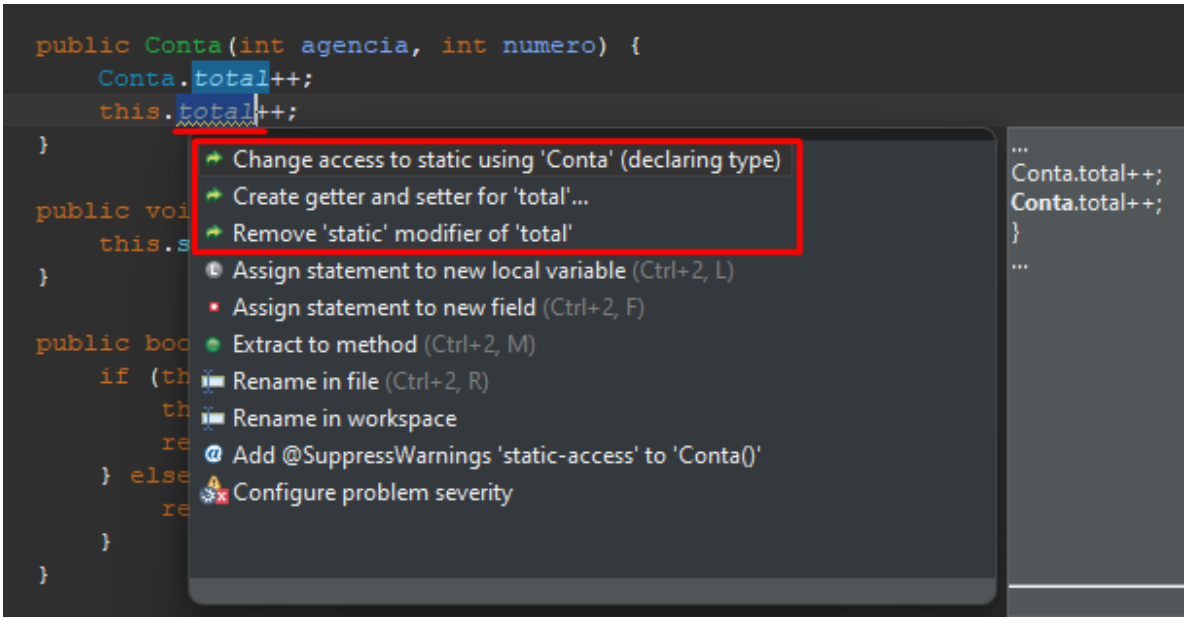
Herança

Anotações de coisas avulsas notadas:

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



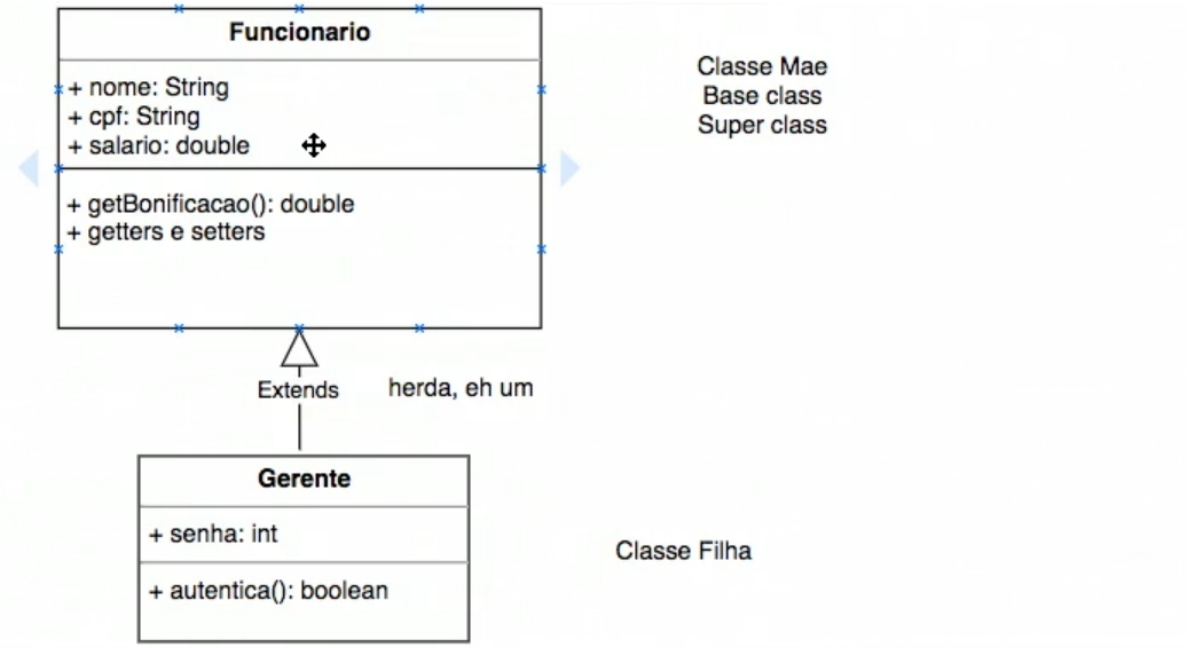
Herança também é um relacionamento entre classes, composição por exemplo é um relacionamento entre classes que é quando damos a um atributo o tipo de outra classe nossa, fugindo do convencional que é String, int, boolean, etc. Ex:

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

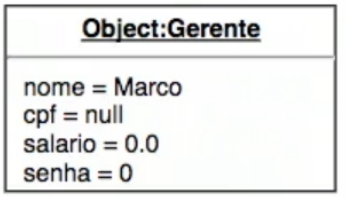
Se não criamos nenhum construtor o compilador automaticamente insere um construtor padrão que é o vazio.

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:
  ```




```
Gerente g1 = new Gerente();
g1.setNome("Marco");
```



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 (instânciada) que vai ser executado.

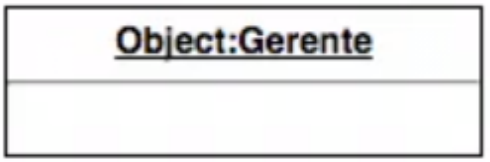
```java
public static void main(String[] args) {

    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.

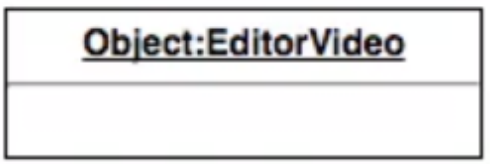
RECAPITULANDO:

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).**

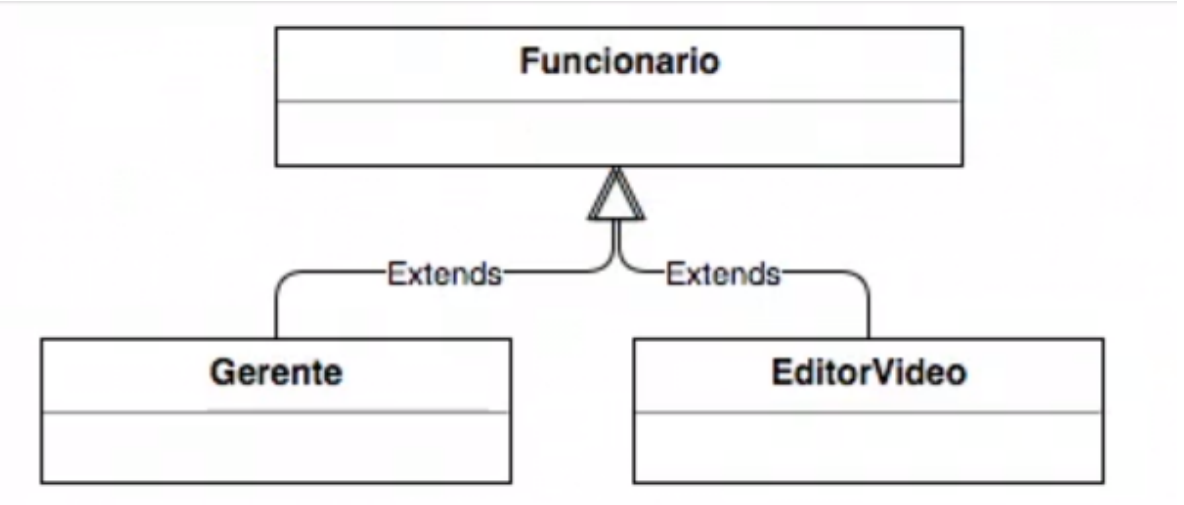
Funcionario gerente = new Gerente();



