



Curso de Extensão  
**Tecnologias Microsoft**



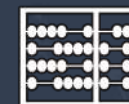
INF-0099

# Introdução a Orientação a Objetos

Profa. Dra. Esther Luna Colombini

[esther@ic.unicamp.br](mailto:esther@ic.unicamp.br)

9 de Maio de 2023



INSTITUTO DE  
COMPUTAÇÃO



- Introdução aos paradigmas de programação
- Abstração de Dados, Objetos, Classes e Tipos
- Propriedades e estados
- Modularização e Visibilidade
- Métodos e Mensagens. Sobrecarga de Métodos
- Hierarquias de generalização/especialização
- Herança Simples e Múltipla
- Relacionamentos: associação, agregação, composição
- Sobrescrita, Polimorfismo e Alocação Dinâmica
- Classes abstratas

The background of the slide features a network of interconnected nodes and lines. There are several large circles in orange, yellow, light blue, and light green, each with a grey border. These are connected by thin grey lines. Additionally, there are several straight lines in orange, light blue, and light green that do not connect to any nodes. The central text is contained within a dark blue horizontal band with yellow borders at the top and bottom.

# O que é POO?

# Programação Orientada a Objetos



- Objetivos:
  - Entender
    - o que é POO
    - para que serve
    - porque OO



# Programação Orientada a Objetos



- Paradigmas de Programação
- Histórico
- Orientação a Objetos



# Paradigmas de Programação



- Orientado a procedimento (Procedural)
  - procedimentos, sequência
- Orientado a lógica
  - regras, padrões e inferências
- Funcional
  - funções matemáticas
- Orientado a objeto
  - abstração, ligação dinâmica, herança



# Orientado a procedimento



- Ênfase maior dada aos procedimentos e funções
  - Modela-se a solução de problemas com base nas funções a serem executadas
  - Dados tratados de forma secundária
  - Paradoxo
    - Dados são mais importantes
    - Sem os dados os procedimentos não teriam utilidade prática
  - Exemplos
    - Pascal, ALGOL, C



# Orientado a procedimento



```
typedef unsigned long NumConta;
typedef int bool;

bool fazDeposito(NumConta conta, float valor);
float fazRetirada(NumConta conta, float valor);
bool transfere(NumConta origem, NumConta destino, float valor);
void imprimeConta(NumConta conta);

struct Conta
{
    char *titular;
    NumConta contaId;
    float saldo;
    char tipo;
};|
```







- Programação Lógica
  - Programação de forma declarative, ou seja, especificando o que deve ser computado ao invés de como deve ser computado
  - Relações são mais genéricas do que mapeamentos, portanto programação lógica é mais alto nível que imperativa ou funcional
  - Sem instruções explícitas e sequenciamento
  - Exemplos
    - PROLOG





```
predecessor(X,Z) :- parent(X,Z).  
predecessor(X,Z) :-  
    parent(X,Y),predecessor(Y,Z).  
sister(X,Y) :- female(X), parent(Z,X), parent(Z,Y), not(X=Y).  
grandparent(X,Z) :- parent(X,Y),parent(Y,Z).|
```





- Uso de expressões e funções no lugar de variáveis, comandos e procedimentos
  - enfatiza a avaliação de expressões
  - não utiliza comandos e algoritmos
  - não utiliza variáveis e atribuições
  - exige bastante disciplina de programação
  - produz programas que podem ser mais facilmente verificados
  - Exemplos
    - Lisp, ML, Haskell



# Paradigma Funcional



```
fun potencia (x:real, n:int) =  
  if n = 1 then  
    x  
  else  
    x * potencia(x, n-1);
```



# Orientado a Objetos



- Programação em POO
  - Ênfase à estrutura de dados, adicionando funcionalidade a elas
- Inicia-se decidindo os objetos necessários, que são, então, caracterizados através de propriedades e comportamento
- O programador vê seu programa como uma coleção de objetos cooperantes comunicando-se através de mensagens
- Exemplos
  - C++, Java, C#



# Orientado a Objetos



```
class Conta
{
    public:
        bool fazDeposito(float valor);
        float fazRetirada(float valor);
        bool transfere(Conta &destino, float valor);
        void imprimeConta() const;
    private:
        char titular[255];
        int numeroConta;
        float saldo;
};
```



# Paradigmas: Comparativo



```
typedef unsigned long NumConta;
typedef int bool;

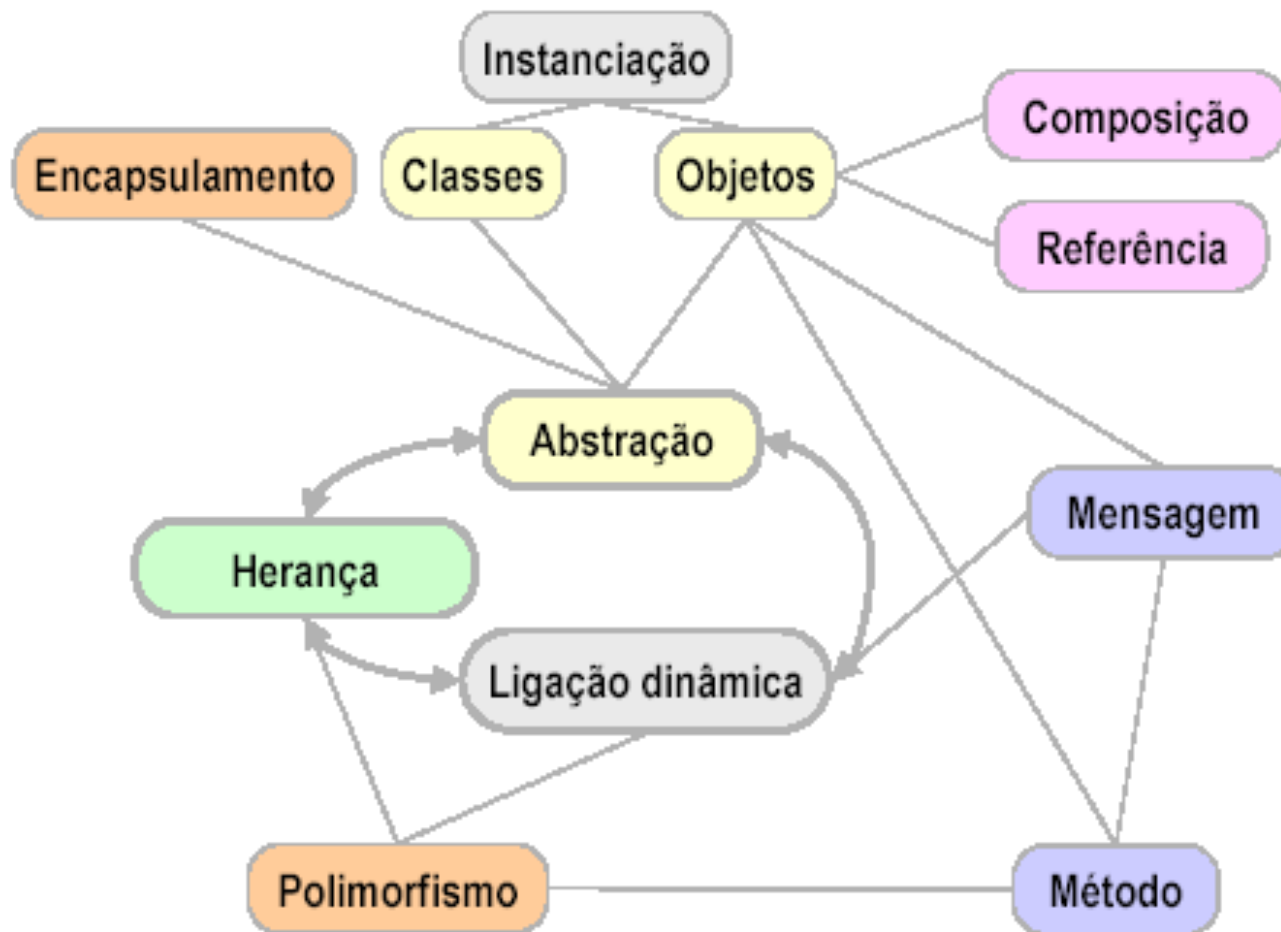
bool fazDeposito(NumConta conta, float valor);
float fazRetirada(NumConta conta, float valor);
bool transfere(NumConta origem, NumConta destino, float valor);
void imprimeConta(NumConta conta);

struct Conta
{
    char *titular;
    NumConta contaId;
    float saldo;
    char tipo;
};
```

```
class Conta
{
public:
    bool fazDeposito(float valor);
    float fazRetirada(float valor);
    bool transfere(Conta &destino, float valor);
    void imprimeConta() const;
private:
    char titular[255];
    int numeroConta;
    float saldo;
};
```



# Orientação a Objetos







- Orientação a Objetos não é um conceito novo
- O conceito de objetos e classes foi primeiro utilizado na linguagem Simula 67 (extensão de ALGOL 60)
- Na década de 70, Barbara Liskov trabalhou com TADs e desenvolveu a linguagem CLU
- Durante a década de 70 pesquisadores do Centro de Pesquisa Xerox Palo Alto desenvolveram a linguagem SmallTalk
  - Linguagem mais representativa (pura)



- Por volta de 1983, Bjarne Stroustrup incorpora classes a C, criando assim a linguagem C++
  - C++ sofreu várias melhorias nas décadas de 80 e 90. Foi iniciado o processo de padronização em 1991 pela ANSI/ISO
- Na década de 80 surgiram inúmeras metodologias de desenvolvimento
- Em meados da década de 90 (1995) surge a linguagem Java, criada por James Gosling
- Em 1997 Booch, Rumbaugh e Jacobson unificam suas metodologias e criam a UML
- Em 1999, a convite da Microsoft, Anders Hejlsberg formou uma equipe de programadores para desenvolver uma nova linguagem de programação
- Mais tarde, em 2003, tornou-se padrão também da ISO, recebendo a especificação de ISO/IEC 23270.



