Aula 13: Orientação a Objetos com Java e UML

Professor(a): Virgínia Fernandes Mota
http://www.dcc.ufmg.br/∼virginiaferm

ALGORITMOS E ESTRUTURAS DE DADOS - SETOR DE INFORMÁTICA

- Ao término desta aula, você será capaz de:
  - dizer o que é e para que serve orientação a objetos;
  - conceituar classes, atributos e comportamentos;
  - entender o significado de variáveis e objetos na memória;
  - criar um diagrama UML.

Motivação

- Já vimos: Orientação a objetos é uma maneira de programar que ajuda na organização.
- Orientação a objetos vai ajudar organizar e escrever menos, além de concentrar as responsabilidades nos pontos certos, flexibilizando sua aplicação, encapsulando a lógica de negócios.
- Outra vantagem é o polimorfismo das referências.

Criando um tipo

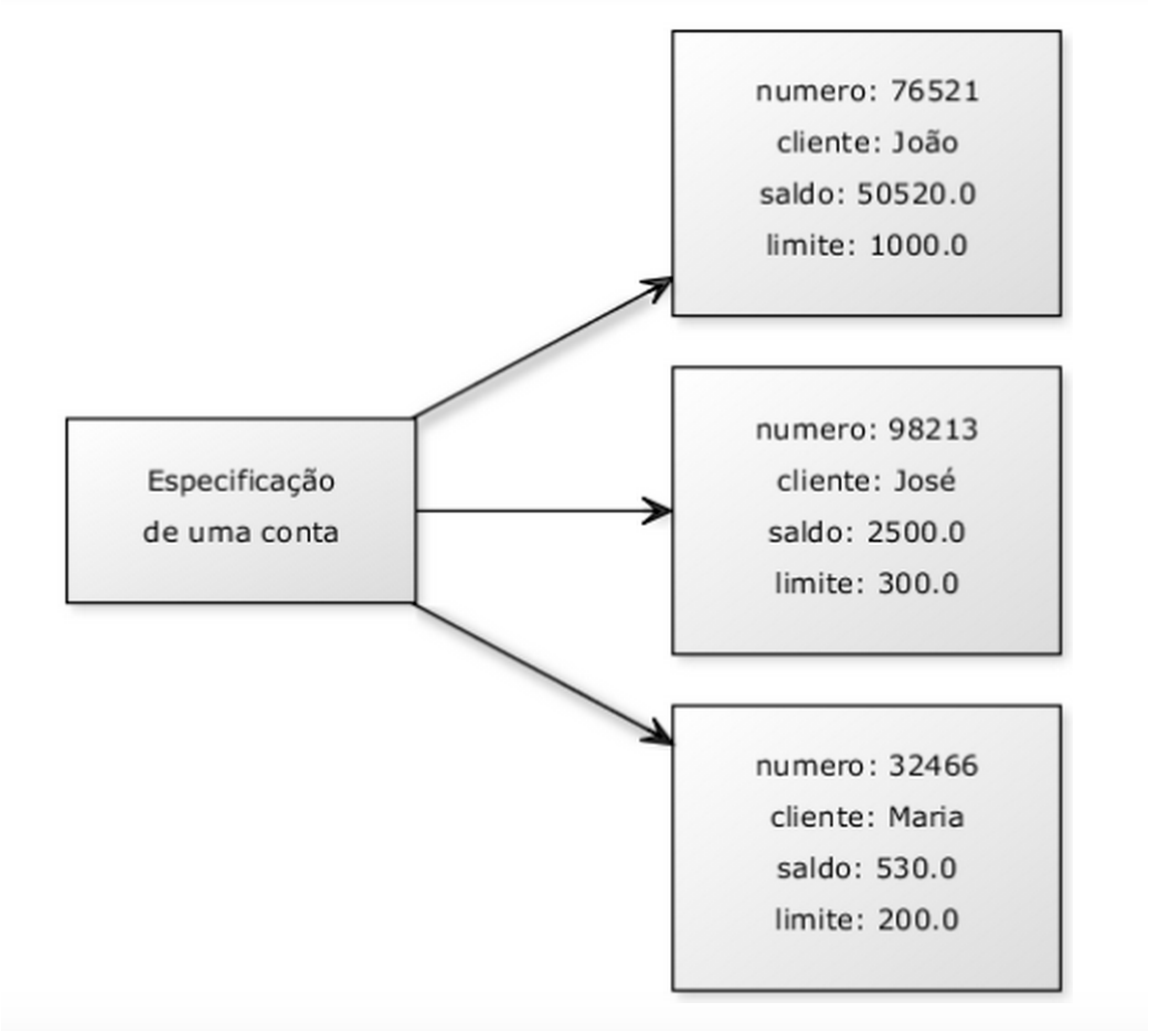
- Considere um programa para um banco, é bem fácil perceber que uma entidade extremamente importante para o nosso sistema é a conta.
- Nossa ideia aqui é generalizarmos alguma informação, juntamente com funcionalidades que toda conta deve ter.
- O que toda conta tem e é importante para nós?
  - número da conta
  - nome do dono da conta
  - saldo
  - limite

Criando um tipo

- O que toda conta faz e é importante para nós? Isto é, o que gostaríamos de "pedir a conta"?
  - saca uma quantidade x
  - deposita uma quantidade x
  - imprime o nome do dono da conta
  - devolve o saldo atual
  - transfere uma quantidade x para uma outra conta y
  - devolve o tipo de conta

Criando um tipo

- Temos o projeto de uma conta bancária.



