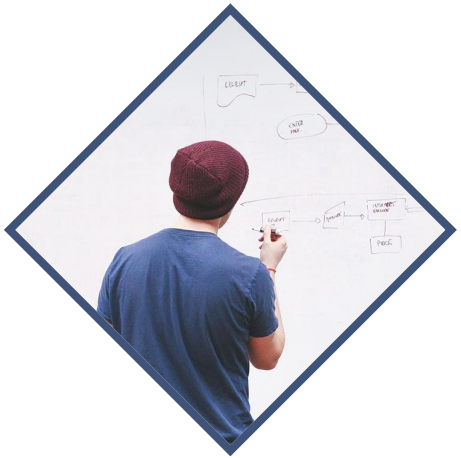
AULA 02: Herança e Polimorfismo

Agradecimentos ao material do Profº **Bruno Nogueira**

Lecionadores



Kaio Mitsuharu Lino Aida

kaiomudkt@gmail.com

Mateus Ragazzi Balbino

mateusragazzi.b@gmail.com

Acadêmicos de Sistemas de Informação

Mário de Araújo Carvalho

mariodearaujocarvalho@gmail.com

Acadêmico de Ciência da Computação.

2

Aula

Herança e Polimorfismo

Especificações Java SE 12

https://docs.oracle.com/javase/specs/jls/se12/html/index.html

Revisão - Introdução Orientação a Objetos

- **Modelagem e abstração voltadas para o problema a ser solucionado**
- Programa é **adaptado** ao problema por meio da **criação de novos objetos**
- Programa é uma **coleção de objetos** que interagem entre si.

Revisão - Conceitos OO

- Abstração
- Encapsulamento
- Herança
- Polimorfismo
- Modularidade

Revisão

- **Classe**
  - É um **conjunto de objetos com características comuns**. Uma classe é como um **modelo** para a criação de objetos, que têm as mesmas características da classe à qual pertence.

Revisão

- **Método construtor:**
  - Criar o objeto em memória, ou seja, **instanciar a classe que foi definida**
- **Objeto:**
  - É um elemento de uma classe que modela da mesma forma, mas que podem ter características diferentes.

Revisão

- **Instancias:**
  - Uma instância de uma classe é um novo objeto criado dessa classe, com o operador **new**. Instanciar uma classe é criar um novo objeto do mesmo tipo dessa classe. Uma classe somente poderá ser utilizada após ser instanciada.

Revisão

- **Modificadores de acesso:**
  - Um objeto não deve **nunca manipular diretamente** os dados internos de outros objetos
    - Public, Protected, Privete e Default
- **Métodos de acesso:**
  - Manipulação de dados de outras classes deve ser feito **por meio de métodos**
    - **Getters e Setters**

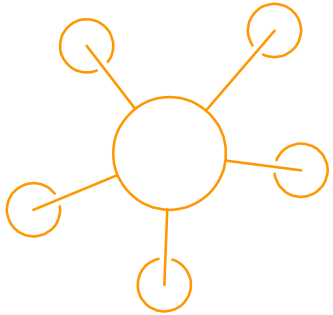
Revisão

- **Método main**
  - O método main é um método associado à classe e não a um objeto específico da classe (static)
  - Determina o ponto de início de execução de qualquer aplicação Java

- Vamos fazer uma classe que simule uma conta bancária:
  - Deve armazenar o saldo, o nome do titular e o cpf
  - Deve armazenar o limite do cheque especial
  - Deve permitir depósito e retirada
    - Caso a retirada seja maior que o saldo, não pode exceder o limite do cheque especial
  - Deve permitir o cálculo dos juros mensais
    - Taxa fixa de 2% ao mês no cheque especial
  - Deve imprimir um resumo da conta, com:
    - Nome do titular e saldo da conta

Orientação a Objetos

Herança



Conceito de herança

Herança é um princípio de orientação a objetos, que **permite que classes compartilhem atributos e métodos, através de "heranças"**. Ela é usada na intenção de reaproveitar código ou comportamento generalizado ou especializar operações ou atributos. O conceito de herança de várias classes é conhecido como herança múltipla.

Herança

Simpson	Burns
acordar();	acordar();
dormir();	dormir();
comer();	comer();
conversar();	conversar();
trabalhar();	trabalhar();
ser_tolo();	ser_mau();
viajar();	ganhar_din();
...	...

Herança

- Relacionamento “**é-um**”
- Uma classe mais especializada (subclasse) **herda as propriedades (métodos e atributos)** e semântica de uma classe mais geral (superclasse)

Herança

- Uma subclasse pode **sobrescrever** o comportamento de uma superclasse
- **Promove o reuso**
  - Economiza tempo no desenvolvimento
  - Aumenta a qualidade

Herança - Exemplo

SerAmarelo

Peso : float
Idade: inteiro
Nome: char

acordar();
dormir();
comer();
conversar();
trabalhar();

Simpson

ser_tolo();
viajar();
...

Burns

ser_mau();
ganhar_din();
...

Classe a ser extendida

```
public class Pai {
    private String nomePai;
    public Pai(String nomePai) {
        this.nomePai = nomePai;
    }
}
```

Classe herdeira

```
public class Filho extends Pai{
    private String nomeFilho;
    private int idade;

}
```

Herança Simples

Em Java, **há herança simples**

- Uma classe só pode **herdar** propriedades de uma **única classe** "pai" – uma única superclasse
- **Uma superclasse, entretanto, pode ter infinitas subclasses!**

- Uso da diretiva extends nas subclasses

Herança Múltipla

- Herança múltipla, em orientação a objetos, é o conceito de **herança de duas ou mais classes**. Ela é implementada nas linguagens de programação C++ e em Python, por exemplo.
- Mas permite que uma classe implementa várias **interfaces**.