# Histórico de POO

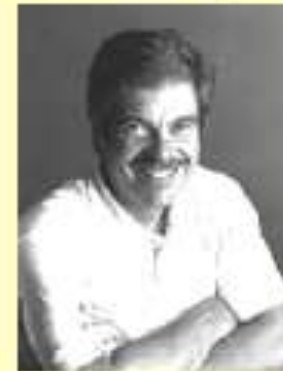


**C++**



Bjarne Stroustrup

**Smalltalk**



Alan Kay

**SIMULA**



Ole-Johan Dahl (1970)



Kristen Nygaard (1970)

**JAVA**



James Gosling



- Objetos
  - Seres humanos classificam o mundo em objetos



Identidade: ferrari	
Estado	Comportamento
Marca	Acelerar
Cor	Ligar
Motor	Verificar Combustível
Chassi	Calibrar Pneus



## Identidade: cachorro

Estado	Comportamento
Nome	Latir
Raça	Comer
Cor	Brincar
Pedigree	Deitar
Altura	Rolar
Peso	



## Identidade: bicicleta

Estado	Comportamento
Marca	Trocar Marcha
Modelo	Pedalar
Marchas	Freiar
Cor	





## Objetos

Entidade autônoma que une a representação da informação (estruturas de dados) com a sua manipulação (procedimentos)

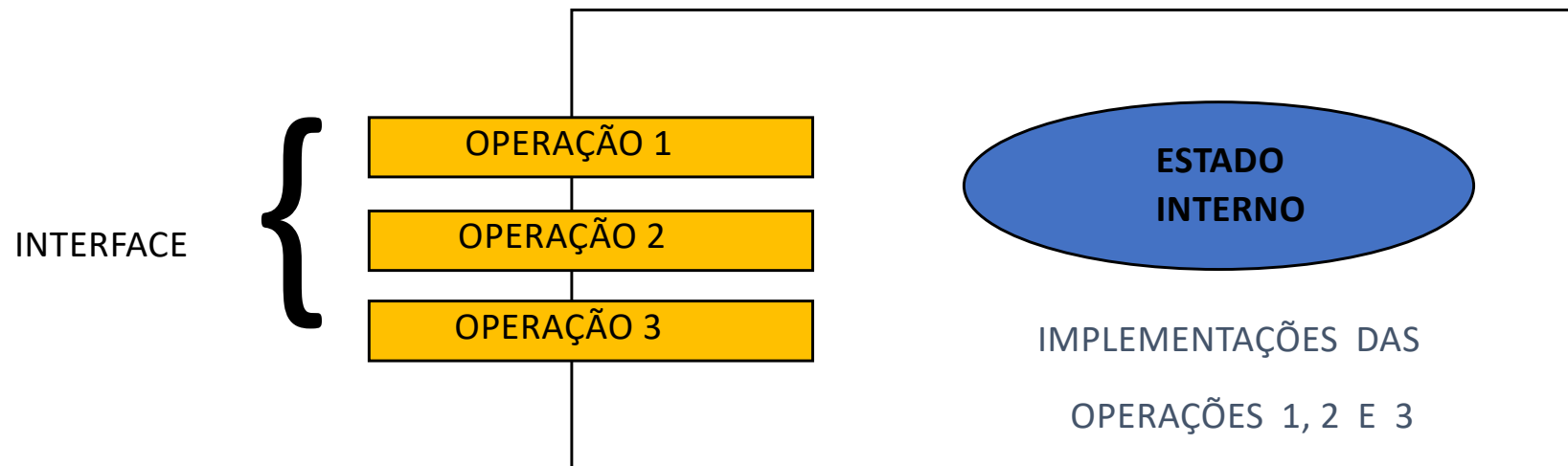
Informalmente um objeto representa uma entidade física, conceitual ou de software

Possui capacidade de processamento e armazena um estado local

Conceito mais próximo na programação convencional

- variável

# Estrutura de um Objeto



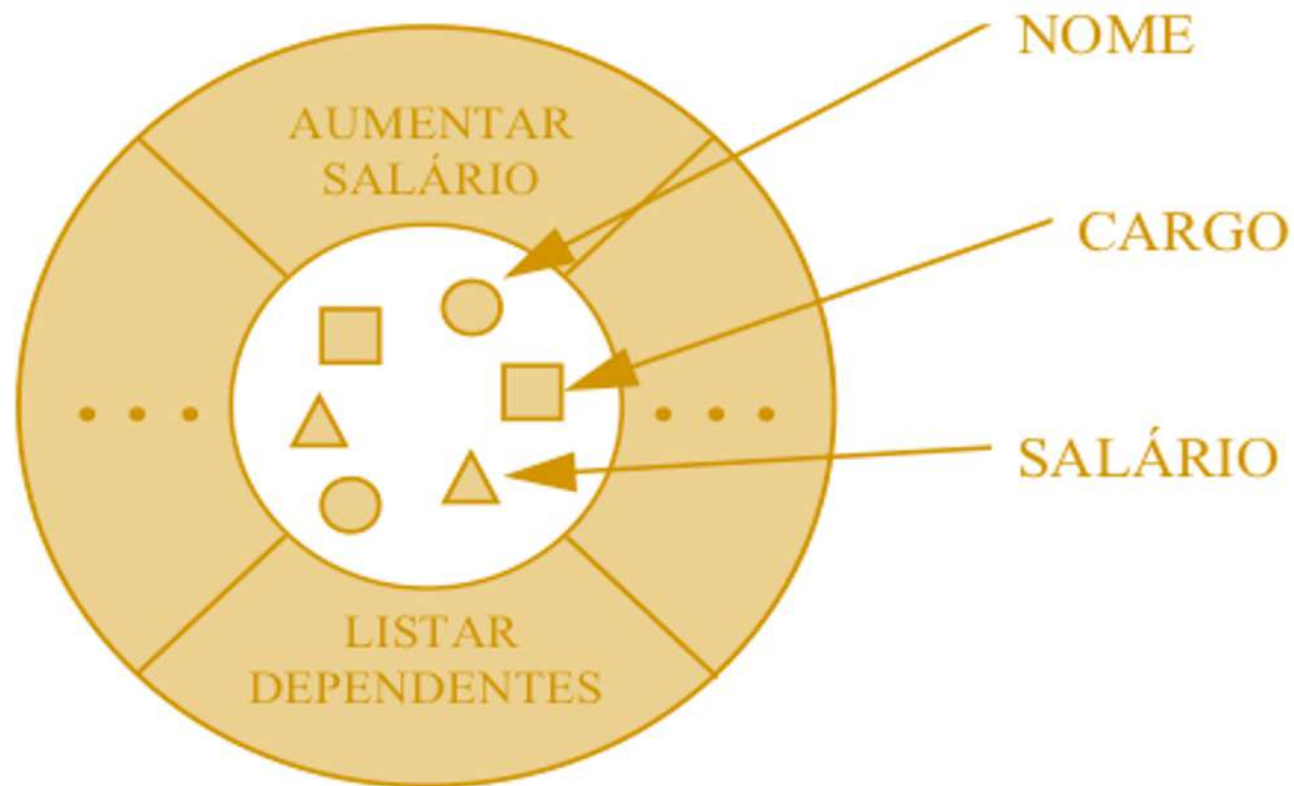
- Em OO os conceitos de dados e procedimentos são reunidos em uma única entidade: o objeto





- Um objeto é composto de:
  - **Propriedades:** informações (estruturas de dados) que representam o estado interno
  - **Comportamento:** conjunto de operações (métodos), que agem sobre as suas propriedades
  - **Identidade:** é uma propriedade que permite diferenciar um objeto de outro, independentemente de sua classe ou estado atual

# Estrutura de um Objeto



# Aspectos importantes da OO



- Programação Orientada a Objetos: é um método de implementação no qual programas são organizados como uma coleção cooperativa de objetos, cada um deles representando uma instância de alguma classe, as quais pertencem a uma hierarquia de classes unidas através de uma relação de herança.
- Object-Oriented Analysis and Design
- G. Booch - 2nd edition - Addison-Wesley



- **Análise Orientada a Objetos:** é um método de análise que examina os requisitos através de perspectivas de classes e objetos encontrados no vocabulário do domínio do problema.
- **Projeto Orientado a Objetos:** é um método de projeto que engloba o processo de decomposição orientada a objetos e uma notação para representar as características lógicas e físicas assim como os aspectos estáticos e dinâmicos dos sistemas.
- **Um Modelo Orientado a Objetos** deve oferecer os seguintes elementos: Abstração, Encapsulamento, Modularidade e Hierarquia, podendo também oferecer as seguintes características: Tipagem, Concorrência e Persistência.

# Etapas importantes em OO

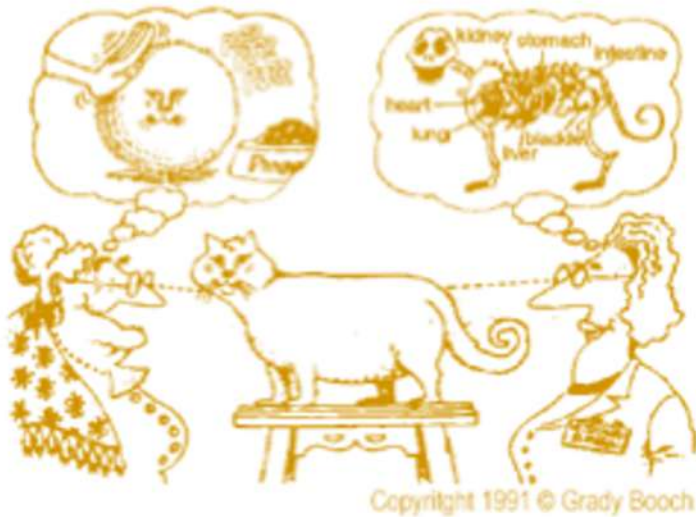


- Etapas fundamentais do design orientado a objetos de um software:
  - Analisar os requisitos que descrevem o sistema desejado.
  - Determinar os objetos necessários para implementar o sistema.
  - Determinar os atributos que os objetos terão.
  - Determinar os comportamentos que esses objetos exibirão.
  - Especificar como ocorre a interação entre os objetos para atender aos requisitos do sistema.

# Benefícios da OO



- Abstração
- Modularidade
- Encapsulamento
- Reusabilidade
- Escalabilidade



- Técnica para lidar com a complexidade de um problema
- Destaca os aspectos importantes do objeto real abstraído segundo o observador
- Ignora detalhes não relevantes para o observador
  - Depende da perspectiva do observador

## Exemplos de Abstração

Um mapa mundi

Um modelo de avião

Uma maquete de edifício

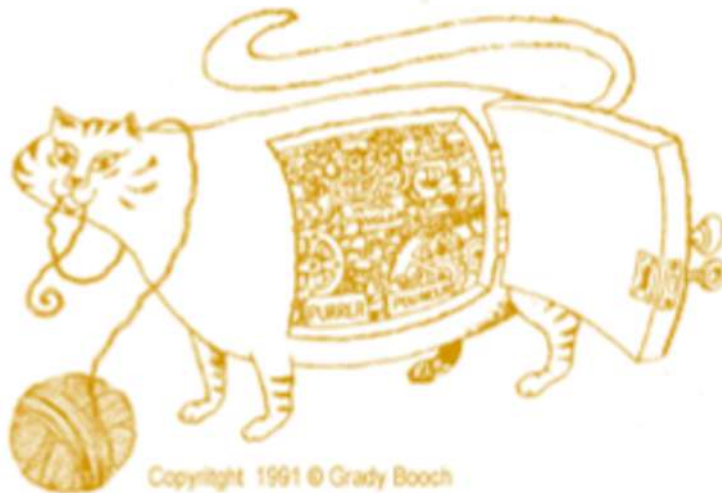
Um tipo de dado abstrato

A planta baixa de uma casa



- Construção de programas a partir da junção de partes menores (módulos), independentes
- É a propriedade que um sistema tem de poder ser decomposto em um conjunto de módulos coesos e fortemente ligados, facilitando sua compreensão.
- Divide um sistema complexo em módulos menores e melhor gerenciáveis individualmente.





- Abstração e encapsulamento são conceitos complementares: a abstração representa comportamento observável do objeto e o encapsulamento, a implementação deste comportamento.
- O encapsulamento é também chamado de ocultamento da informação
  - os segredos de um objeto que não contribuem para a definição de suas características essenciais (seus dados e operações) são escondidos, ou seja, cliente só conhece a interface.
- A interação de um objeto com o meio externo é realizada exclusivamente através de seus métodos.