Criando um tipo

- Ao projeto da conta, isto é, a definição da conta, damos o nome de **classe**.
- Ao que podemos construir a partir desse projeto, as contas de verdade, damos o nome de **objetos**.
- Para utilizar a conta, precisamos **instanciá-la**, criar um objeto conta a partir dessa especificação (a classe) para utilizá-la.
- Podemos criar centenas de contas a partir dessa classe, eles podem ser bem semelhantes, alguns até idênticos, mas são objetos diferentes.

Uma classe em Java

- Vamos começar apenas com o que uma *Conta* tem, e não com o que ela faz.

```
1  class Conta {
2    int numero;
3    String dono;
4    double saldo;
5    double limite;
6
7    // ..
8  }
```

- *String é uma classe em Java.*
- Estes são os **atributos** que toda conta, quando criada, vai ter.

Criando e usando um objeto

- Já temos uma classe. Como usá-la??
- Além dessa classe, ainda teremos o **Programa.java** e a partir dele é que vamos utilizar a classe *Conta*.

```
1  class Programa {
2    public static void main(String[] args) {
3      new Conta();
4    }
5  }
```

- O código acima cria um objeto do tipo *Conta*, mas como acessar esse objeto que foi criado?

Criando e usando um objeto

- Precisamos ter alguma forma de nos referenciarmos a esse objeto. Precisamos de uma variável:

```
1  class Programa {
2    public static void main(String[] args) {
3      Conta minhaConta;
4      minhaConta = new Conta();
5    }
6  }
```

Criando e usando um objeto

- Através da variável minhaConta, podemos acessar o objeto recém criado para alterar seu dono, seu saldo, etc:

```
1  class Programa {
2    public static void main(String[] args) {
3      Conta minhaConta;
4      minhaConta = new Conta();
5
6      minhaConta.dono = "Maria";
7      minhaConta.saldo = 1000.0;
8
9      System.out.println("Saldo atual: " + minhaConta.
           saldo);
10   }
11 }
```

Métodos

- Dentro da classe, também declararemos o que cada conta faz e como isto é feito - os comportamentos que cada classe tem, isto é, o que ela faz.
- Especificaremos isso dentro da própria classe Conta, e não em um local desatrelado das informações da própria Conta.
- É por isso que essas "funções"são chamadas de **métodos** $\rightarrow$ faz uma operação com um objeto.

Métodos

- Exemplo: Queremos criar um método que saca uma determinada quantidade e não devolve nenhuma informação para quem acionar esse método.

```
1  class Conta {
2     double salario;
3     // ... outros atributos ...
4
5     void saca(double quantidade) {
6       double novoSaldo = this.saldo - quantidade;
7       this.saldo = novoSaldo;
8     }
9   }
```

Métodos

- Da mesma forma, temos o método para depositar alguma quantia:

```
1  class Conta {
2     // ... outros atributos e métodos ...
3
4     void deposita(double quantidade) {
5       this.saldo += quantidade;
6     }
7   }
```

- Em nenhum caso tratamos erros... Veremos depois uma forma elegante!

Métodos

- Para mandar uma mensagem ao objeto e pedir que ele execute um método, também usamos o ponto: **invocação de método**.

```
1  class TestaAlgunsMetodos {
2     public static void main(String[] args) {
3       // criando a conta
4       Conta minhaConta;
5       minhaConta = new Conta();
6
7       // alterando os valores de minhaConta
8       minhaConta.dono = "Duke";
9       minhaConta.saldo = 1000;
10
11      // saca 200 reais
12      minhaConta.saca(200);
13
14      // deposita 500 reais
15      minhaConta.deposita(500);
16      System.out.println(minhaConta.saldo);
17    }
18  }
```

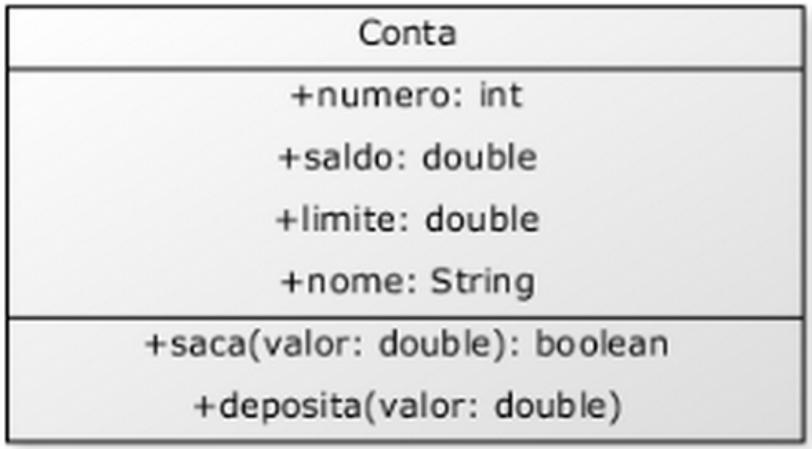
Métodos com retorno

- Um método pode retornar um valor para o código que o chamou.

```
1  class Conta {
2     // ... outros métodos e atributos ...
3
4     boolean saca(double valor) {
5       if (this.saldo < valor) {
6         return false;
7       }
8       else {
9         this.saldo = this.saldo - valor;
10        return true;
11      }
12    }
13  }
```

Métodos com retorno

- Modelo da conta até o momento:



Métodos com retorno

- Meu programa pode manter na memória não apenas uma conta, como mais de uma:

```
1  class TestaDuasContas {
2     public static void main(String[] args) {
3
4        Conta minhaConta;
5        minhaConta = new Conta();
6        minhaConta.saldo = 1000;
7
8        Conta meuSonho;
9        meuSonho = new Conta();
10       meuSonho.saldo = 1500000;
11    }
12 }
```

Objetos são acessados por referências

- Quando declaramos uma variável para associar a um objeto, na verdade, essa variável não guarda o objeto, e sim uma maneira de acessá-lo, chamada de **referência**.
- É por esse motivo que, diferente dos tipos primitivos como int e long, precisamos dar *new* depois de declarada a variável:

```
1  public static void main(String args[]) {
2      Conta c1;
3      c1 = new Conta();
4
5      Conta c2;
6      c2 = new Conta();
7   }
```

- É parecido com um *ponteiro*!

Objetos são acessados por referências

- Um outro exemplo:

```
1  class TestaReferencias {
2     public static void main(String args[]) {
3       Conta c1 = new Conta();
4       c1.deposita(100);
5
6       Conta c2 = c1;  // linha importante!
7       c2.deposita(200);
8
9       System.out.println(c1.saldo);
10       System.out.println(c2.saldo);
11    }
12  }
```

- Qual é o resultado do código acima? O que aparece ao rodar?

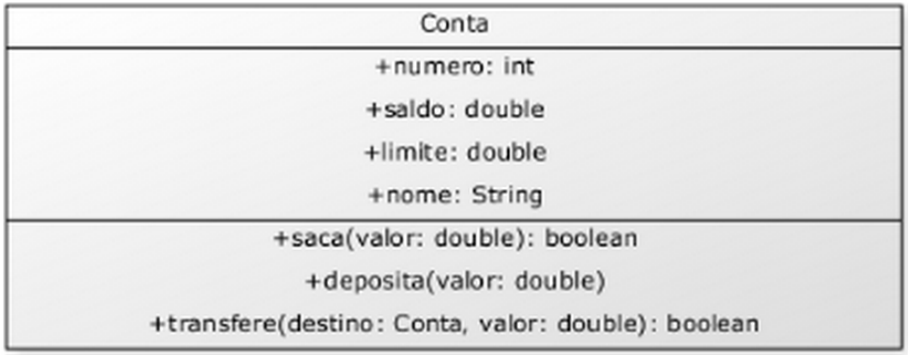
Objetos são acessados por referências

- Comparação: Para saber se dois objetos têm o mesmo conteúdo, você precisa comparar atributo por atributo.
- Veremos uma solução mais elegante para isso...

O método transfere()

- E se quisermos ter um método que transfere dinheiro entre duas contas?
- *Maneira procedural*: criar um método que recebe dois parâmetros: conta1 e conta2 do tipo Conta.
- *Maneira orientada a objetos*: quando chamarmos o método transfere, já teremos um objeto do tipo Conta (o *this*), portanto o método recebe apenas um parâmetro do tipo Conta, a Conta destino (além do valor).

O método transfere()




```
1  class Conta {
2
3    // atributos e métodos...
4
5    boolean transfere(Conta destino, double valor) {
6      boolean retirou = this.saca(valor);
7      if (retirou == false) {
8        // não deu pra sacar!
9        return false;
10     }
11     else {
12       destino.deposita(valor);
13       return true;
14     }
15   }
16 }
```

O método transfere()

- Quando passamos uma Conta como argumento, o que será que acontece na memória? Será que o objeto é *clonado*?
- A passagem de parâmetro funciona como uma simples atribuição como no uso do "=": copia valor da variável.
- Valor: um endereço, uma referência, nunca um objeto. Por isso não há cópia de objetos aqui.

Continuando com atributos

- As variáveis do tipo atributo, diferentemente das variáveis temporárias (declaradas dentro de um método), recebem um valor padrão.
- No caso numérico, valem 0, no caso de boolean, valem false. E para valores default?

```
1  class Conta {
2     int numero = 42;
3     String dono = "Dent";
4     String cpf = "123.456.789-10";
5     double saldo = 1000;
6     double limite = 1000;
7   }
```

Continuando com atributos

- Seus atributos também podem ser referências para outras classes.

```java
class Cliente {
   String nome;
   String sobrenome;
   String cpf;
 }

 class Conta {
   int numero;
   double saldo;
   double limite;
   Cliente titular;
   // ..
 }
```

```java
class Teste {
  public static void main(
      String[] args) {
    Conta minhaConta = new
        Conta();
    Cliente c = new Cliente()
        ;
    minhaConta.titular = c;
    minhaConta.titular.nome =
        "M";
    // ...
  }
}
```

Continuando com atributos

- Mas, e se dentro do meu código eu não desse new em Cliente e tentasse acessá-lo diretamente?

```
1  class Teste {
2    public static void main(String[] args) {
3      Conta minhaConta = new Conta();
4
5      minhaConta.titular.nome = "Manoel";
6      // ...
7    }
8  }
```