Classe *Object* em Java

- Por padrão, **toda classe** em Java **estende** a classe ***Object***
  - ***Mesmo sem uso explícito do "extends"***
  - *Tem acesso aos membros internos* ***visíveis*** *da classe Object.*

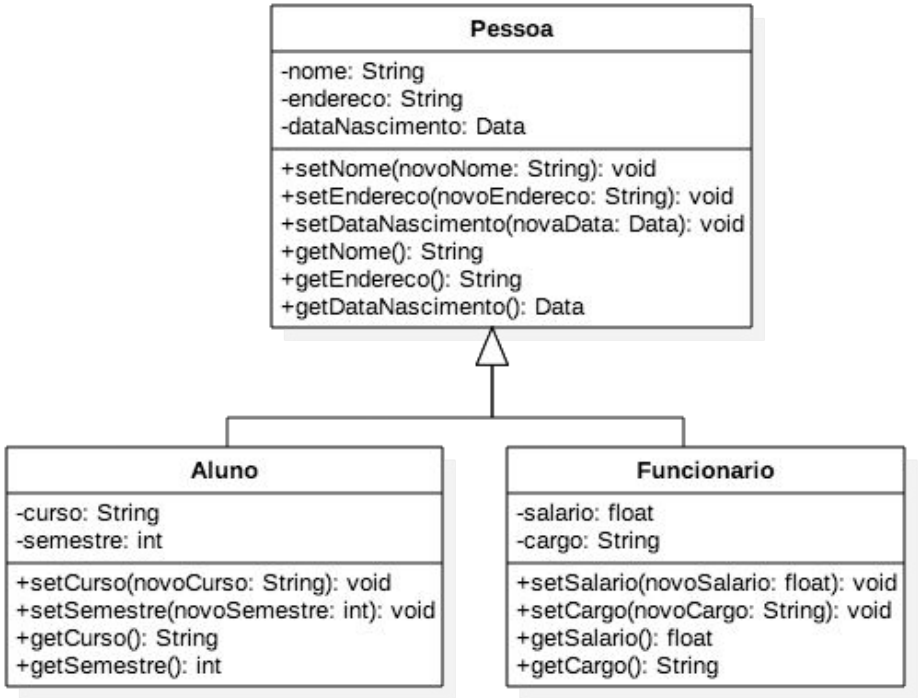
Classe Object

- *Classe Object possui métodos e atributos úteis à manipulação de objetos em Java*
  - *boolean equals(Object obj)*
  - *void finalize() throws Throwable*
  - *final Class getClass()*
  - *int hashCode()*
  - *String toString()*
  - *Etc.*

Herança - Super

A herança é um princípio da POO que permite a criação de novas classes a partir de outras previamente criadas. Essas **novas classes são chamadas de subclasses,** ou classes derivadas; e as **classes já existentes**, que deram origem às subclasses, são chamadas de **superclasses**, ou classes base.

Herança - Super Exemplo



Herança - Super

- A classe Pessoa representa o conjunto de todas as pessoas, sejam alunos ou funcionários.
  - A classe pessoa é a superclasse
  - Define atributos e métodos genéricos – comuns às classes que herdam – subclasses de Pessoa (Aluno e Funcionario)

Herança - Super

- As classes Aluno e Funcionario representam especificidades encontradas apenas em alunos e funcionários, respectivamente
  - Subclasses de Pessoa
  - Herdam o que há na classe Pessoa
    - Atenção à visibilidade dos membros!

Super - Método construtor da classe herdeira

```
public Filho(String nomeFilho,
                    int idade, String nomePai) {
  super(nomePai);
    this.nomeFilho = nomeFilho;
    this.idade = idade;
}
```

Final

A palavra-chave final em Java serve para **atribuir valores constantes**, ou seja, que a partir da sua declaração, **seus valores não poderão mais serem alterados**.

Final - Variável

- Variável
  - Constante, geralmente são “static”
  - Exemplos PI e Pitágoras (√2)
  - Não possui método setter

```
final float PI = 3.1416F;
```

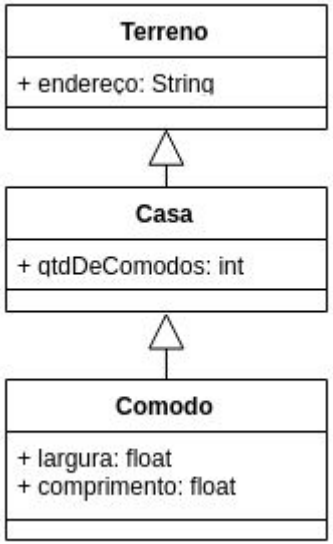
Final - Método

Um método final é imutável e não pode ser sobrescrito, mas é herdado.

```
public final int soma (int a, int b) {
        return a + b;
}
```

Final - Classe

- **Classe**
  - Quando é aplicado na classe, **não permite estende-la**.



Final - Classe

- Em alguns contextos, podemos querer **garantir que uma classe nunca seja estendida**
  - Dessa forma, não poderá ser superclasse de nenhuma outra classe
  - Por exemplo, para garantir que o comportamento da superclasse nunca seja alterado

Final - Classe

- Fará mais sentido quando falarmos de polimorfismo
- Por enquanto, basta saber que classes final não podem ser estendidas

Final - Classe

- No exemplo abaixo, nenhuma outra classe estará abaixo da classe Aluno na hierarquia de classes

```
public final class Aluno extends Pessoa {
    . . .
}
```

Abstract

- **Classes Abstratas**
  - **Uma classe abstrata não pode ser instanciada**, ou seja, não pode ser chamada pelos seus construtores. Se houver alguma declaração de um método como abstract (abstrato), a classe também deve ser marcada como abstract.