- Reusabilidade de código entre os objetos da classe
  - Todos os objetos instanciados a partir de uma classe incorporam as suas propriedades e seu comportamento
  - Caso não existissem classes, para cada novo objeto criado, seria preciso uma definição completa do objeto.
- Reusabilidade de código de terceiros



- Uma classe
  - é um padrão para uma categoria de itens estruturalmente idênticos e um mecanismo para criar estes itens, as instâncias, baseando-se neste padrão.
- **Objetos de estrutura** e **comportamento** idênticos são descritos em classes
- A descrição das propriedades pode ser feita de uma só vez, independente da quantidade de objetos a serem criados
  - Cada objeto criado a partir de uma classe é denominado de instância
  - Cada instância pertence a uma classe e uma classe pode possuir múltiplas instâncias



- A Classe é um **modelo** para os objetos
  - Na criação de um objeto ele recebe cópia das estruturas de dados, mas os métodos residem nas classes
  - Métodos definem o comportamento dos objetos da classe e este é único
  - Economiza-se o espaço que seria ocupado pelo código dos métodos em cada objeto da classe

# Classes



  
MINISTÉRIO DA DEFESA  
INSTITUTO TECNOLÓGICO DE AERONÁUTICA  
PRO-REITORIA DE PÓS-GRADUAÇÃO E PESQUISA

**DATA DE MATRICULA**  
Consultar o site da Pós-Graduação

**FORMULÁRIO DE INSCRIÇÃO DE DISCIPLINA ISOLADA**

A. DADOS PESSOAIS:

NOME COMPLETO: \_\_\_\_\_

Estado civil: \_\_\_\_\_ Sexo: ☐ Masculino ☐ Feminino

Data de nascimento: \_\_\_\_/\_\_\_\_/\_\_\_\_ Cidade: \_\_\_\_\_ UF: \_\_\_\_\_

FILIAÇÃO: Pai: \_\_\_\_\_ Mãe: \_\_\_\_\_

ENDEREÇO RESIDENCIAL: \_\_\_\_\_ Telefone: \_\_\_\_\_ E-mail: \_\_\_\_\_

ENDEREÇO COMERCIAL: \_\_\_\_\_ Telefone: \_\_\_\_\_ E-mail: \_\_\_\_\_

NACIONALIDADE: ☐ Brasileira ☐ Estrangeira ☐ Naturalizada

RG: \_\_\_\_\_ Passaporte Nº: \_\_\_\_\_ CBE: \_\_\_\_\_

Órgão Emissor: \_\_\_\_\_ Visto validade \_\_\_\_/\_\_\_\_/\_\_\_\_ Validade \_\_\_\_/\_\_\_\_/\_\_\_\_

CK: \_\_\_\_\_

Atenção: Disciplina Isolada em EAD - 1º Sem. - 1º Nível

Encaminhar formulário, com o sigilo de disciplina isolada, ao II Anuário.

  
MINISTÉRIO DA DEFESA  
INSTITUTO TECNOLÓGICO DE AERONÁUTICA  
PRO-REITORIA DE PÓS-GRADUAÇÃO E PESQUISA

**DATA DE MATRICULA**  
Consultar o site da Pós-Graduação

**FORMULÁRIO DE INSCRIÇÃO DE DISCIPLINA ISOLADA**

A. DADOS PESSOAIS:

NOME COMPLETO: **DARTH VADER**

Estado civil: \_\_\_\_\_ Sexo: ☐ Masculino ☐ Feminino

Data de nascimento: \_\_\_\_/\_\_\_\_/\_\_\_\_ Cidade: \_\_\_\_\_ UF: \_\_\_\_\_

FILIAÇÃO: Pai: \_\_\_\_\_ Mãe: \_\_\_\_\_

ENDEREÇO RESIDENCIAL: \_\_\_\_\_ Telefone: \_\_\_\_\_ E-mail: \_\_\_\_\_

ENDEREÇO COMERCIAL: \_\_\_\_\_ Telefone: \_\_\_\_\_ E-mail: \_\_\_\_\_

NACIONALIDADE: ☐ Brasileira ☐ Estrangeira ☐ Naturalizada

RG: \_\_\_\_\_ Passaporte Nº: \_\_\_\_\_ CBE: \_\_\_\_\_

Órgão Emissor: \_\_\_\_\_ Visto validade \_\_\_\_/\_\_\_\_/\_\_\_\_ Validade \_\_\_\_/\_\_\_\_/\_\_\_\_

CK: \_\_\_\_\_

Atenção: Disciplina Isolada em EAD - 1º Sem. - 1º Nível

Encaminhar formulário, com o sigilo de disciplina isolada, ao II Anuário.



**DARTH VADER**  
Estrela da Morte

  
MINISTÉRIO DA DEFESA  
INSTITUTO TECNOLÓGICO DE AERONÁUTICA  
PRO-REITORIA DE PÓS-GRADUAÇÃO E PESQUISA

**DATA DE MATRICULA**  
Consultar o site da Pós-Graduação

**FORMULÁRIO DE INSCRIÇÃO DE DISCIPLINA ISOLADA**

A. DADOS PESSOAIS:

NOME COMPLETO: **YODA**

Estado civil: \_\_\_\_\_ Sexo: ☐ Masculino ☐ Feminino

Data de nascimento: \_\_\_\_/\_\_\_\_/\_\_\_\_ Cidade: \_\_\_\_\_ UF: \_\_\_\_\_

FILIAÇÃO: Pai: \_\_\_\_\_ Mãe: \_\_\_\_\_

ENDEREÇO RESIDENCIAL: \_\_\_\_\_ Telefone: \_\_\_\_\_ E-mail: \_\_\_\_\_

ENDEREÇO COMERCIAL: \_\_\_\_\_ Telefone: \_\_\_\_\_ E-mail: \_\_\_\_\_

NACIONALIDADE: ☐ Brasileira ☐ Estrangeira ☐ Naturalizada

RG: \_\_\_\_\_ Passaporte Nº: \_\_\_\_\_ CBE: \_\_\_\_\_

Órgão Emissor: \_\_\_\_\_ Visto validade \_\_\_\_/\_\_\_\_/\_\_\_\_ Validade \_\_\_\_/\_\_\_\_/\_\_\_\_

CK: \_\_\_\_\_

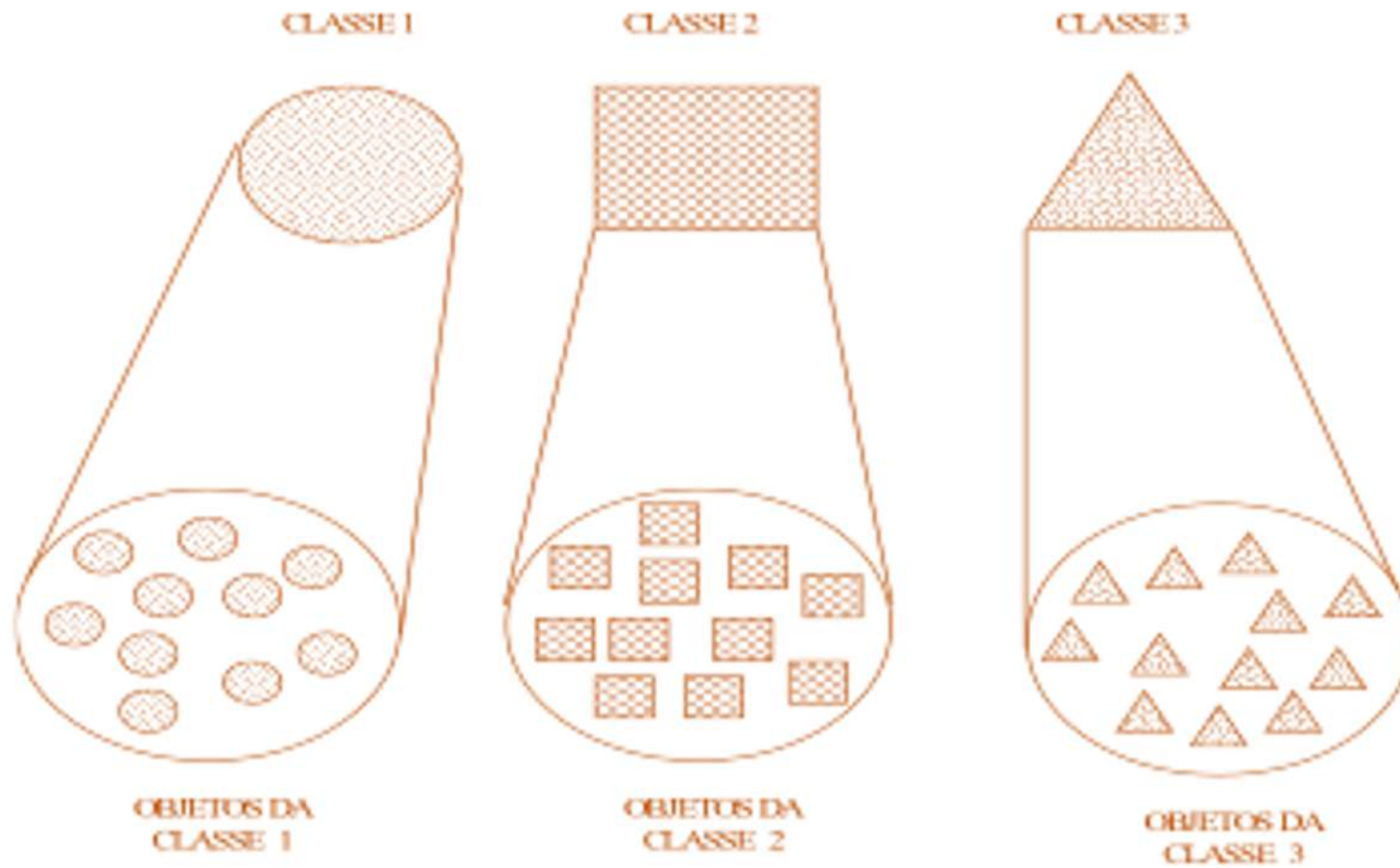
Atenção: Disciplina Isolada em EAD - 1º Sem. - 1º Nível

Encaminhar formulário, com o sigilo de disciplina isolada, ao II Anuário.



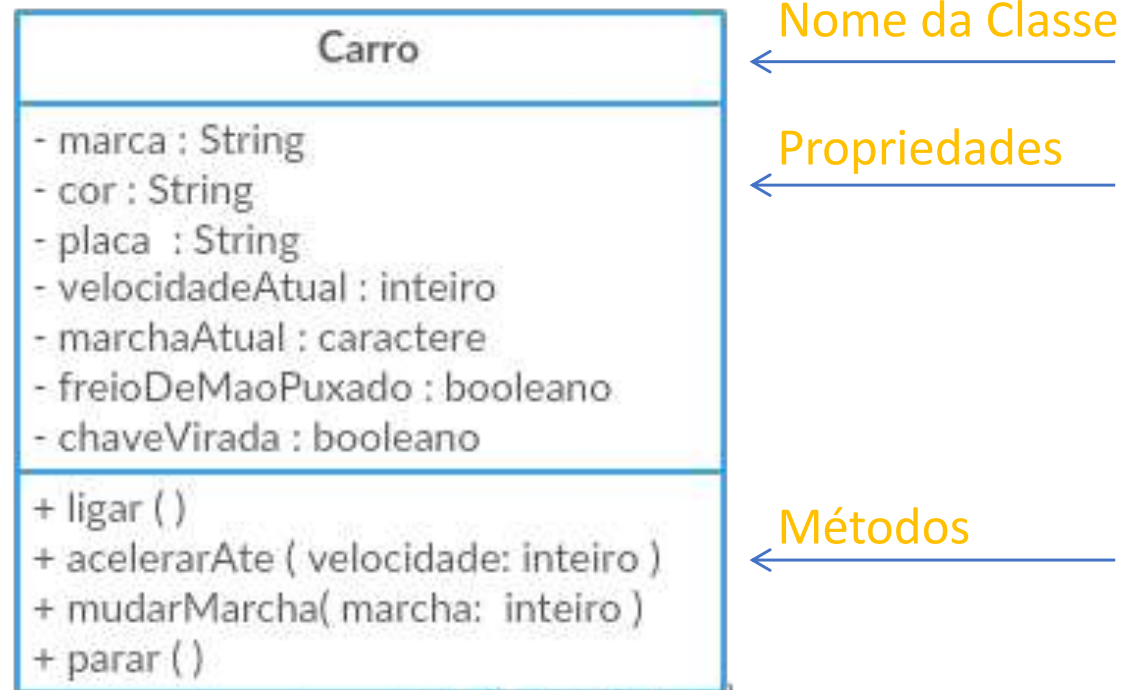
**YODA**

# Classes





- Em UML







- Em Java

```
class Empregado
{
    private String nome;
    private float salario;
    private static int numeroEmpregados=0;

    public Empregado(String umNome)
    {
        nome = umNome;
        salario = 0.0;
        ++numeroDeEmpregados;
    }
    public void setNome(String umNome)
    {
        nome = umNome;
    }
    public String obtemNome() {
        return nome;
    }
    public void setSalario(float salario){
        this.salario = salario;
    }
    public float obtemSalario(){
        return salario;
    }
}
```