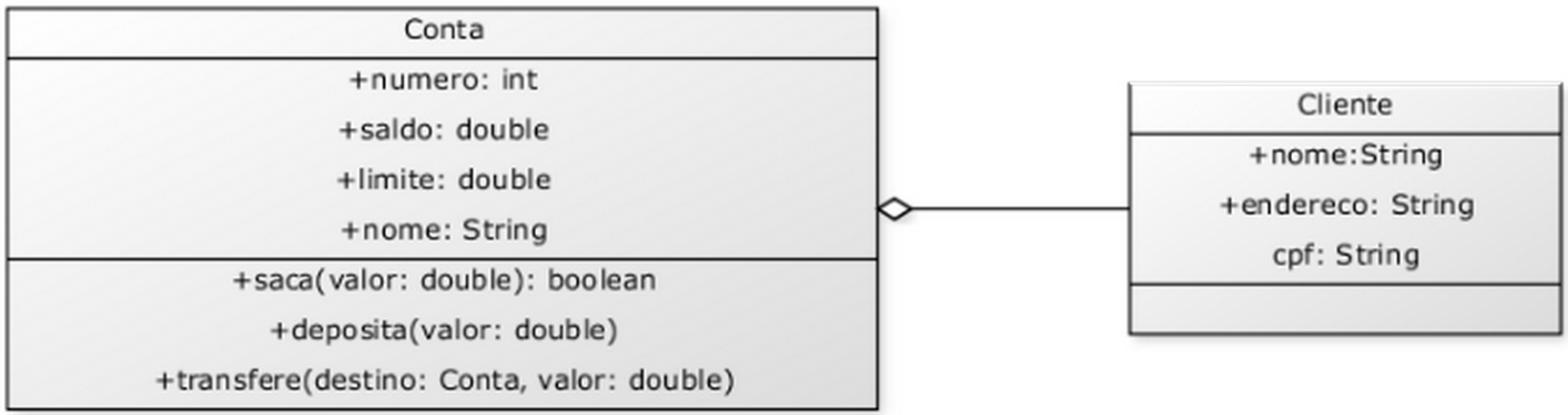
Continuando com atributos

- Quando damos new em um objeto, ele o inicializa com seus valores default, 0 para números, false para boolean e null para referências.
- null é uma palavra chave em java, que indica uma referência para nenhum objeto.
- Se, em algum caso, você tentar acessar um atributo ou método de alguém que está se referenciando para null, você receberá um erro durante a execução: *NullPointerException*.

Continuando com atributos

- Uma ideia:

```
1  class Conta {
2     int numero;
3     double saldo;
4     double limite;
5     Cliente titular = new Cliente();  // quando chamarem
          new Conta,
6                         //havera um new Cliente para ele.
7   }
```



Exemplo: Uma Fábrica de Carros

```java
class Carro {
    String cor;
    String modelo;
    double velocidadeAtual;
    double velocidadeMaxima;

    //liga o carro
    void liga() {
      System.out.println("O
          carro está ligado");
    }

    //acelera uma certa
        quantidade
    void acelera(double
        quantidade) {
      double velocidadeNova =
          this.velocidadeAtual
          + quantidade;
      this.velocidadeAtual =
          velocidadeNova;
    }
```

```java
//devolve a marcha do carro
    int pegaMarcha() {
      if (this.velocidadeAtual
          < 0) {
        return -1;
      }
      if (this.velocidadeAtual
          >= 0 && this.
          velocidadeAtual < 40)
           {
        return 1;
      }
      if (this.velocidadeAtual
          >= 40 && this.
          velocidadeAtual < 80)
           {
        return 2;
      }
      return 3;
    }
 }
```

Exemplo: Uma Fábrica de Carros

```
1  class TestaCarro {
2     public static void main(String[] args) {
3        Carro meuCarro;
4        meuCarro = new Carro();
5        meuCarro.cor = "Verde";
6        meuCarro.modelo = "Fusca";
7        meuCarro.velocidadeAtual = 0;
8        meuCarro.velocidadeMaxima = 80;
9
10       // liga o carro
11       meuCarro.liga();
12
13       // acelera o carro
14       meuCarro.acelera(20);
15       System.out.println(meuCarro.velocidadeAtual);
16    }
17 }
```

Exemplo: Uma Fábrica de Carros

- Nosso carro pode conter também um Motor:

```
1  class Motor {
2     int potencia;
3     String tipo;
4   }
5
6   class Carro {
7     String cor;
8     String modelo;
9     double velocidadeAtual;
10    double velocidadeMaxima;
11    Motor motor;
12
13    // ..
14  }
```

- Podemos criar diversos Carros e mexer com seus atributos e métodos, assim como fizemos no exemplo do Banco.

Modelando sistemas - UML

- Agora que estamos vendo como a orientação a objetos funciona em Java, vamos pensar em uma maneira de organizar nossas idéias ao criar um sistema.
- A linguagem UML procura fornecer meios para auxiliar no levantamento dos requisitos que irão constituir um sistema, além de recursos para a modelagem de estruturas que farão parte do mesmo.
- UML = Unified Modeling Language

Modelando sistemas - UML

- UML é uma linguagem para modelagem de estruturas que irão compor uma aplicação, estando fortemente amparada em conceitos de Orientação a Objetos.
- Em termos práticos, a UML contempla uma série de notações para a construção de diagramas representando diferentes aspectos de um software, além de não estar presa a metodologias ou tecnologias específicas de desenvolvimento.

- As diversas notações da UML podem ser utilizadas em várias situações:
  - Para esboçar estruturas de um sistema em discussões a respeito do mesmo. Isto costuma acontecer de um modo informal, através do desenho de um componente ou processo da aplicação considerada, buscando assim um melhor entendimento daquilo que está analisando;
  - Como documentação que servirá de base para atividades de codificação das estruturas de um sistema, bem como elaboração de testes das funcionalidades implementadas;
  - Na documentação de estruturas já existentes de um sistema, ou seja, como uma ferramenta de engenharia reversa, a partir da qual serão documentadas funcionalidades e outras estruturas da aplicação em questão.