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.
pontos abordados até agora:

- objetos não mudam de tipo;
- a referência pode mudar, e aí entra o polimorfismo;
- o polimorfismo permite usar referências mais genéricas para a comunicação com um objeto;
- o uso de referências mais genéricas permite desacoplar sistemas.
- **Na herança na classe filha herdamos os métodos e atributos, mas não herdamos os construtores, o construtor é somente da classe ele nunca é passado para outras classes**.

Continuação herança:

Se uma classe não tem um construtor padrão, ou seja seu construtor foi escrito e não é vazio, ao gerarmos uma classe filha ela ira ter erro, pois essa classe filhatambém não pode ter um construtor vazio pelo motivo de o Java ao ver essa classe filha com o construtor vazio ele vai tentar pegar o construtor da mãe e isso não pode a não ser que a mãe tenha um construtor vazio. **(Mesmo se não colocarmos o super o Java coloca implicitamente sem precisar também de escrevermos um construtor vazio, pois construtores também se não escritos ficam vazios implicitamente)**.

- ```
  public class ContaCorrente extends Conta {

      public ContaCorrente() {
          super();
      }
  }
  ```

  Para resolver isso podemos também chamar um construtor especifico colocando os parâmetros no nosso construtor da classe filha iguais aos da classe mãe e colocando no Super os objetos que serão enviados para esse construtor. (Podemos ter um construtor específico e outro vazio mãe, com isso poderíamos sim não especificar nada e utilizar o construtor vazio na filha).

  - ```
    public class ContaCorrente extends Conta {

        public ContaCorrente(int agencia, int numero) {
            super(agencia, numero);
        }
    }
    ```

    A anotação @Override é muito útil para quando formos reescrever um método da classe mãe na classe filha, ele impede de errarmos algo na reescrita, por exemplo: Ele impede que ao reescrevermos o método **saca** a gente coloque **sacar** que é escrito de maneira diferente, impedindo assim que criemos um método novo ao invés de reescrever um já existente por erro.

Classes abstratas:

Quando criamos uma classe mãe geralmente não queremos que existam algum objeto instanciado do tipo dela e sim de seus filhos que são classes mais específicas, porém que tem todos os métodos e atributos da classe mãe, isso ocorre porquê a classe mãe é um conceito, algo abstrato, exemplo: Classe mãe Funcionário, não deveríamos ter uma pessoa que é somente Funcionária e sim com o cargo dela, no caso o modelo é somente Funcionário e as classes especificas representarem realmente sua função no trabalho, *Classes abstratas podem ter construtores, pois os mesmo podem ser chamados em suas classes filhas*.

O que não é para termos:

- ```
  Funcionario f = new Funcionario();
  ```

  Para impedirmos isso podemos colocar na nossa classe que ela é abstrata:

- ```
  public abstract class Funcionario {
  ```

  E ele fará com que não seja possível instanciar um objeto da maneira abaixo:

- ```
  Funcionario = new Funcionario();
  ```
  Mas podemos instanciar objetos com suas classes filhas:
- ```
  Designer d = new Designer();
  ```

Métodos abstratos:

Assim como existem classes abstratas também existe métodos abstratos, são métodos que temos na classe mãe que não usamos na classe mãe e sim nas filhas, geralmente métodos que são personalizados para cada um das filhas e a mãe não deve ter uma execução do mesmo e sim somente disponibilizar o método para as filhas sobrescreverem.

Exemplo de uso de método que vem da classe mãe:

- ```
  public void registra(Funcionario f) {
      double boni = f.getBonificacao();
      this.soma = this.soma + boni;
  }
  ```
  Nesse caso não queremos pegar a bonificação da classe mãe (Funcionario) e sim das filhas que são os cargos, mas se fossemos passar o parâmetro de cada tipo íamos ter que fazer um método deregistra() para cada classe filha, então usamos o método getBonificação() da classe mãe e sobrescrevemos o método nas classes filhas, assim o registra() consegue aceitar um objeto do tipo Funcionario e pegar a bonificação dos mesmos já que aclasse mãe também contém o método em questão.

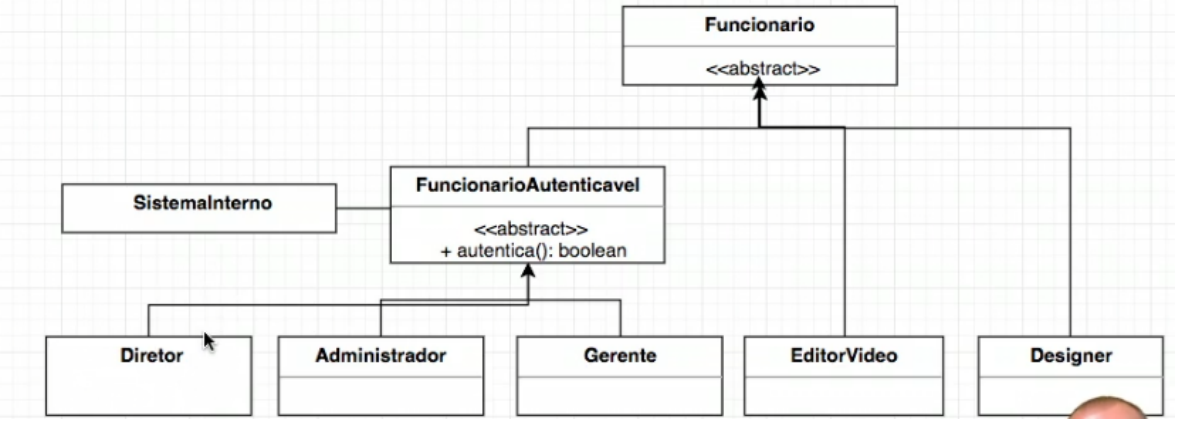
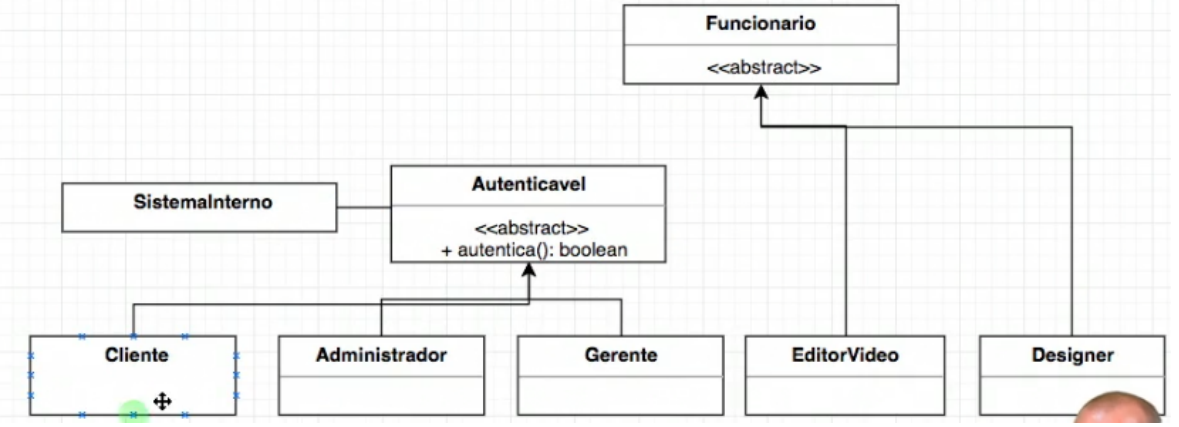
  Para o método getBonificacao() da classe mãe ser somente vazio para disponibilizar o método somente a gente colocar ele como abstrato:
- ```
  public abstract double getBonificacao();
  ```
  Sendo assim podemos concluir que se não existe uma implementação padrão esse método não é concreto e sim abstrato.
  - Abstrato na classe significa que não podemos instanciar objetos dessa classe, no método significa que ele não tem corpo (implementação), mas nas classes filhas tem que ter implementação do mesmo se não tiver gera erro, pois métodos abstratos devem ser implementados em todas as classes filhas a não ser que ela também seja abstrata.

Herança múltipla:

Quando temos uma classe filha que é usada como parâmetro em algum método específico de uma classe e outra classe filha também tem que ter esse método e precisa ser passada como parâmetro no mesmo acaba que teríamos que duplicar código.