## Em C#

```
public class Pessoa {
    // Propriedades da classe
    public string Nome { get; set; }
    public int Idade { get; set; }

    // Construtor da classe
    public Pessoa(string nome, int idade) {
        Nome = nome;
        Idade = idade;
    }

    // Método da classe
    public void Apresentar() {
        Console.WriteLine("Olá, meu nome é " +
Nome + " e eu tenho " + Idade + " anos.");
    }
}
```





# Classes

# Classes: definição



- A declaração de uma classe é composta por:
  - Modificador de acesso: indica a visibilidade da classe (public, protected, private)
  - Identificação: o nome da classe
  - Herança: o nome da superclasse, se houver, precedida de : em C# , extends em Java
  - Interfaces: a lista das interfaces implementadas (caso haja), separadas por vírgulas, precedida pela palavra-chave implements
  - Corpo da classe: envolto por chaves {}, contém os atributos, construtores e métodos da classe.



# Definição de classes



## Exemplo em C#

Modificador de  
Visibilidade

```
public class Pessoa {  
    // Propriedades da classe  
    public string Nome { get; set; }  
    public int Idade { get; set; }  
  
    // Construtor da classe  
    public Pessoa(string nome, int idade) {  
        Nome = nome;  
        Idade = idade;  
    }  
  
    // Métodos  
    public void Apresentar() {  
        Console.WriteLine("Olá, meu nome é " +  
Nome + " e eu tenho " + Idade + " anos.");  
    }  
}
```

Propriedades

Construtor

# Atributos de Instância



- Os atributos de instância:
  - Podem ser de qualquer tipo básico da linguagem
  - Podem ser um objeto de outra classe
  - Pode ser um vetor de qualquer tipo básico ou mesmo um vetor de objetos
  - São automaticamente inicializados pelo compilador se o programador não o fizer explicitamente
  - São copiados para todos os objetos

```
public class Pessoa
{
    // Atributos de instância
    public string Nome { get; set; }
    public int Idade { get; set; }

    // Método de instância
    public void Falar()
    {
        Console.WriteLine("Oi, meu nome é " + Nome + " e eu tenho " + Idade + "
anos.");
    }
}
```

Atributos de Instância



# Atributos de Instância



- São declarados no formato:

```
<modificador> tipo nomeAtributo;
```

- Onde:
  - modificador: indica a visibilidade da variável (public, protected, private)
  - tipo: o tipo (primitivo ou por referência) da variável
  - nomeAtributo: o identificador da variável



- Uma classe pode ter variáveis contendo informações úteis, como:
  - número de objetos instanciados da classe até certo instante
  - valor médio de determinada propriedade
- Essas variáveis não devem ser criadas para cada objeto
  - Para cada classe existirá apenas uma única cópia de cada variável: variável de classe ou atributo de classe
  - A variável de classe existe a partir do ponto em que a classe que a define é carregada na memória





- As variáveis de classe:
  - Podem ser de qualquer tipo básico ou de referência
  - NÃO são copiados para todos os objetos
  - São declarados no formato:

```
<modificador> static tipo nomeAtributoClasse;
```

- Onde:
  - modificador: indica a visibilidade da variável (public, protected, private)
    - Em Java, Se final, seu valor, uma vez atribuído, não pode ser modificado
  - tipo: o tipo (primitivo ou por referência) da variável
  - nomeAtributoClasse: o identificador da variável

# Variável de classe



- Esta variável é da classe e não de cada objeto. Exemplo em Java. Mesma palavra reservada em C#.

```
public class ContaBancaria
{
    private String nomeCorrentista;
    private double saldo;
    private int numero;
    private static int numeroContas;
```

Variável de Classe





# Atributo de instância x de classe



- Considere a classe Quadrado com um único atributo de instância: cor

```
public class Quadrado {  
    public String cor = "Azul";  
  
    public static void main(String[] args) {  
        Quadrado q1 = new Quadrado();  
        Quadrado q2 = new Quadrado();  
  
        System.out.println("A cor do quadrado 1 eh: " + q1.cor);  
        System.out.println("A cor do quadrado 2 eh: " + q2.cor);  
    }  
}
```

```
A cor do quadrado 1 eh: Azul  
A cor do quadrado 2 eh: Azul
```

q1

cor="Azul"

q2

cor="Azul"

# Atributo de instância x de classe

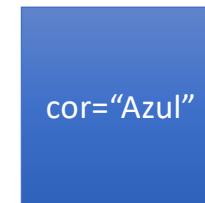


- Considere a classe Quadrado com um único atributo de instância: cor

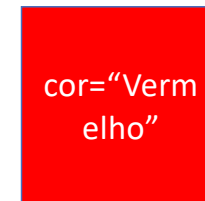
```
public class Quadrado {  
    public String cor = "Azul";  
  
    public static void main(String[] args) {  
        Quadrado q1 = new Quadrado();  
        Quadrado q2 = new Quadrado();  
  
        System.out.println("A cor do quadrado 1 eh: " + q1.cor);  
        System.out.println("A cor do quadrado 2 eh: " + q2.cor);  
        q2.cor = "Vermelho";  
        System.out.println("A cor do quadrado 1 eh: " + q1.cor);  
        System.out.println("A cor do quadrado 2 eh: " + q2.cor);  
    }  
}
```

```
A cor do quadrado 1 eh: Azul  
A cor do quadrado 2 eh: Azul  
A cor do quadrado 1 eh: Azul  
A cor do quadrado 2 eh: Vermelho
```

q1



q2



# Atributo de instância x de classe



- Neste exemplo, o valor padrão do atributo de instância cor é “Azul”, mas:
  - Ele pode ser modificado para cada objeto que o contém
  - Modificar o valor do atributo de uma instância não modifica o de outra
  - O nome do objeto é usado para acessar seu atributo

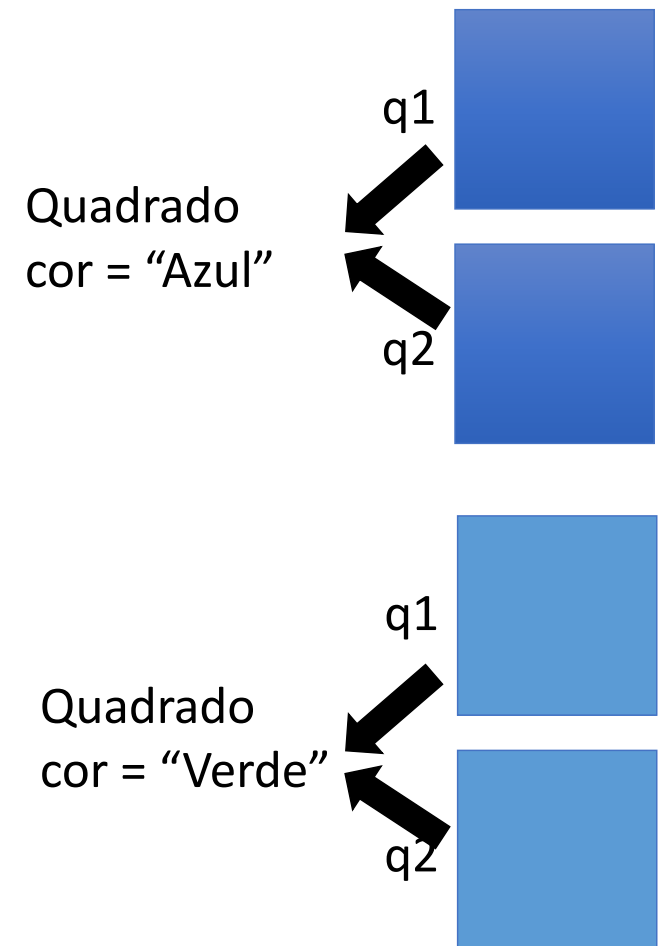
# Atributo de instância x de classe



- Considere agora que o atributo cor foi declarado como sendo da classe.

```
public class Quadrado {  
    public static String cor = "Azul";  
  
    public static void main(String[] args) {  
        Quadrado q1 = new Quadrado();  
        Quadrado q2 = new Quadrado();  
  
        System.out.println("A cor do quadrado 1 eh: " + q1.cor);  
        System.out.println("A cor do quadrado 2 eh: " + q2.cor);  
        q1.cor = "Verde";  
        System.out.println("A cor do quadrado 1 eh: " + q1.cor);  
        System.out.println("A cor do quadrado 2 eh: " + q2.cor);  
    }  
}
```

```
A cor do quadrado 1 eh: Azul  
A cor do quadrado 2 eh: Azul  
A cor do quadrado 1 eh: Verde  
A cor do quadrado 2 eh: Verde
```