- Os diferentes diagramas que compõem a UML podem ser agrupados em categorias, levando em conta para isto o contexto em que cada uma dessas representações pode vir a ser empregada:
  - Diagramas Estruturais;
  - Diagramas Comportamentais;
  - Diagramas de Interação.

UML: Diagramas Estruturais

- **Diagrama de classes**
- Diagrama de componentes
- Diagrama de pacotes
- Diagrama de objetos
- Diagrama de estrutura composta
- Diagrama de instalação
- Diagrama de perfil

UML: Diagramas Comportamentais

- Diagrama de casos de uso
- Diagrama de atividades
- Diagrama de Transição de estados

UML: Diagramas de Interação

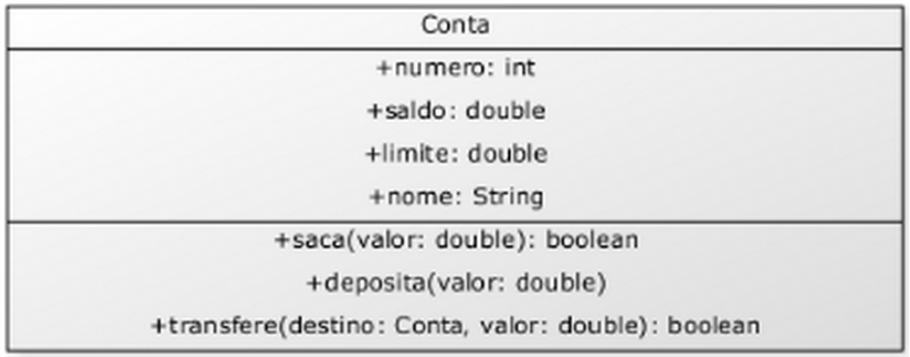
- Diagrama de sequência
- Diagrama de interatividade
- Diagrama de colaboração ou comunicação
- Diagrama de tempo

UML: Diagramas de Classes

- Mostra um conjunto de classes e seus relacionamentos.
- É o diagrama central da modelagem orientada a objetos.
- Elementos:
  - Classes
  - Relacionamentos

UML: Diagramas de Classes

- Classes: graficamente, as classes são representadas por retângulos incluindo nome, atributos e métodos.
- É comum adotar um padrão para nomeá-las: todos os nomes de classes serão substantivos singulares com a primeira letra maiúscula.



UML: Diagramas de Classes

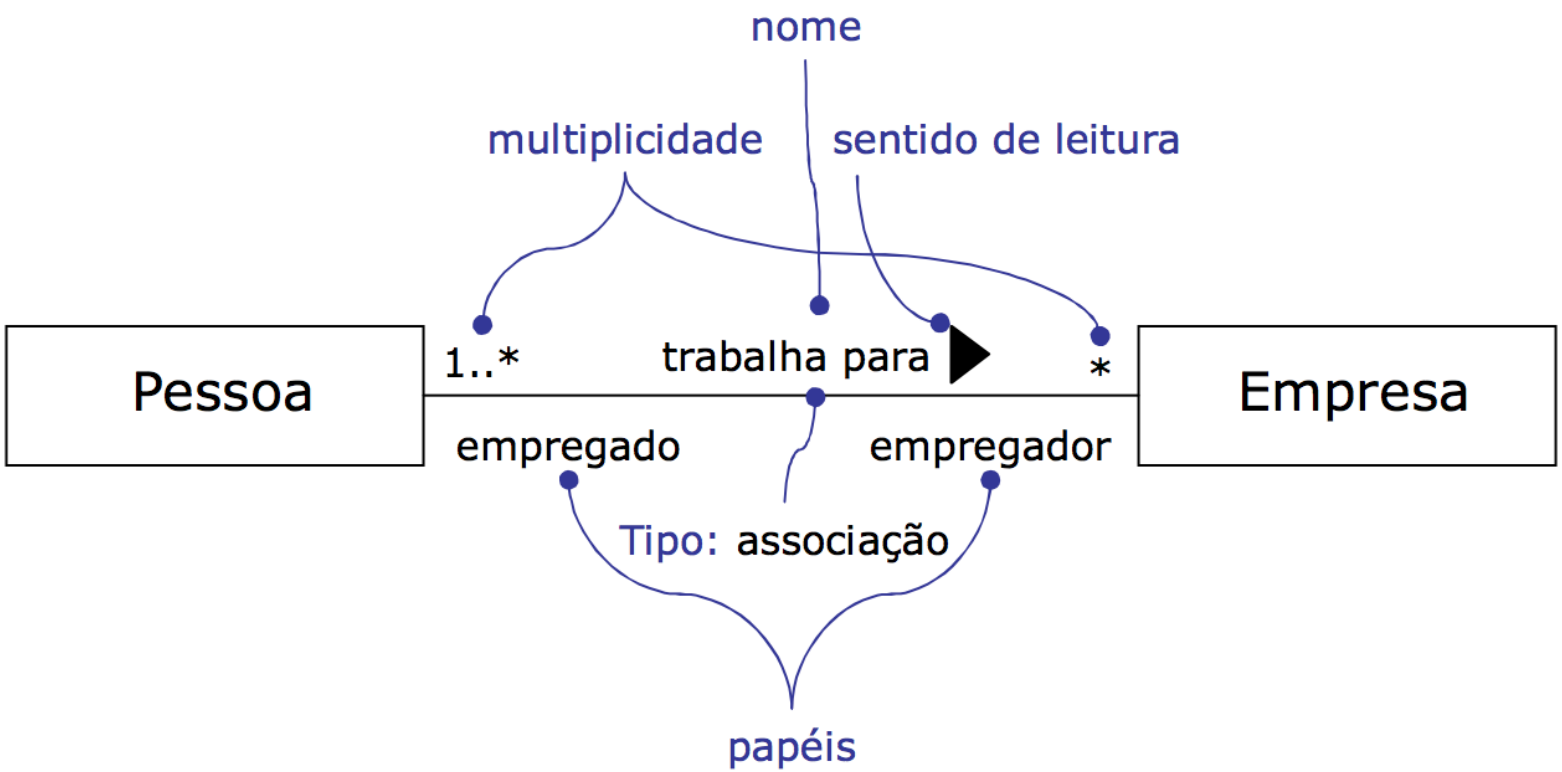
- Atributos: Representam o conjunto de características (estado) dos objetos daquela classe
- Visibilidade
  - + público: visível em qualquer classe de qualquer pacote
  - # protegido: visível para classes do mesmo pacote
  - - privado: visível somente para classe