Abstract

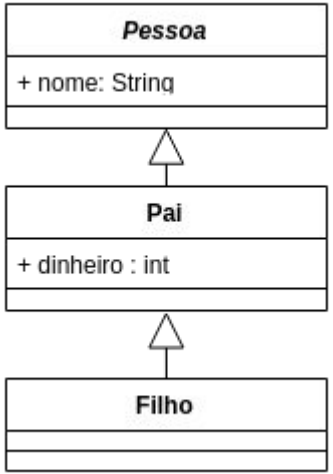
- **Expor características essenciais enquanto se esconde outros detalhes irrelevantes**
  - Reduz a complexidade do código
  - Organiza o projeto / código
- **Classes são abstrações de conceitos**
  - Devem ser o mais simples possível
  - Sem, no entanto, perder a representatividade ante ao conceito no mundo real

Abstract

- Ao subir na hierarquia de heranças, as classes tornam-se mais genéricas
  - **Podem não representar algo tangível**
  - Tornam-se modelos para classes

Abstract

- Nesses casos, pode ser desejável **não permitir a instanciação de objetos destas classes**
  - Por exemplo, não faz muito sentido permitir objetos da classe Pessoa serem instanciados



Abstract

- Em geral (mas não obrigatoriamente), classes abstratas possuem um ou mais métodos abstratos
  - **Sem implementação definida**
  - Também definidos com o modificador abstract

Abstract

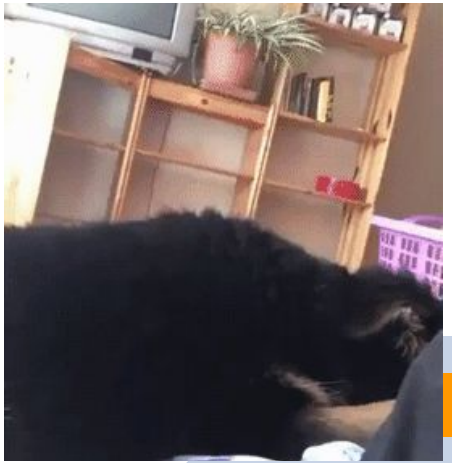
```
public abstract class Animal {
        public void comer (Comida comida) {
            ....
        }
        public abstract void falar ();
}
```

Abstract

- **Métodos Abstract**
  - Esses métodos são implementados em suas subclasses com o objetivo de definir seu comportamento específico. O **método abstrato define apenas a assinatura do método e, portanto, não tem código**.

Exemplo Método abstrato

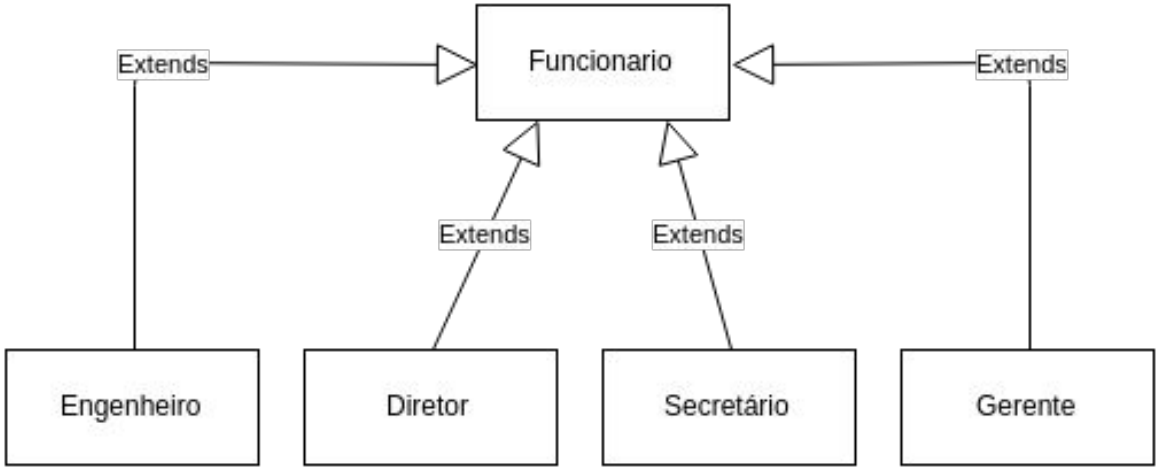
```
public class Cachorro extends Animal {

        public void falar () {
            System.out.print("Au Au");
        }

}
```



Herança vs Acoplamento

- Note que o uso de herança aumenta o acoplamento entre as classes, isto é, **o quanto uma classe depende de outra.**
- A relação entre classe mãe e filha é muito forte e isso acaba fazendo com que o programador das classes filhas tenha que conhecer a implementação da classe pai e vice-versa- **fica difícil fazer uma mudança pontual no sistema.**

Herança vs Acoplamento



Herança vs Acoplamento

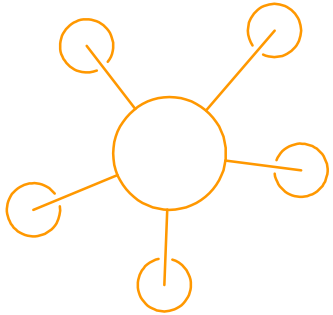
Por exemplo, imagine se tivermos que mudar algo na nossa classe Funcionario, mas não quiséssemos que todos os funcionários sofressem a mesma mudança. Precisaríamos passar por cada uma das filhas de Funcionário verificando se ela se comporta como deveria ou se devemos sobrescrever o tal método modificado.

Herança vs Acoplamento

Esse é um problema da herança, e não do polimorfismo, que resolveremos mais tarde com a ajuda de **Interfaces**.