- ```
  public class SistemaInterno {

      private int senha = 2222;

      public void autentica(Gerente g) {
          boolean autenticou = g.autentica(this.senha);
          if(autenticou) {
              System.out.println("Pode entrar no sistema!");
          } else {
              System.out.println("Não pode entrar no sistema!");
          }
      }

  }
  ```

  Uma classe administrador herdada da classe mãe Funcionario ocasionaria nesse problema de duplicação de código, porém resolver isso é muito simples, a classe filha Gerente tem como **mãe** a classe Funcionario, ou seja ela tem todos os métodos e atributos da classe Funcionario e é considerado um Funcionario, então já que nosso Administrador também é um Funcionario e precisa acessar um método que recebe um objeto do tipo gerente nós podemos ao invés de estender na classe admistrador a classe Funcionario estendermos a classe Gerente que aí evitaríamos essa duplicação de código e o método iria aceitar Administrador como parâmetro já que é um Gerente também.
- ```
  public class Administrador extends Gerente {
  }
  ```
- ```
  public class TestaSistema {

      public static void main(String[] args) {
          Gerente g = new Gerente();
          g.setSenha(2222);

          Administrador a = new Administrador();
          a.setSenha(2453);

          SistemaInterno si = new SistemaInterno();
          si.autentica(g);

          si.autentica(a);

      }

  }
  ```

Resultado da execução acima:
  ```

- ```
  Pode entrar no sistema!
  Não pode entrar no sistema!
  ```

  Ou como forma de melhor segurança e design do projeto podemos criar uma classe herdada de Funcionario que tenha esse método autentica do gerente e que ao invés de gerente ser aceito como parâmetro no autentica de SistemaInterno ser essa classe no lugar, sendo assim essa é uma melhor maneira.



  Mas e se tivermos uma classe que não era pra ser funcionário, mas deveria ter acesso ao Sistema Interno? Simplesmente a herança nesse caso não é uma solução já que no Java não temos herança múltipla, se seguirmos com a herança iria fazer com que o mesmo se tornasse um Funcionario para isso que seria errado ou então faria com que Gerente e Administrador deixassem de ser funcionários.



  Para resolvermos isso usaremos as interfaces, pois herança nesse caso não nos atende.

Interfaces:

Podemos pensar na interface como uma classe abstrata com todos os métodos abstratos, dentro dela não há nada concreto, então uma classe não contém atributos e nem construtores somente métodos abstratos (sem implementação).
  ```

Quem implementar a interface precisa implementar todos os métodos também fazendo-os serem concretos (dando código a eles).

```
//contrato Autenticavel
    //quem assinar esse contrato, precisa implementar
        //metodo setSenha
        //metodo autentica
public abstract interface Autenticavel {

    public abstract void setSenha(int senha) ;

    public abstract boolean autentica(int senha) ;
}
```

Diferente das classes que só podemos estender uma as interfaces podemos implementar quantas quisermos, então podemos concluir que interfaces são um contrato que tem as obrigações (métodos obrigatórios) que temos que escrever.