- Métodos: Representam o conjunto de operações (comportamento) que a classe fornece
- Visibilidade
  - + público: visível em qualquer classe de qualquer pacote
  - # protegido: visível para classes do mesmo pacote
  - - privado: visível somente para classe

UML: Diagrama de Classes

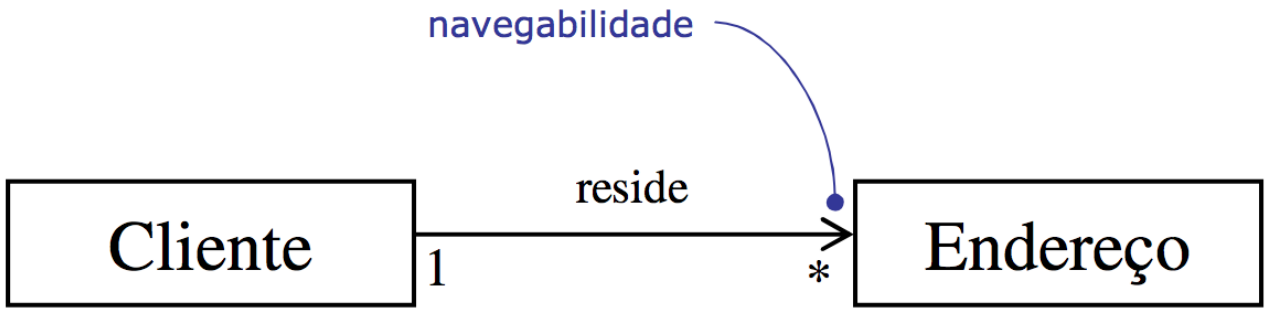
- Relacionamentos:
  - Associação: Agregação; Composição.
  - Generalização
  - Dependência
- Possuem: nome, sentido de leitura, navegabilidade, multiplicidade, tipo, papéis.

UML: Diagrama de Classes



E a navegabilidade?

UML: Diagrama de Classes

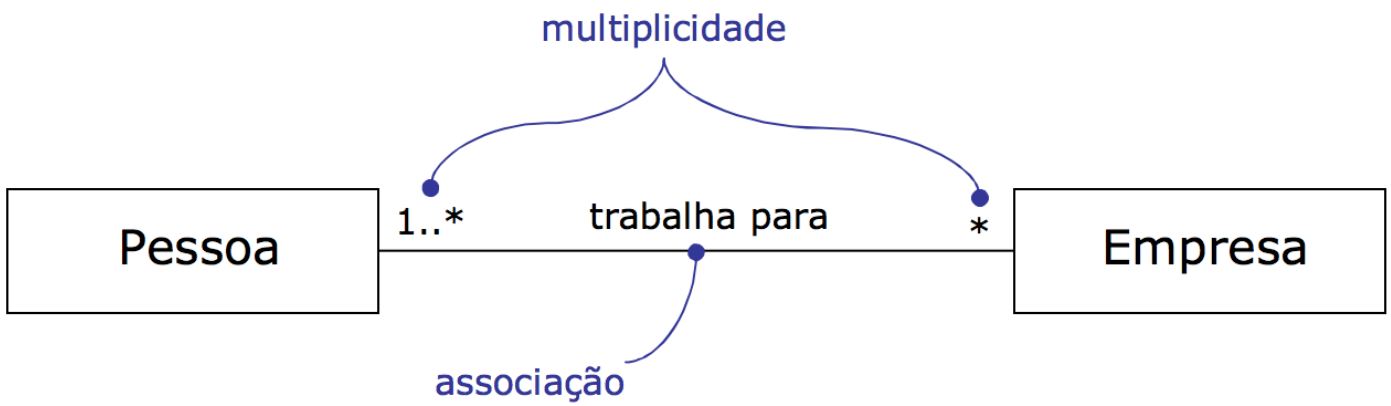


- O cliente sabe quais são seus endereços, mas o endereço não sabe a quais clientes pertence