# Atributo de instância x de classe

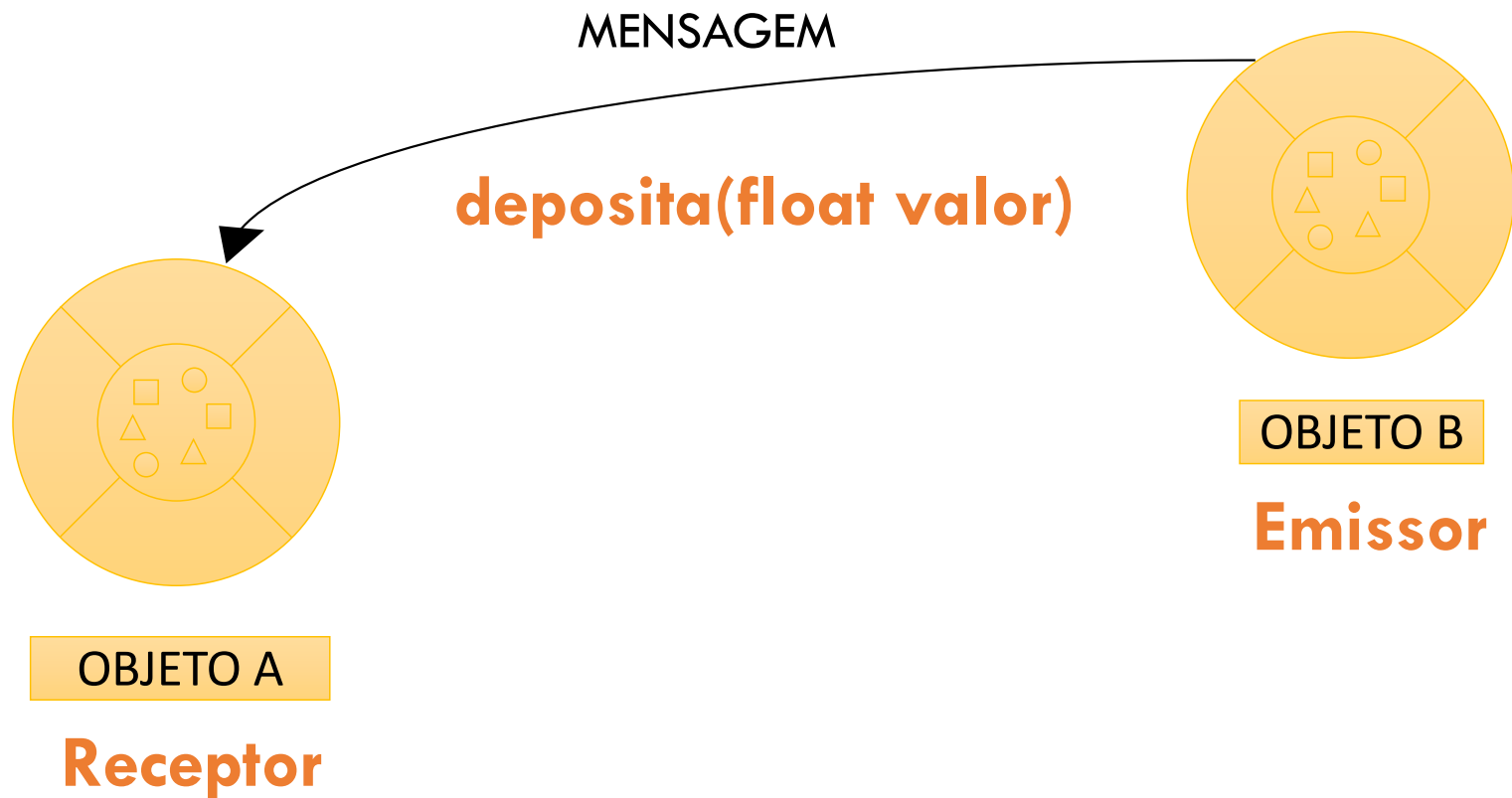


- Como a variável cor agora é um atributo da classe, não precisamos sequer instanciar objetos para acessá-la:

```
public class Quadrado {  
    public static String cor = "Azul";  
  
    public static void main(String[] args) {  
        System.out.println("A cor do qualquer quadrado sera: " + Quadrado.cor);  
    }  
}
```

```
A cor do qualquer quadrado sera: Azul
```

# Mensagens





- Objetos interagem com os outros
  - grande funcionalidade
  - comportamentos mais complexos
- Mecanismos de interação entre objetos: mensagens
- Mensagens são semelhantes a chamadas de subrotinas



- Componentes básicos:
  - o objeto receptor
  - o nome do método que se deseja executar
  - os parâmetros necessários ao método
- Adição em SmallTalk: “5 + 2”
  - A mensagem “+” é enviada ao objeto “5” com o parâmetro “2”
  - O resultado é o objeto “7”





- Benefícios proporcionados pelas mensagens:
  - Ações dos objetos expressadas pelos seus métodos: troca de mensagens suporta todos os tipos possíveis de interação
  - Para trocar mensagens, objetos não precisam estar no mesmo processo ou máquina



- Um método implementa algum aspecto do comportamento do objeto
- Um método é uma subrotina definida na classe que pode acessar o estado interno de um objeto para realizar alguma operação
- Um método é uma sequência de ações, executada por um objeto, que pode alterar ou verificar seu estado (valor dos atributos)
- Enquanto o valor dos atributos (de instância) reside no objeto, o método reside na classe



- Assinatura de um método:
  - Modificadores: tratam da visibilidade do método (public, protected, private) e se ele não pertence ao objeto, mas à classe (static)
  - Tipo de retorno: o tipo de dado (primitivo ou por referência) que o método deverá retornar, ou void se o método não retorna informação
  - Identificador do método: nome do método
  - Lista de parâmetros: lista de argumentos do método;
- Corpo do método:
  - Contém a implementação do método
- Mensagens e Métodos
  - Mensagem é o conceito
  - Método é a implementação



- Assinatura de um método:

```
public double getSaldo(void)
```

- Implementação:

```
public double getSaldo(void)
{
    return saldo;
}
```



- Algumas linguagens implementam métodos especiais:
  - Construtores: usados para criar e inicializar objetos novos
    - Em C++, C# e Java ganham o mesmo nome da classe
    - Pode-se ter várias versões de construtores para cada classe, mudando a quantidade e o tipo de parâmetros
  - Destrutores: usados para destruir objetos
    - Em C++ e C# ganham o nome da classe precedido de um ~ (til) e não recebem parâmetros.
    - Java não implementa o conceito de destrutor e faz a coleta de lixo automática. C# também o faz

# Construtores e Destrutores



- Construtor
  - Possuem o mesmo nome da classe
  - Devem ser públicos
  - Não tem tipo de retorno
  - É um método especial da classe responsável pela inicialização de um objeto
  - Se não for explicitamente declarado um construtor, o compilador fornece um construtor padrão que configura as variáveis de instância para seus valores padrão, que dependem do tipo de cada
    - Invocado apenas 1 vez por objeto criado

```
public ContaBancaria(String n, double s)
{
    nomeCorrentista = n;
    saldo = s;
}
```



- São métodos criados para permitir que atributos protegidos possam ter seu valores lidos e modificados por elementos de outras classes
- No geral, são distribuídos em dois grupos:
  - Sets: permitem que o atributo em questão seja alterado
  - Gets: permite que o valor corrente do atributo seja lido



- Exemplos

```
public double getSaldo()  
{  
    return saldo;  
}  
public void setSaldo(double s)  
{  
    saldo = s;  
}
```



# Sobrecarga de Métodos



- Podemos ter métodos com o mesmo nome desde que tenham assinaturas diferentes. Quando isso acontece, dizemos que temos sobrecarga de métodos (overloading).
- O tipo de retorno e a visibilidade dos métodos não entra como um critério de diferenciação entre métodos, portanto declarar mais de um método com a mesma lista de parâmetros, ainda que com tipos de retorno distintos, resulta em erro de compilação.
- Exemplo:

```
public ContaBancaria()  
{  
    saldo = 0;  
}  
public ContaBancaria(String n, double s)  
{  
    nomeCorrentista = n;  
    saldo = s;  
}
```

# Java: Instanciando Objetos



```
public class Main
{
    public static void main(String args[])
    {
        ContaBancaria cb;

        cb = new ContaBancaria();
        System.out.println("Saldo atual: " + cb.getSaldo());
    }
}
```

Saldo atual: 0.0

Program exited with code #0 after 2.39 seconds.


[copy output](#)

# Java: Instanciando Objetos



```
public class Main
{
    public static void main(String args[])
    {
        ContaBancaria cb;

        cb = new ContaBancaria();
        System.out.println("Saldo atual: " + cb.getSaldo());
        cb.deposita(100);
        System.out.println("Saldo atual: " + cb.getSaldo());
    }
}
```



```
Saldo atual: 0.0
Saldo atual: 100.0
```

Program exited with code #0 after 0.77 seconds.

[copy output](#)

# C#: Instanciando Objetos



```
public class Pessoa
{
    // Atributos de instância
    public string Nome { get; set; }
    public int Idade { get; set; }

    // Método de instância
    public void Falar()
    {
        Console.WriteLine("Oi, meu nome é " + Nome + " e eu tenho " + Idade + " anos.");
    }
}

// Utilizando a classe Pessoa
static void Main(string[] args)
{
    // Criando um objeto da classe Pessoa
    Pessoa pessoa1 = new Pessoa();
    pessoa1.Nome = "João";
    pessoa1.Idade = 30;
    pessoa1.Falar();

    // Criando outro objeto da classe Pessoa
    Pessoa pessoa2 = new Pessoa();
    pessoa2.Nome = "Maria";
    pessoa2.Idade = 25;
    pessoa2.Falar();
}
```

The background features a network of gray lines connecting various colored circles (orange, yellow, blue, green) in a non-linear fashion. A dark blue horizontal band with a yellow border is positioned across the middle of the slide.

# Variáveis e Métodos



- Em Orientação a Objetos, há três tipos de módulos:
  - Pacotes: na tecnologia Java nada mais é do que um conjunto de classes localizadas na mesma estrutura hierárquica de diretórios
  - Classes: são projetos de um objeto, aonde têm características e comportamentos, ou seja, permite armazenar propriedades e métodos dentro dela
  - Métodos: são serviços implementados na forma de um conjunto de instruções da linguagem (procedimentos algorítmicos) que realizam alguma tarefa específica e podem, como resultado, retornar um valor



```
package meusuperpacote.meupacote;  
  
[modificador] class NomeDaClasse {  
  
    // Atributos da classe e atributos de instância  
  
    // Métodos da classe e métodos de instância  
    [modificador] tipoRetorno nomeMétodo ([parâmetros]) {  
        // Corpo do método  
    }  
}
```

# Modularização: Benefícios



- Abordagem dividir-para-conquistar: facilita a construção de programas a partir de blocos menores e mais simples
  - Facilidade em depurar e validar
- Reutilização de código: podem ser usados em várias partes do programa, quantas vezes for necessário, no caso dos métodos. Podem ser importados em outros projetos, nos casos das classes ou pacotes.
- Ocultamento de código: uma vez que apenas as rotinas e sub-rotinas ficam disponíveis para outros programadores





- Parâmetros e argumentos
  - Parâmetros referem-se à lista de variáveis declarada em um método
  - Argumentos referem-se aos valores que são passados em uma chamada de função.
- Quando um método é chamado, os argumentos precisam ser correspondentes aos tipos e à ordem dos parâmetros declarados
  - Os tipos dos parâmetros podem ser primitivos ou referenciados.
  - A linguagem Java não permite a passagem de métodos como parâmetro, no entanto, é possível passar um objeto contendo métodos que poderão ser chamados

# Métodos: Passagem de Parâmetros



- Passagem de tipos primitivos
  - Em Java, a passagem de argumentos de tipos primitivos é realizada por valor
  - Em C#, pode ser por valor (padrão) ou por referência (usando a palavra ref)
  - Isso implica que qualquer mudança de valor dos parâmetros persiste somente no escopo do método
  - Quando o método retorna, os parâmetros são liberados da memória e os valores dos argumentos são recuperados para seus valores iniciais



- Passagem de tipos referenciados
  - Em Java, a passagem de argumentos de tipos referenciados, como objetos e vetores, também é realizada por valor (cópia do valor do endereço do objeto)
  - Em C#, pode ser por valor ou por referência (usando a palavra ref)
  - Isso significa que após o retorno do método o argumento vai continuar referenciando o mesmo objeto de antes da chamada do método
  - Ainda assim, os atributos do objeto referenciado poderão ter seus valores modificados, se seus modificadores de acesso assim o permitirem

# Métodos: Passagem de Parâmetros



```
public class Cliente
{
    private String nome;           // propriedade nome do cliente
    private String cpf;           // propriedade cpf do cliente
    private int idade;            // propriedade idade do cliente

    public Cliente(String nome, String cpf, int idade)
    {
        this.nome = nome;
        this.cpf = cpf;
        this.idade = idade;
    }

    public String getNome()
    {
        return nome;
    }

    public String getCpf()
    {
        return cpf;
    }

    public int getIdade()
    {
        return idade;
    }

    public void setNome(String nome)
    {
        this.nome = nome;
    }

    public void setIdade(int idade)
    {
        this.idade = idade;
    }
}
```

```
/* @Override */
public String toString()
{
    String out = "";
    out += "Cliente: " + getNome() + " de cpf # " + getCpf() + "\nIdade: " + getIdade();
    return out;
}
}
```