Orientação a Objetos

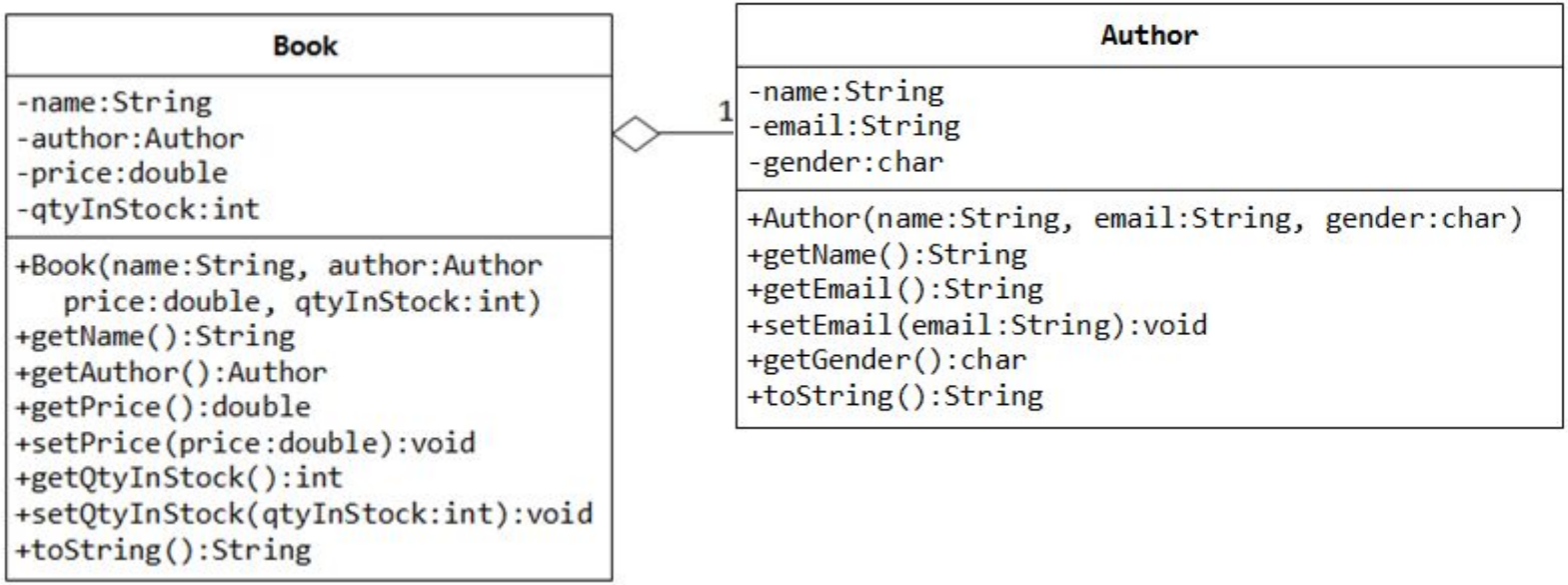
UML Classe



UML de Classe

É o mais importante e utilizado do diagrama da **UML**

UML de Classe



UML de Classe

Uma classe num Diagrama de Classes (ou até mesmo no código fonte) é apenas um conceito. **Um conceito em forma de desenho** (se num diagrama) ou texto (se em código fonte).

Quando a Classe é materializada através de um software, (quando o software está “rodando”) torna-se um objeto (isso se dá quando é alocado um ponteiro de memória para esta classe).

UML de Classe

O diagrama de classes **ilustra graficamente como será a estrutura** do software (em nível micro ou macro – veremos adiante sobre as possibilidades de uso do diagrama), e como cada um dos componentes da sua estrutura estarão interligados.

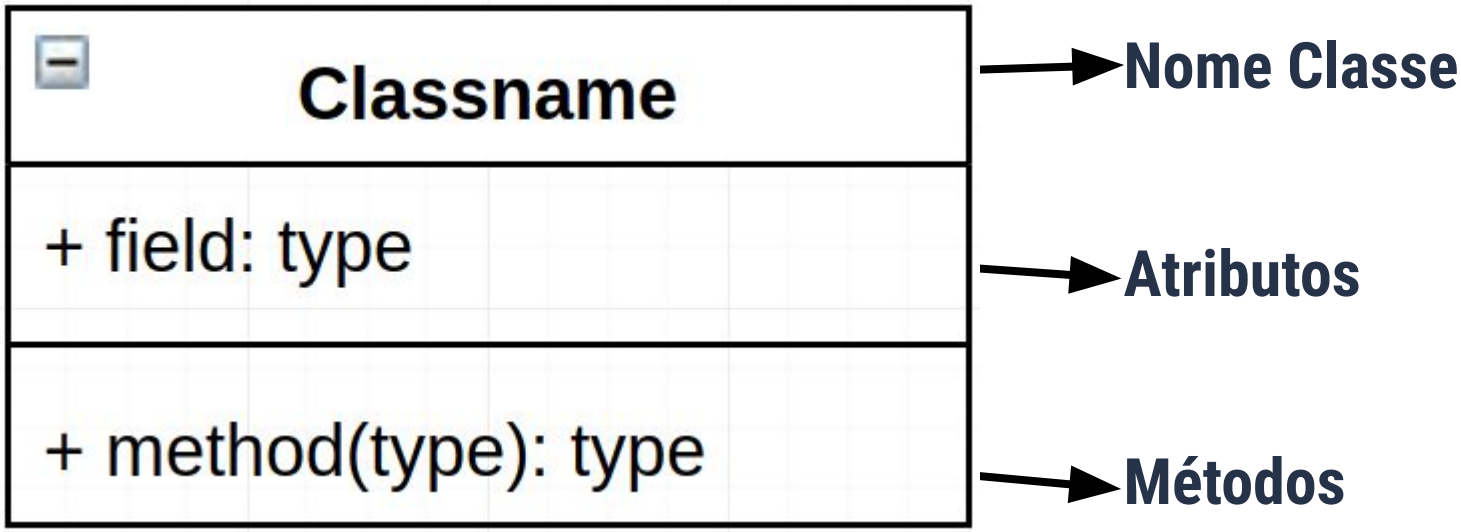
UML de Classe

- Objetivo principal
  - Especificações dos componentes do software e como estes se interligam, do ponto de vista estrutural, ou seja, **uma visão estática de como as classes se relacionam.**