UML: Diagrama de Classes

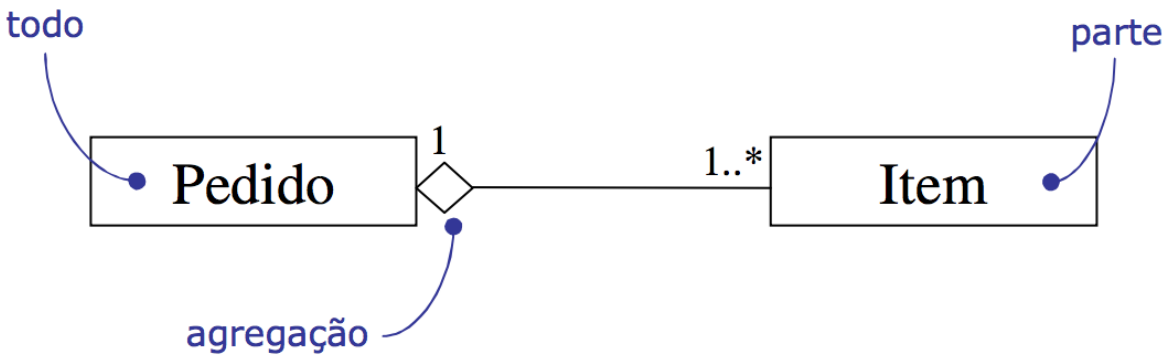
- Relacionamentos: Associação.
  - Uma **associação** é um relacionamento estrutural que indica que os objetos de uma classe estão vinculados a objetos de outra classe.
  - Uma associação é representada por uma linha sólida conectando duas classes.
  - Multiplicidade:
    - 1 Exatamente um
    - 1..* Um ou mais
    - 0..* Zero ou mais (muitos)
    - * Zero ou mais (muitos)
    - 0..1 Zero ou um
    - m..n Faixa de valores

UML: Diagrama de Classes

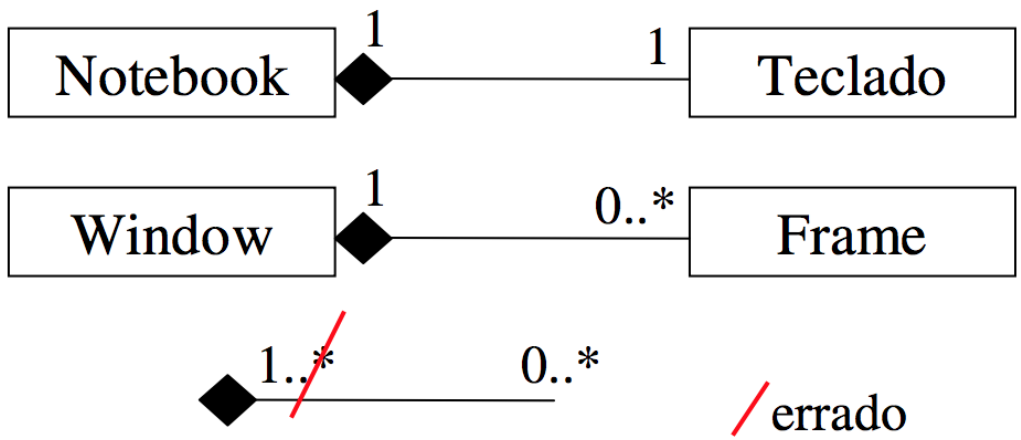


UML: Diagrama de Classes

- Relacionamentos: Agregação.
  - É um tipo especial de associação
  - Utilizada para indicar "todo-parte"

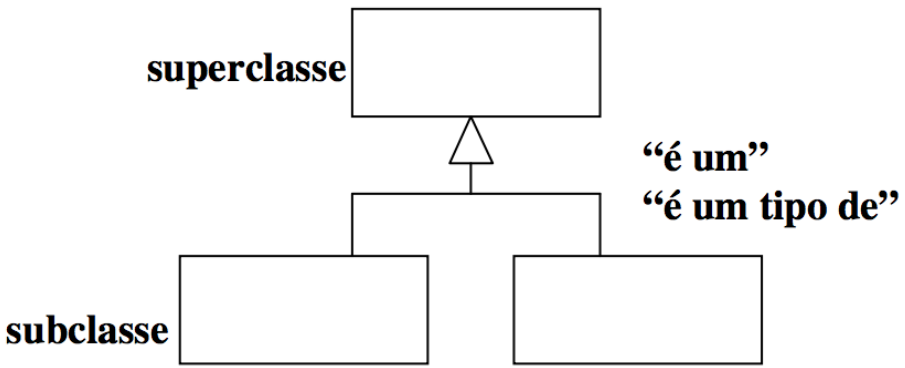


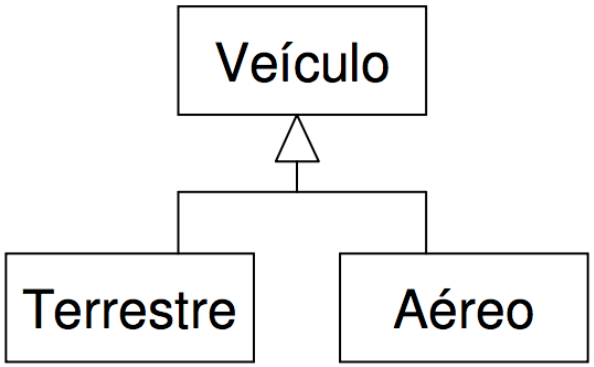
- Relacionamentos: Composição.
  - É uma variante semanticamente mais "forte" da agregação
  - Os objetos "parte" só podem pertencer a um único objeto "todo" e têm o seu tempo de vida coincidente com o dele



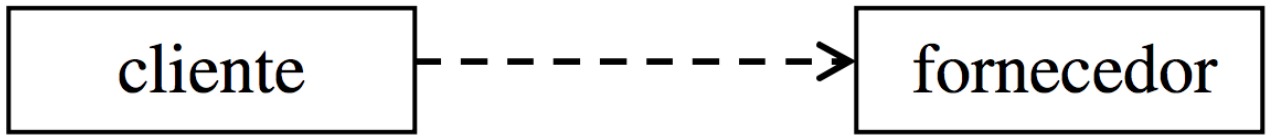
UML: Diagrama de Classes

- Relacionamentos: Generalização.
  - É um relacionamento entre itens gerais (superclasses) e itens mais específicos (subclasses): HERANÇA!