# Métodos: Passagem de Parâmetros



- Observe que os objetos c1 e c2 permanecem intactos!

```
class TesteCliente {
    public static void main(String[] args) {
        Cliente c1 = new Cliente("Albus Dumbledore", "123.456", 1000);
        Cliente c2 = new Cliente("Sirius Black", "555.555", 200);
        System.out.println("***** Antes de chamar o método *****");
        System.out.println(c1);
        System.out.println(c2);
        trocaPersonagem(c1, c2);
        System.out.println("***** Após chamada do método *****");
        System.out.println(c1);
        System.out.println(c2);
    }

    public static void trocaPersonagem(Cliente cc1, Cliente cc2) {
        Cliente temp;
        temp = cc1;
        cc1 = cc2;
        cc2 = temp;
        System.out.println("***** Dentro do método *****");
        System.out.println(cc1);
        System.out.println(cc2);
    }
}
```

```
***** Antes de chamar o método *****
Cliente: Albus Dumbledore de cpf # 123.456
Idade: 1000
Cliente: Sirius Black de cpf # 555.555
Idade: 200
***** Dentro do método *****
Cliente: Sirius Black de cpf # 555.555
Idade: 200
Cliente: Albus Dumbledore de cpf # 123.456
Idade: 1000
***** Após chamada do método *****
Cliente: Albus Dumbledore de cpf # 123.456
Idade: 1000
Cliente: Sirius Black de cpf # 555.555
Idade: 200
```



# Métodos: Passagem de Parâmetros



- Observe que o objeto c1 teve sua propriedade nome alterada!

```
class TesteCliente {
    public static void main(String[] args) {
        Cliente c1 = new Cliente("Albus Dumbledore", "123.456", 1000);
        Cliente c2 = new Cliente("Sirius Black", "555.555", 200);
        System.out.println("***** Antes de chamar o método *****");
        System.out.println(c1);
        System.out.println(c2);
        trocaPersonagem(c1, c2);
        System.out.println("***** Após chamada do método *****");
        System.out.println(c1);
        System.out.println(c2);
    }

    public static void trocaPersonagem(Cliente cc1, Cliente cc2) {
        Cliente temp;
        temp = cc1;
        cc1 = cc2;
        cc2 = temp;
        System.out.println("***** Dentro do método *****");
        System.out.println(cc1);
        System.out.println(cc2);
        cc2.setNome("Lord Voldemort");
    }
}
```

```
***** Antes de chamar o método *****
Cliente: Albus Dumbledore de cpf # 123.456
Idade: 1000
Cliente: Sirius Black de cpf # 555.555
Idade: 200
***** Dentro do método *****
Cliente: Sirius Black de cpf # 555.555
Idade: 200
Cliente: Albus Dumbledore de cpf # 123.456
Idade: 1000
***** Após chamada do método *****
Cliente: Lord Voldemort de cpf # 123.456
Idade: 1000
Cliente: Sirius Black de cpf # 555.555
Idade: 200
```



- Muitas vezes faz-se necessário acessar parte do comportamento de uma classe independentemente de suas instâncias
  - Método `dataAtual` em uma classe `Data`
  - Método `new` que cria um novo objeto da classe
- Neste caso, o comportamento é da classe e não de um objeto em particular
  - É como se a classe tivesse operações
  - próprias
    - Método de classe



# Métodos de classe: Declaração e Acesso



- Para declarar um método de classe, usaremos a palavra chave static
- Por exemplo, um método estático myStaticMethod() pertencente a uma classe MyClass, pode ser chamado da seguinte forma:
- Se o método myStaticMethod for chamado dentro da classe em que foi definido, o nome da classe pode ser omitido.

```
MyClass.myStaticMethod( [argumentos] );
```



# Métodos de classe: Declaração e Acesso



- ContaBancaria.java

```
public static int getContas()  
{  
    return numeroContas;  
}
```

```
public class Main  
{  
    public static void main(String args[])  
    {  
        ContaBancaria cb1, cb2;  
  
        cb1 = new ContaBancaria();  
        System.out.println("Número de contas: " + ContaBancaria.getContas());  
        cb2 = new ContaBancaria();  
        System.out.println("Número de contas: " + ContaBancaria.getContas());  
    }  
}
```

```
Número de contas: 1  
Número de contas: 2
```

Program exited with code #0 after 0.89 seconds.

[copy output](#)



- Métodos de classe ou estáticos são bastante úteis quando desejamos construir, por exemplo, uma biblioteca de funções
- A API de Java define várias classes com métodos estáticos:
  - Vamos usar o exemplo da classe Math



- A classe Math contém muitos métodos estáticos para realizar operações matemáticas como exponenciação, logaritmo, raiz quadrada e funções trigonométricas
  - Está definida no pacote java.lang, implicitamente importado pelo compilador
  - A chamada de um método da classe Math segue a mesma estrutura dos métodos estáticos:
    - `Math.nomeMetodoEstatico([argumentos]);`

# Classe Math: exemplo



```
class ExemploClasse {  
    public static void main(String[] args) {  
        float precoProdutoA[] = { 11.2f, 15.12f };  
        float precoProdutoB[] = { 19.7f, 20 };  
        System.out.println("O maior preço do produto A é "  
            + Math.max(precoProdutoA[0], precoProdutoA[1]));  
        System.out.println("O menor preço do produto B é "  
            + Math.min(precoProdutoB[0], precoProdutoB[1]));  
    }  
}
```

```
O maior preço do produto A é 15.12  
O menor preço do produto B é 19.7
```



- Vamos construir uma classe `ConversaoDeUnidadesDeTemperatura` que contenha métodos estáticos para calcular a conversão entre diferentes escalas de temperatura
  - Considere as fórmulas:
    - De graus Celsius(C) para graus Fahrenheit(F):  $F = (9 * C/5) + 32$
    - De graus Fahrenheit (F) para graus Celsius (C):  $C = (F - 32) * 5/9$
    - De graus Celsius (C) para graus Kelvin (K):  $K = C + 273.15$
    - De graus Kelvin (K) para graus Celsius (C):  $C = K - 273.15$
    - De graus Celsius (C) para graus Réaumur (Re):  $Re = C * 4/5$
    - De graus Réaumur (Re) para graus Celsius (C):  $C = Re * 5/4$
    - De graus Kelvin (K) para graus Rankine (R):  $R = K * 1.8$
    - De graus Rankine (R) para graus Kelvin (K):  $K = R/1.8$

# Métodos de Classe: Exemplo



```
class ConversaoDeUnidadesDeTemperatura {  
  
    public static double celsiusToFahrenheit (double c) {  
        double f = ((9 * c/5) + 32);  
        return f;  
    }  
  
    public static double fahrenheitToCelsius (double f) {  
        double c = (f - 32) * 5/9;  
        return c;  
    }  
  
    public static double celsiusToKelvin (double c) {  
        double k = c + 273.15;  
        return k;  
    }  
  
    public static double KelvinToCelsius (double k) {  
        double c = k - 273.15;  
        return c;  
    }  
  
    public static double celsiusToReaumur (double c) {  
        double re = c * 4/5;  
        return re;  
    }  
}
```

```
    public static double ReaumurToCelsius (double re) {  
        double c = re * 5/4;  
        return c;  
    }  
  
    public static double KelvinToRankine (double k) {  
        double r = k * 1.8;  
        return r;  
    }  
  
    public static double RankineToKelvin (double r) {  
        double k = r / 1.8;  
        return k;  
    }  
}
```



# Métodos de Classe: Exemplo



```
import java.util.Scanner;

class TesteConversao {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.println("Entre com a temperatura em Celsius: 23.2");
        double c = input.nextDouble();
        System.out.printf("%.2f graus celsius equivale a:\n",c);
        System.out.printf("\t %.2f graus Fahrenheit\n", ConversaoDeUnidadesDeTemperatura.celsiusToFahrenheit(c));
        System.out.printf("\t %.2f graus Kelvin\n", ConversaoDeUnidadesDeTemperatura.celsiusToKelvin(c));
        System.out.printf("\t %.2f graus Reaumur\n", ConversaoDeUnidadesDeTemperatura.celsiusToReaumur(c));
        System.out.printf("\t %.2f graus Rankine\n", ConversaoDeUnidadesDeTemperatura.KelvinToRankine(ConversaoDeUnidadesDeTemperatura.celsiusToKelvin(c)));
    }
}
```

```
Entre com a temperatura em Celsius: 23.2
23,20 graus celsius equivale a:
    73,76 graus Fahrenheit
    296,35 graus Kelvin
    18,56 graus Reaumur
    533,43 graus Rankine
```

nciar um objeto da classe para que



- Por que o método main é declarado estático?
  - Para que a JVM possa executar esse método sem necessidade de instanciar um objeto da classe que o contém



# Métodos de Classe: Exemplo II



- Métodos de Classe e Sobrecarga
  - Considere que precisamos ter vários métodos que devem realizar a soma entre duas variáveis
  - Devemos ter métodos que somam:
    - 2 inteiros
    - 2 doubles
    - 2 floats
    - 2 Strings
  - Como implementar?

# Métodos de Classe: Exemplo II



```
public class Mat {  
  
    public static int soma(int x1, int x2) {  
        return x1+x2;  
    }  
  
    public static double soma(double x1, double x2) {  
        return x1+x2;  
    }  
  
    public static float soma(float x1, float x2) {  
        return x1+x2;  
    }  
  
    public static String soma(String x1, String x2) {  
        String out = "";  
        out += x1 + x2;  
        return out;  
    }  
  
    public static void main(String args[]) {  
        System.out.println("Soma de 2 e 3: " + soma(2,3) + "\n");  
        System.out.println("Soma de 2.3f e 3.2f: " + soma(2.3f,3.2f) + "\n");  
        System.out.println("Soma de 2.3 e 3.2: " + soma(2.3,3.2) + "\n");  
        System.out.println("Soma de boa e noite: " + soma("boa","noite") + "\n");  
    }  
}
```

Soma de 2 e 3: 5

Soma de 2.3f e 3.2f: 5.5

Soma de 2.3 e 3.2: 5.5

Soma de boa e noite: boa noite



- Um método estático não consegue enxergar métodos e atributos não estáticos da classe
- Para que um método consiga acessar um método ou atributo não estático da classe, é necessário que o método tenha a referência de um objeto
  - Justificativa: supondo que um método estático chame um método não estático diretamente.
    - Como saber a qual objeto o método estático está se referindo? E se nenhum objeto da classe existir no momento da chamada?