UML de Classe - Classe

- Classe
  - É a classe propriamente dita. Usamos este elemento quando queremos demostrar visualmente a classe no diagrama (exemplos mais à frente).

UML Classe


Classname
+ field: type
+ method(type): type
Nome Classe
Atributos
Métodos

UML Classe - Classe

- Nome da classe
  - Algumas mudanças de acordo com a classe
    - Abstract
    - Interface
    - Final

UML Classe - Tipos de Visibilidade

- Pública (**+**)
- Protegida (**#**)
- Privada (**-**)

UML Classe - Método

- São apenas declarados neste diagrama
  - Não define a implementação
- Sintaxe
  - + setCliente(cliente: GestãoCliente):void
  - + getCliente(): GestaoCliente

UML Classe - Atributos

- Permite a identificação de cada objeto de uma classe
- Valores dos atributos podem variar de instância para instância
- Atributos podem conter o tipo de dados a ser armazenado
  - boolean, int, double, String, etc

UML Classe - Atributos - Sintaxe

- Modificador de acesso
- Nome
- Tipo do atributo

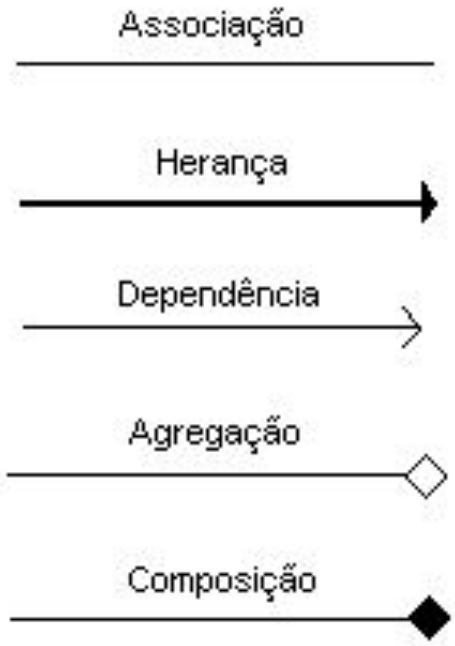
UML Classe - Associação - Multiplicidade

0..1	**No máximo um.** Indica que os objetos da classe associada não precisam obrigatoriamente estar relacionados.
1..1	**Um e somente um.** Indica que apenas um objeto da classe se relaciona com os objetos da outra classe.
0..*	**Muitos.** Indica que podem haver muitos objetos da classe envolvidos no relacionamento.
1..*	**Um ou muitos.** Indica que há pelo menos um objeto envolvido no relacionamento.
3..5	Valores específicos.

Tipos de relacionamentos

- **Classes possuem relacionamentos entre si**
  - Compartilham informações
  - Colaboram umas com as outras
- Principais Relacionamentos
  - **Associação**
  - **Agregação e Composição**
  - **Herança**
  - **Dependência**

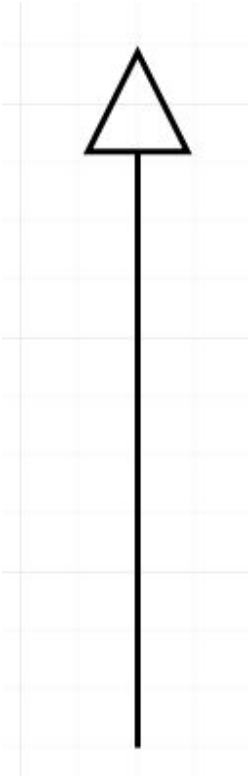
UML Classe - Conectores dos relacionamentos



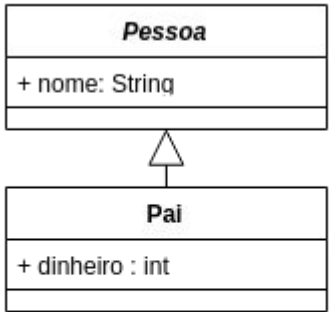
Herança - Conector com seta em duas das pontas

É um tipo de relacionamento onde a classe generalizada (onde a “ponta da seta” do conector fica) **fornece recursos para a classe especializada (herdeira)**. Excetuando conceitos mais avançados (como padrões de projeto, interfaces, visibilidades específicas etc.), tudo que a classe mãe (generalizada) tem, a filha (especializada) terá.

Herança - Conector



Herança



Herança - Em código

```
public class Registradora extends Venda {
    private int id;
    private Venda vendaCorrente;
}
```

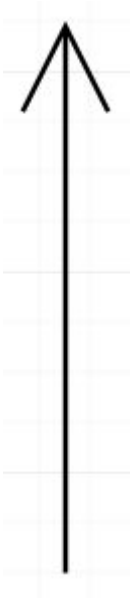
Herança

- Uma subclasse herda:
  - Atributos
  - Operações
  - Relacionamentos
  - Uma subclasse pode:
    - Adicionar novos atributos, operações e relacionamentos.
    - Redefinir operações herdadas.

