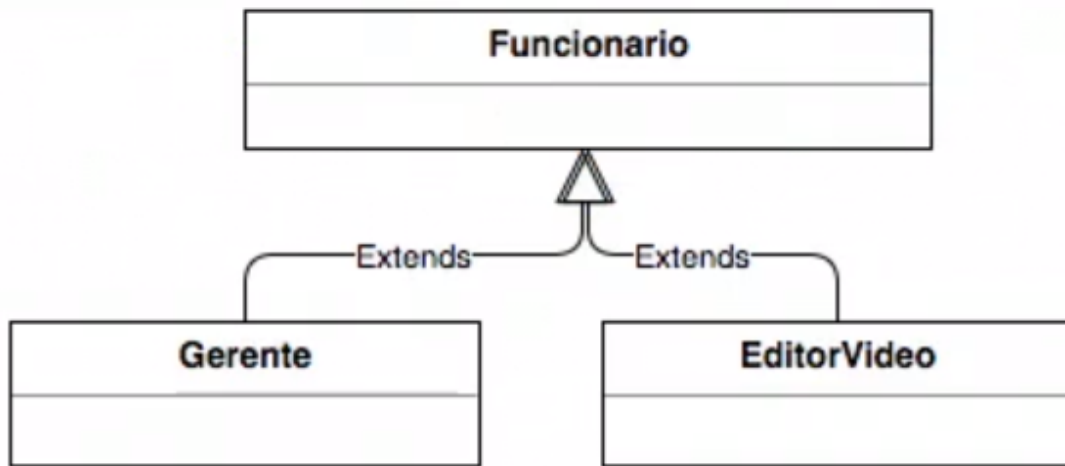


**EditorVideo editor = new EditorVideo();**



Reparem que temos a mesma referência genérica (Funcionario) que aponta para dois objetos de tipos diferentes.

Isso funciona porque o gerente é um Funcionario.



Com o **polimorfismo** podemos criar vários tipos novos como ex: fotografo, pintor, etc. desde que eles estendem de Funcionario, os métodos que aceitem como parâmetro objetos do tipo Funcionario não precisam ser alterados em nada.