- Resumo das visibilidades entre métodos e atributos de instância e de classe
  - Métodos de instância;
    - podem acessar variáveis de instância e métodos de instância diretamente
    - podem acessar variáveis de classe e métodos de classe diretamente
  - Métodos de classe:
    - podem acessar variáveis de classe e métodos de classe diretamente
    - não podem acessar variáveis de instância e métodos de instância diretamente. É necessário o uso de uma referência a um objeto. Além disso, métodos de classe não podem utilizar a palavra-chave `this` dado que não há nenhuma instância associada



- O escopo de declaração de uma classe, método ou variável refere-se à porção do código onde a entidade declarada pode ser chamada pelo seu nome
- Java possui as seguintes regras básicas de escopo:
  - O escopo de um parâmetro encontra-se no corpo do método onde ele está declarado
  - O escopo de uma variável local se inicia no ponto de sua declaração até o final do bloco (método, laço, desvio condicional) que a contém
  - O escopo de um método ou atributo é definido pelo corpo da classe onde eles estão declarados
- Se uma variável local ou parâmetro tem o mesmo nome de um atributo, então o atributo é ocultado em um processo denominado **sombreamento**



# Herança



- Relação do tipo generalização/especialização
  - Objetos de um elemento especializado (um filho) podem ser substituídos por objetos do elemento geral (o pai)
  - É uma maneira natural de modelar o mundo real
  - Relação transitiva e anti-simétrica entre classes
  - A modelagem pode ser testada verificando se o elemento especializado “é do tipo” do elemento geral
- Generalização
  - nome do relacionamento
- Herança
  - mecanismo que implementa o relacionamento

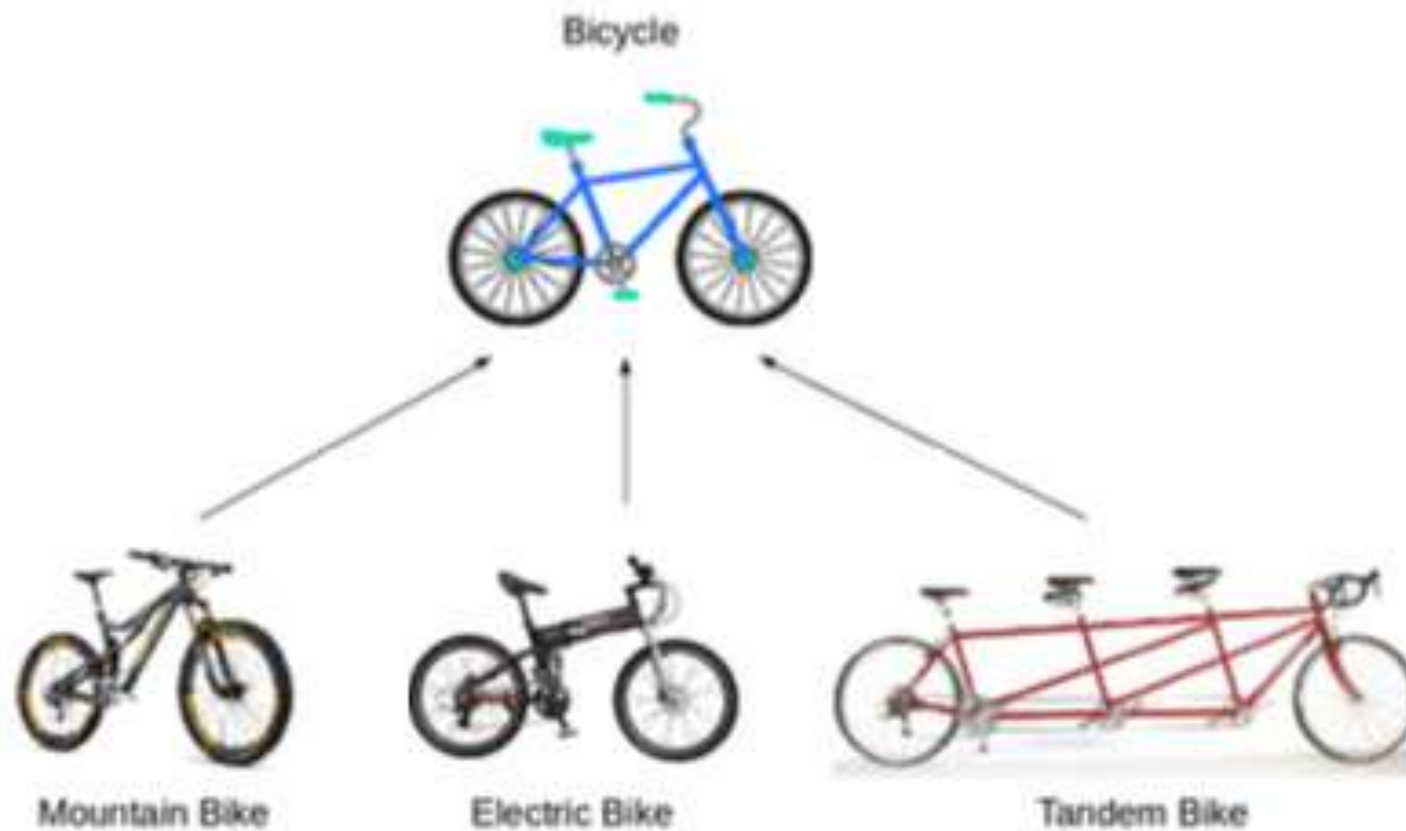


- Herança é o mecanismo através do qual elementos mais específicos incorporam a estrutura e comportamento de elementos mais gerais
  - Super-Classe (classe mãe ou classe base) - Elemento mais geral utilizado para evitar as redundâncias de descrição.
  - Sub-Classe (classe filha ou classe) Elemento mais específico utilizado para descrever as particularidades das classes.
- A sub-classe pode ser utilizada onde a super-classe é mas não o contrário.
- Serve para reutilizar campos e métodos já criados, descrevendo as diferenças entre classes existentes e novas classes
- Um objeto de uma subclasse também é um objeto de uma superclasse.





- Atributos, operações e relacionamentos comuns devem ser representados no mais alto nível da hierarquia
- O que é herdado:
  - As sub-classes criadas herdam o estado (atributos e relacionamentos) e o comportamento (operações) da(s) super-classe(s) associada(s) (herança simples ou múltipla)
  - As sub-classes podem incluir, alterar ou suprimir atributos, operações e relacionamentos da super-classe
  - Os métodos construtores não são herdados, mas são acessíveis às subclasses



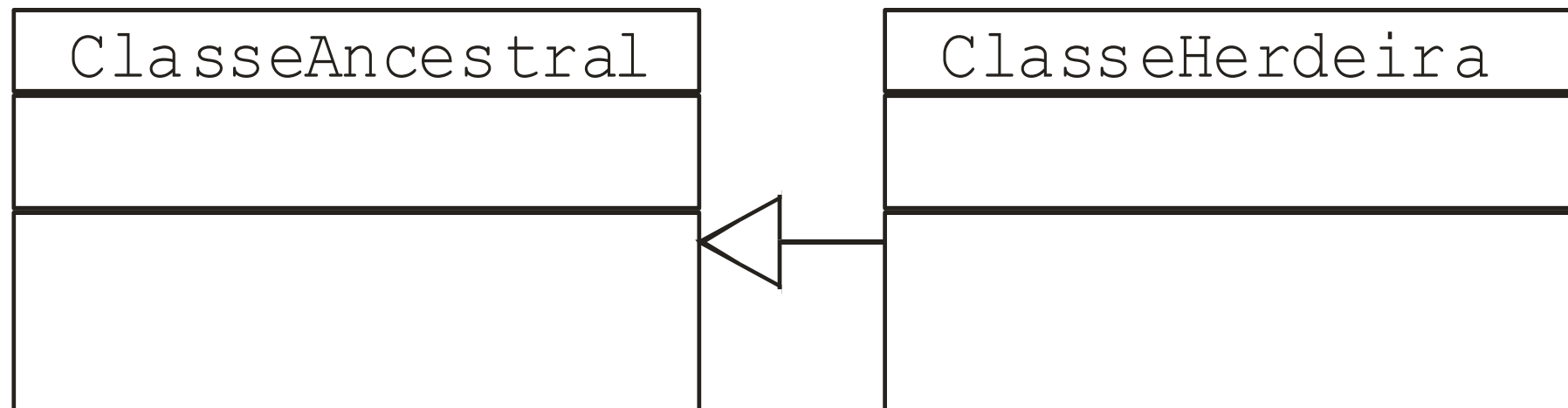


- Classe primordial Object (Java) ou System (C#)
  - Em Java, a classe Object é denominada classe primordial, pois não possui superclasse e é superclasse de todas as demais classes.
  - Uma hierarquia de classes começa com a classe Object (pacote java.lang), a partir da qual todas as classes herdam direta ou indiretamente.
  - Os métodos da classe Object são herdados por todas as classes.

# Herança: Representação



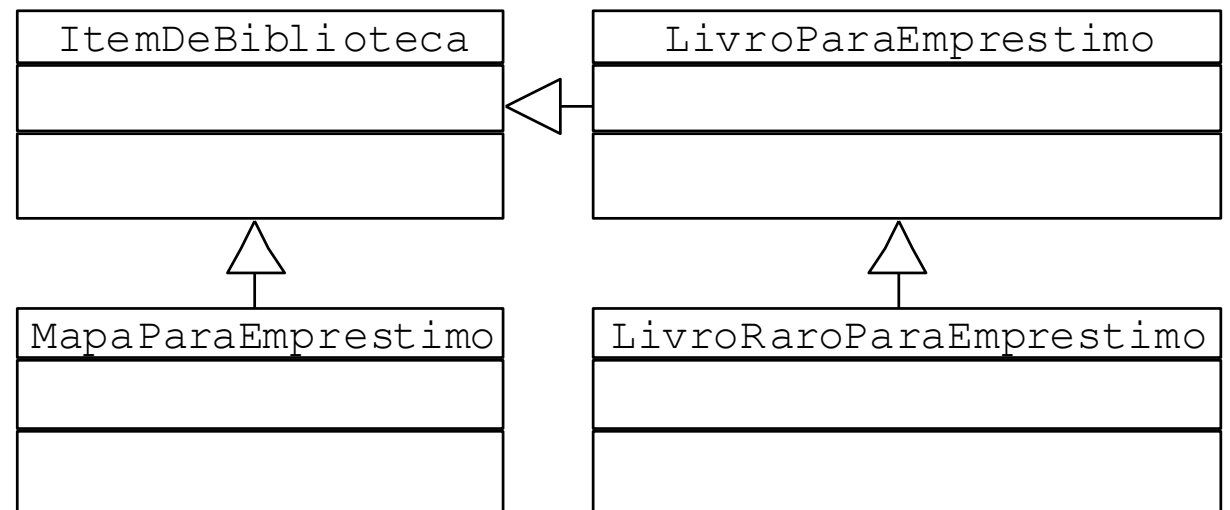
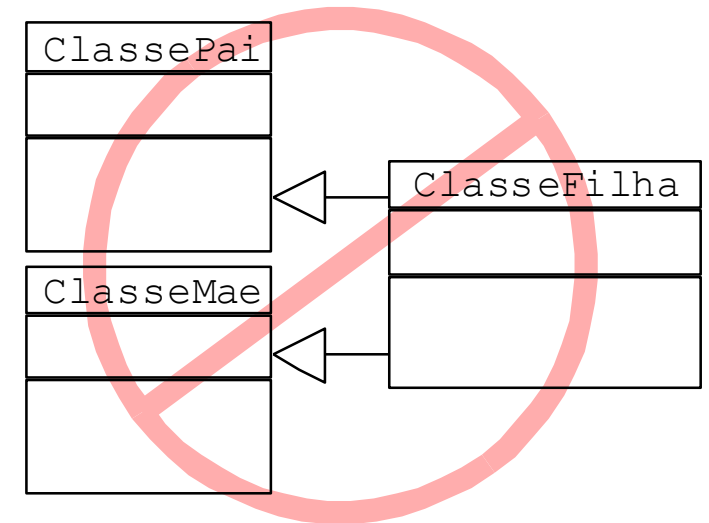
- ClasseHerdeira herda todos os campos e métodos da ClasseAncestral
- Classe herdeira só pode acessar diretamente campos e métodos públicos da classe ancestral!
- Importância de métodos setXXX e getXXX para acessar e modificar valores de campos



# Herança: Herança Múltipla



- Herança Múltipla: classe herdeira herda de duas ou mais classes ancestrais: não existe em Java e C#
- Uma classe ancestral pode ter várias classes descendentes
- Uma classe herdeira pode ter, por sua vez, outras classes herdeiras.