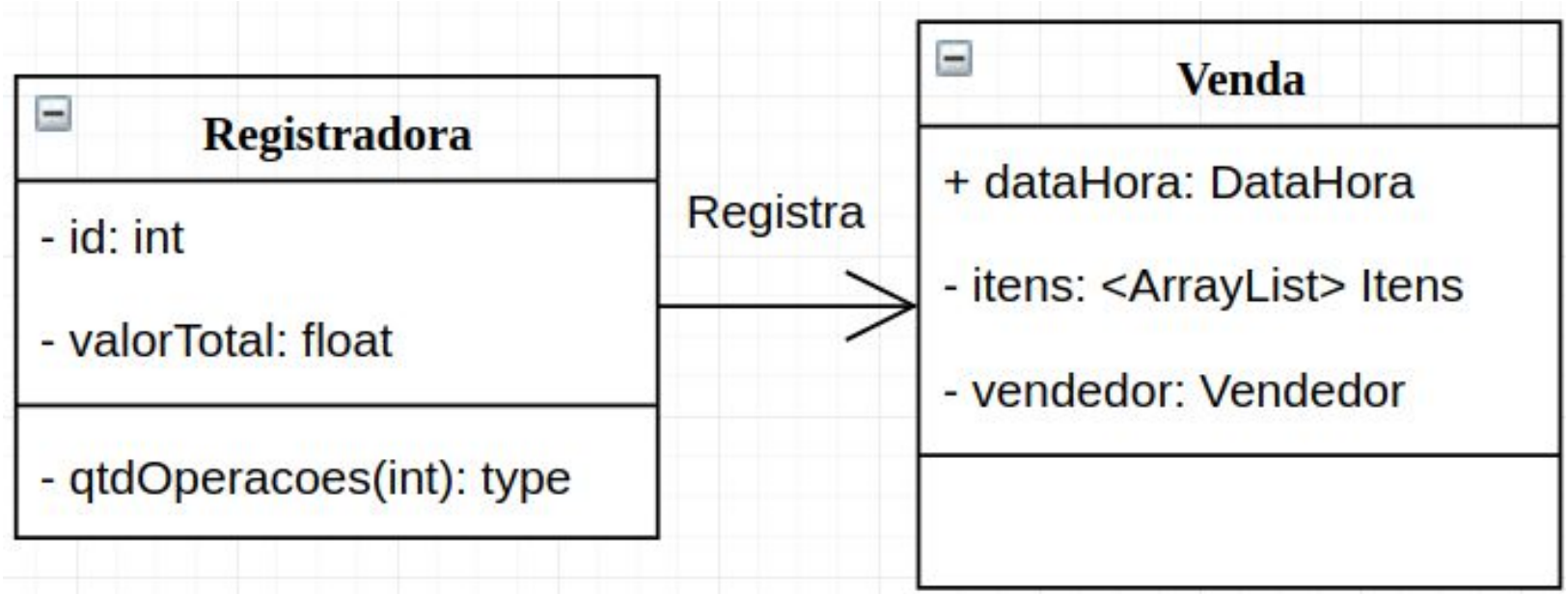
Associação

É um tipo de relacionamento usado entre classes. Aplicável a classes que são independentes **(vivem sem dependência umas das outras)**, mas que em algum momento no ciclo de vida do software (enquanto ele está em execução) podem ter alguma relação conceitual.

UML Classe - Associação - Conector



Associação - Conectores com seta vazia



Associação - Em código

```
public class Registradora {
    private int id;
    private Venda vendaCorrente;
}
```

Agregação - Conector com uma seta fechada vazia

Agregação – conector com um “diamante” vazado na ponta – é um tipo de relacionamento onde a **classe agregada usa outra classes para “existir”, mas pode viver sem ela**. Por exemplo, a classe “CorpoHumano” possui uma agregação com a classe “Mao”. Sem a “Mao” a classe “CorpoHumano” pode existir. Mais detalhe neste post completo sobre Agregação.

UML Classe - Agregação - Conector

Agregação

- Tipo especial de associação
- Demonstra que as informações de um objeto precisam ser **complementadas** por um objeto de outra classe

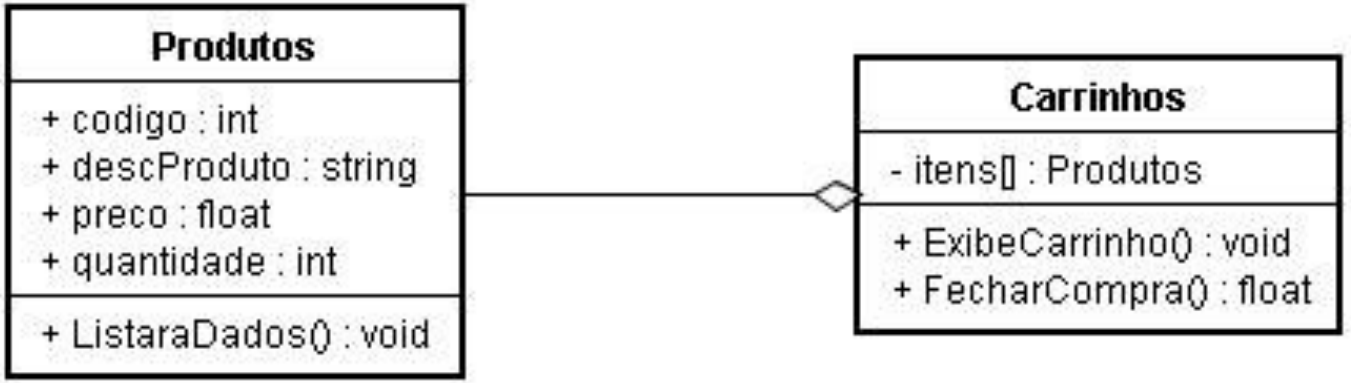
Agregação

- Associação **todo-parte**
  - **Objeto-todo**
  - **Objeto-parte**

Agregação

- Na Agregação, a existência do **Objeto-Parte faz sentido, mesmo não existindo o Objeto-Todo**.
- Relação 'parte-de', 'tem-um', 'todo-parte'
- Vejamos o exemplo Time-Atleta:

Agregação


Produtos
+ codigo : int
+ descProduto : string
+ preco : float
+ quantidade : int
+ ListaraDados() : void
Carrinhos
- itens[] : Produtos
+ ExibeCarrinho() : void
+ FecharCompra() : float

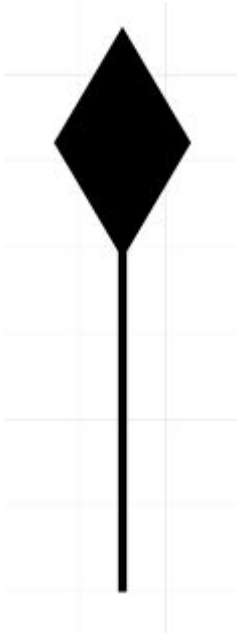
Agregação

Um time é formado por atletas, ou seja, os atletas são parte integrante de um time, **mas os atletas existem independentemente de um time existir**. Nesse caso, chamamos esse relacionamento de AGREGAÇÃO.

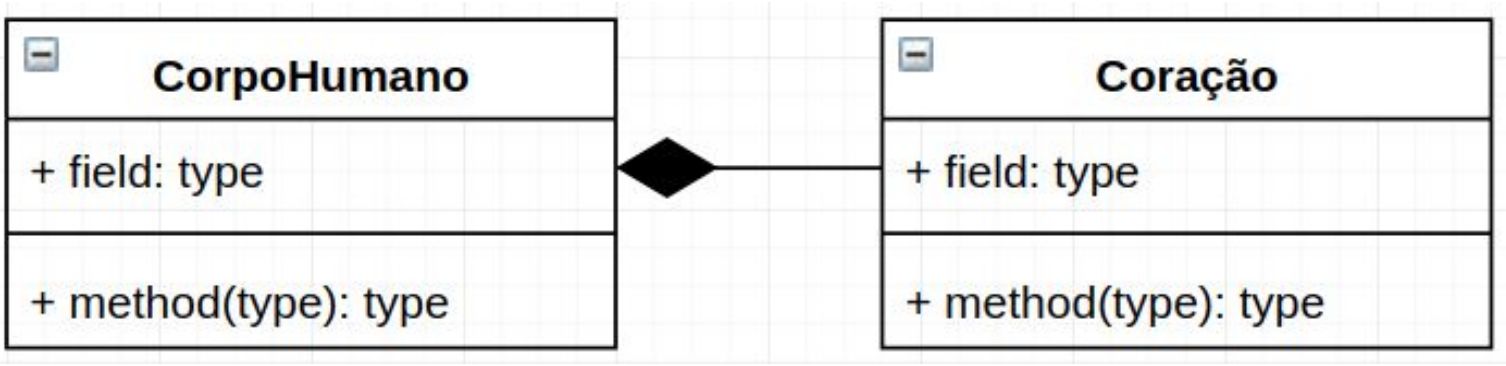
Composição

Composição – conector com um “diamante” hachurado na ponta) – É um tipo de relacionamento onde a **classe composta depende de outras classes para “existir”**. Por exemplo, a classe “CorpoHumano” possui um composição com a classe “Coracao”. Sem a classe “Coracao”, a classe “CorpoHumano” não pode existir.

UML Classe - Composição - Conector



UML Classe - Composição - Exemplo



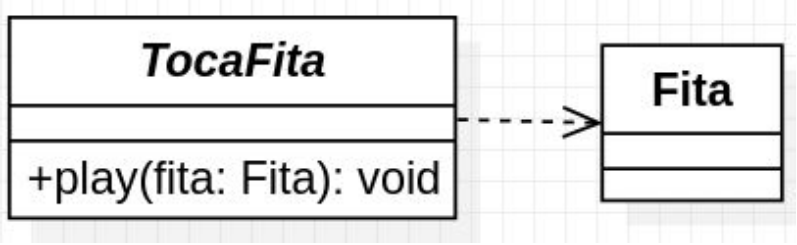
UML Classe - Composição - Em código

```
public class CorpoHumano {
    private Coracao coracao;
}
```

Dependencia

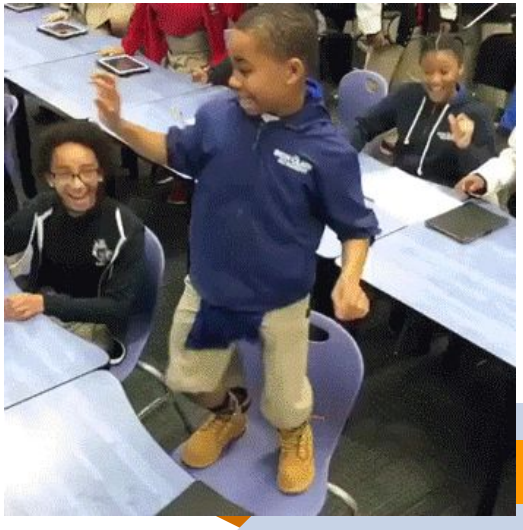
A dependência entre classes indica que os objetos de uma classe **utiliza serviços** de outra classe.