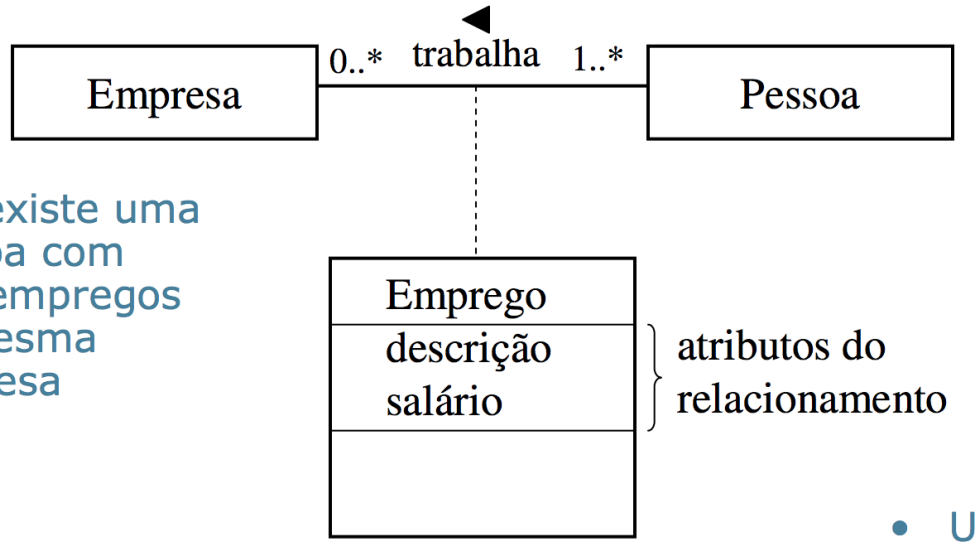


- Relacionamentos: Dependência.
  - Representa que a alteração de um objeto (o objeto indepedendente) pode afetar outro objeto (o objeto dependente)



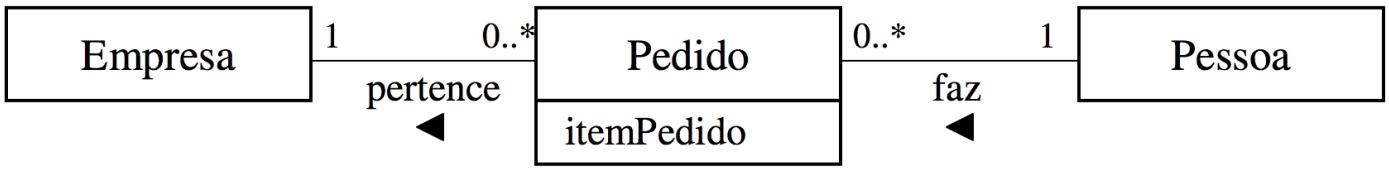
UML: Diagrama de Classes

- Classe de associação: Usada quando uma associação entre duas classes contiver atributos da associação



- Não existe uma pessoa com dois empregos na mesma empresa

- Uma pessoa pode fazer mais de um pedido na mesma empresa



Exercícios: Modelo de funcionários

O objetivo aqui é criar um sistema para gerenciar os funcionários do Banco!

- Modele um funcionário usando UML. Ele deve ter o nome do funcionário, o departamento onde trabalha, seu salário (double), a data de entrada no banco (String) e seu RG (String).
- Transforme o modelo acima em uma classe Java. Teste-a, usando uma outra classe que tenha o main. Você deve criar a classe do funcionário com o nome Funcionario, mas pode nomear como quiser a classe de testes, contudo, ela deve possuir o método main.
  - javac *.java

Exercícios: Modelo de funcionários

- Crie um método mostra(), que não recebe nem devolve parâmetro algum e simplesmente imprime todos os atributos do nosso funcionário. Dessa maneira, você não precisa ficar copiando e colando um monte de System.out.println() para cada mudança e teste que fizer com cada um de seus funcionários.
- Construa dois funcionários com o new e compare-os com o ==. E se eles tiverem os mesmos atributos? Para isso você vai precisar criar outra referência.

Exercícios: Modelo de funcionários

- Crie duas referências para o mesmo funcionário, compare-os com o ==. Tire suas conclusões. Para criar duas referências pro mesmo funcionário.
- Em vez de utilizar uma String para representar a data, crie uma outra classe, chamada Data. Ela possui 3 campos int, para dia, mês e ano. Faça com que seu funcionário passe a usá-la. (é parecido com o último exemplo, em que a Conta passou a ter referência para um Cliente).
- Modifique seu método mostra para que ele imprima o valor da dataDeEntrada daquele Funcionario.

Na próxima aula...

Arrays