UML Classe - Dependencias - Conector



UML Classe - Dependencias

```
public class TocaFita {
    public void play (Fita fita) {
        ……
    }
}
```



Orientação a Objetos

Interface

"Uma imagem vale mil palavras. Uma interface vale mil imagens." -- Ben Shneiderman

Conceitos de Interface

"Obriga" a um determinado grupo de classes a **ter métodos ou propriedades em comum** para existir em um determinado contexto, contudo os **métodos podem ser implementados** em cada classe de uma **maneira diferente**.

Interface

- Uma interface é semelhante a uma classe, mas **não possui atributos e nem implementação de seus métodos.**
- Não pode ser instanciada, logo não tem construtores.

Interface - implements

Uma classe não estende uma interface, ou seja, uma classe não herda uma interface, mas, **uma classe implementa uma ou diversas interface(s).**

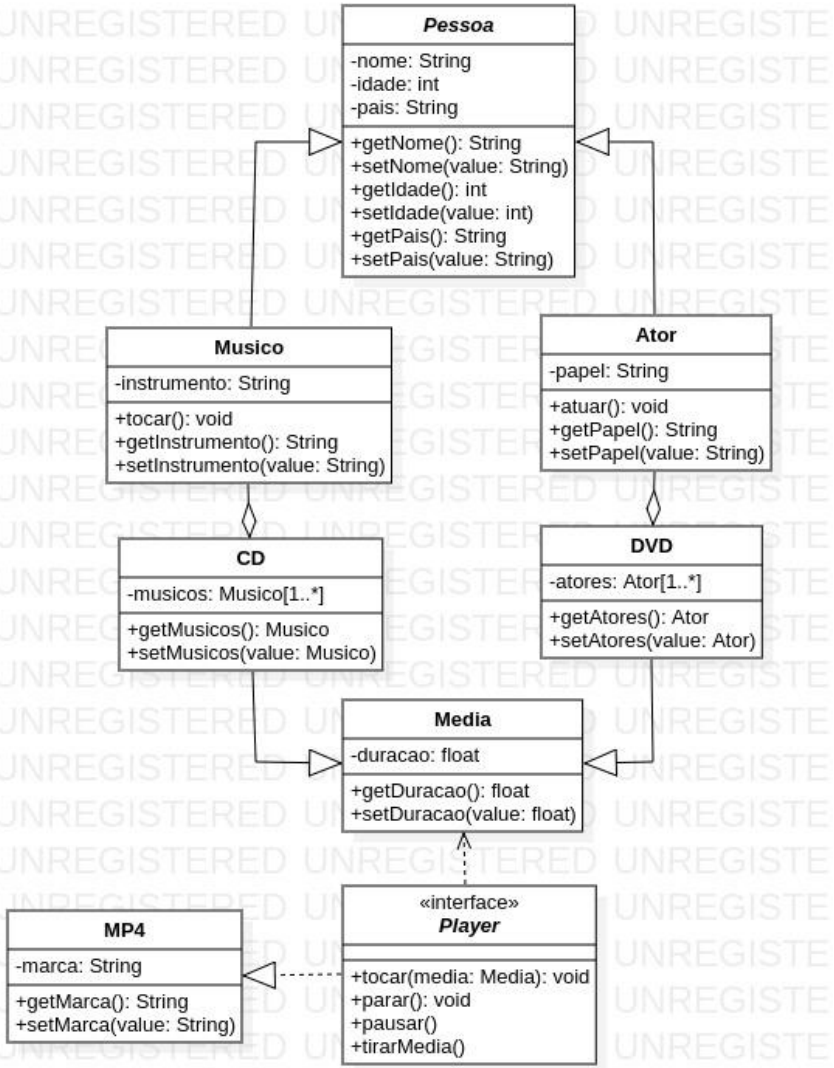
Exemplo de interface

```
public interface ControleRemoto {
    void mudarCanal(int canal);
    void aumentarVolume (int taxa);
    void diminuirVolume (int taxa);
    boolean ligar();
    boolean desligar();
}
```

Exemplo de interface

```
public class TV implements ControleRemoto {
    void mudarCanal(int canal) { }
    void aumentarVolume (int taxa) { }
    void diminuirVolume (int taxa) { }
    boolean ligar() { }
    boolean desligar() { }
}
```

Exercício


Pessoa
-nome: String
-idade: int
-pais: String
+getNome(): String
+setNome(value: String)
+getIdade(): int
+setIdade(value: int)
+getPais(): String
+setPais(value: String)
Musico
-instrumento: String
+tocar(): void
+getInstrumento(): String
+setInstrumento(value: String)
Ator
-papel: String
+atuar(): void
+getPapel(): String
+setPapel(value: String)
CD
-musicos: Musico[1..*]
+getMusicos(): Musico
+setMusicos(value: Musico)
DVD
-atores: Ator[1..*]
+getAtores(): Ator
+setAtores(value: Ator)
Media
-duracao: float
+getDuracao(): float
+setDuracao(value: float)
«interface»
Player
+tocar(media: Media): void
+parar(): void
+pausar()
+tirarMedia()
MP4
-marca: String
+getMarca(): String
+setMarca(value: String)

Dúvidas?

Kaio Mitsuharu Lino Aida
kaiomudkt@gmail.com
Mateus Ragazzi Balbino
mateusragazzi.b@gmail.com
Mário de Araújo Carvalho
mariodearaujocarvalho@gmail.com

http://www.linhadecodigo.com.br/artigo/943/uml-unified-modeling-language-generalizacao-agregacao-composicao-e-dependencia.aspx#ixzz5mtNZVRST

http://www.macoratti.net/net_uml1.htm

https://www.draw.io/

https://imasters.com.br/back-end/uml-composicao-x-agregacao

https://www.devmedia.com.br/interfaces-programacao-orientada-a-objetos/18695