# Herança: Exemplos

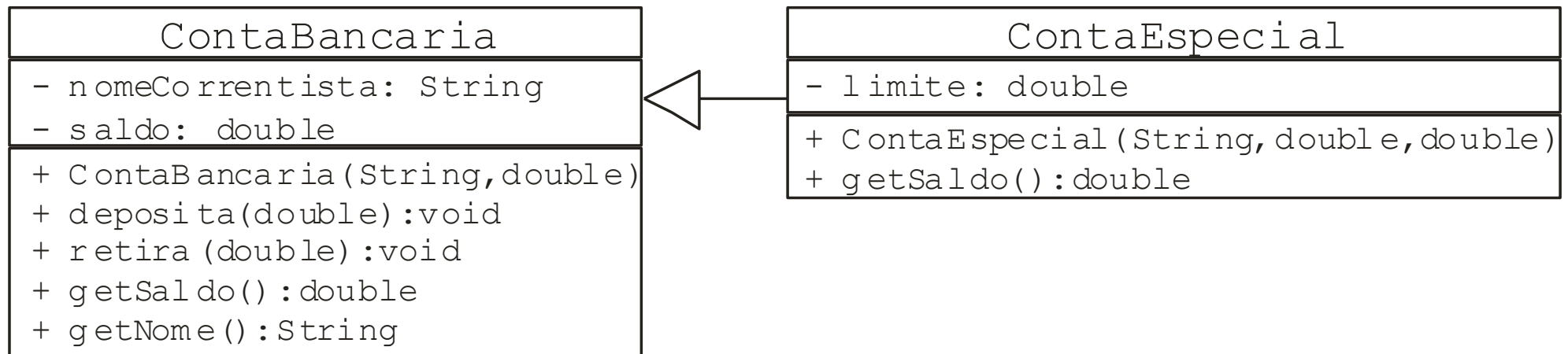


- ContaEspecial é um tipo de ContaBancaria
- ContaPoupanca é um tipo de ContaBancaria
- Gerente é um tipo de Funcionario
- LivroParaEmprestimo é um tipo de ItemDoAcervo

# Exemplo 1



- ContaEspecial herdará todos os métodos de ContaBancaria menos o construtor e mais os definidos para ContaEspecial
- Conta especial simulada como tendo saldo adicional (limite)
- Problema: método `getSaldo()` de `ContaBancaria` não serviria para `ContaEspecial`!
- Solução: sobrescrita do método.



# Exemplo 1: ContaBancaria



```
public class ContaBancaria
{
    private String nomeCorrentista;
    private double saldo;
    private int numero;
    private static int numeroContas=0;

    public ContaBancaria()
    {
        saldo = 0;
        numeroContas++;
    }
    public ContaBancaria(String n,double s)
    {
        nomeCorrentista = n;
        saldo = s;
        numeroContas++;
    }
    public double getSaldo()
    {
        return saldo;
    }
}
```

```
    public void setSaldo(double s)
    {
        saldo = s;
    }
    public void deposita(double quantia)
    {
        saldo = saldo + quantia;
    }
    public void retira(double quantia)
    {
        if (saldo >= quantia)
            saldo = saldo - quantia;
    }
    public static int getContas()
    {
        return numeroContas;
    }
}
```



# Exemplo 1: ContaEspecial



```
public class ContaEspecial extends ContaBancaria
{
    private double limite;

    public ContaEspecial()
    {
        super();
        limite = 0;
    }

    public ContaEspecial(String n, double s, double l)
    {
        super(n,s);
        limite = l;
        deposita(limite); // Simula o dinheiro a mais do limite!
    }

    // Método sobrecarregado
    public double getSaldo()
    {
        return super.getSaldo()-limite;
    }
}
```

Indica qual classe será usada como base (ancestral) para herança

Chamada do **construtor** da classe ancestral com os argumentos

Chamada do método **getSaldo** da classe ancestral



- A subclasse ContaEspecial possui os atributos e métodos da superclasse ContaBancaria, mas o adicional da subclasse só implementa o que for necessário na mesma
  - Isso torna o código da subclasse mais fácil de entender
  - tem um atributo a mais que a superclasse
    - limite



- Os métodos construtores não são herdados, mas são acessíveis às subclasses
  - Chamada de construtores da superclasse
    - A primeira tarefa de um construtor de uma subclasse é chamar o construtor da superclasse, explícita ou implicitamente, para assegurar que as variáveis de instância herdadas serão inicializadas corretamente
    - Se o construtor da subclasse não chama o construtor da superclasse explicitamente, então o compilador gera uma instrução que chama o construtor default ou o construtor sem argumento da superclasse. Se não houver tal construtor na superclasse ocorre um erro de compilação
    - A chamada explícita de um construtor de superclasse deve ser a primeira instrução a ser chamada no construtor da subclasse e sua sintaxe é fornecida a seguir:
      - `super(argumentos);`

# Herança: Sobrescrita de métodos



- Sobrescrita (override) está diretamente relacionada à herança
- Permite especializar os métodos herdados das superclasses, alterando o seu comportamento nas subclasses por um mais específico.
  - Cria-se um novo método na classe filha contendo a mesma assinatura e mesmo tipo de retorno do método sobrescrito
  - O método deve possuir a mesma assinatura
    - nome
    - quantidade de parâmetros
    - tipo de parâmetros
- Tipo de retorno
  - pode ser um subtipo do tipo de retorno do método sobrescrito
    - exemplo: List (superclasse) -> ArrayList (subclasse)

# Herança: Sobreescrita de métodos



- Como se sabe que método invocar?
  - Primeiramente busca-se um método com o nome indicado na classe atual
  - Caso o método exista na subclasse, este método será executado
  - Caso contrário, busca-se e se o método na superclasse

Método da superclasse

```
public double getSaldo()  
{  
    return saldo;  
}
```

Método da subclasse

```
// Método sobrescrito  
public double getSaldo()  
{  
    return super.getSaldo()-limite;  
}
```

# Sobrescrita x Sobrecarga



- Sobrescrita
  - Mesmo nome
  - Mesma assinatura
  - Requer herança
- Sobrecarga
  - Mesmo nome
  - Assinaturas diferentes (ordem ou tipo dos parâmetros – não inclui o retorno)
  - Mesma classe ou na hierarquia

```
public ContaEspecial()  
{  
    super();  
    limite = 0;  
}  
public ContaEspecial(String n, double s, double l)  
{  
    super(n,s);  
    limite = l;  
    deposita(limite); // Simula o dinheiro a mais do limite!  
}
```

## Sobrecarga

```
public double getSaldo()  
{  
    return saldo;  
}  
// Método sobrescrito  
public double getSaldo()  
{  
    return super.getSaldo()-limite;  
}
```

## Sobrescrita



# Exemplo 1: Main



```
public class Main
{
    public static void main(String args[])
    {
        ContaBancaria minha = new ContaBancaria("Eu",200);
        System.out.println("Saldo inicial da minha conta: " + minha.getSaldo());
        minha.retira(120);
        minha.retira(100);
        System.out.println("Saldo da minha conta: " + minha.getSaldo());
        System.out.println("*****");
        ContaEspecial bill = new ContaEspecial("Bill",10000,2000);
        System.out.println("Saldo inicial da conta do Bill: " + bill.getSaldo()); // -1000
        bill.retira(5000);
        bill.retira(6000);
        System.out.println("Saldo da conta do Bill: " + bill.getSaldo()); // -1000
        bill.retira(2000);
        System.out.println("Saldo da conta do Bill: " + bill.getSaldo()); // -1000
    }
}
```

```
Saldo inicial da minha conta: 200.0
Saldo da minha conta: 80.0
*****
Saldo inicial da conta do Bill: 10000.0
Saldo da conta do Bill: -1000.0
Saldo da conta do Bill: -1000.0
```



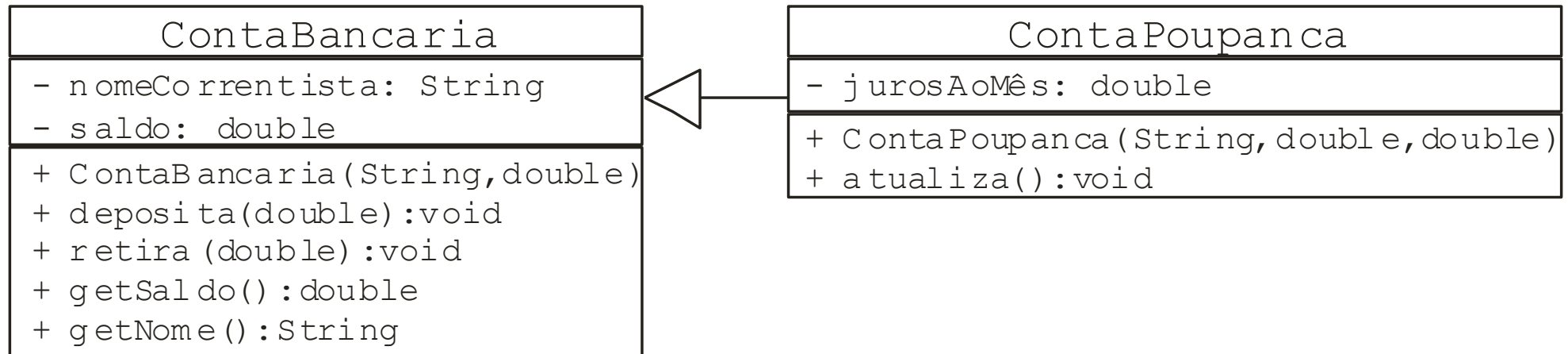
- Regras para métodos:
  - Chamar `super.nomeDoMétodo` com argumentos se existirem
  - Qualquer método público da classe ancestral pode ser chamado a partir de qualquer método da classe descendente
  - Pode ser usado para distinguir entre métodos da classe ancestral e métodos superpostos da própria classe
  - `super` representa a classe ancestral onde necessário
  - Pode ser usado para acessar métodos e campos públicos
  - Só pode ser usado com relação à classe imediatamente superior
    - `super.super.metodoInstancia();` ☹ Não!!!!



# Exemplo 2: ContaPoupanca



- Herda praticamente tudo de ContaBancaria
- Não sobrepõe nenhum método
- Método atualiza calcula juros e atualiza saldo (deve ser chamado todo mês)



# Exemplo 2: ContaPoupanca



```
public class ContaPoupanca extends ContaBancaria
{
    private double jurosAoMes;

    public ContaPoupanca(String n, double s, double j)
    {
        super(n, s);
        jurosAoMes = j;
    }

    public void atualiza()
    {
        double valor = getSaldo() * (jurosAoMes / 100);
        deposita(valor);
    }
}
```

Indica qual classe será usada como base (ancestral) para herança

Chamada do **construtor** da classe ancestral com os argumentos

Chamada do método **getSaldo** da classe ancestral

Chamada do método **deposita** da classe ancestral com o argumento

# Exemplo 2: Main



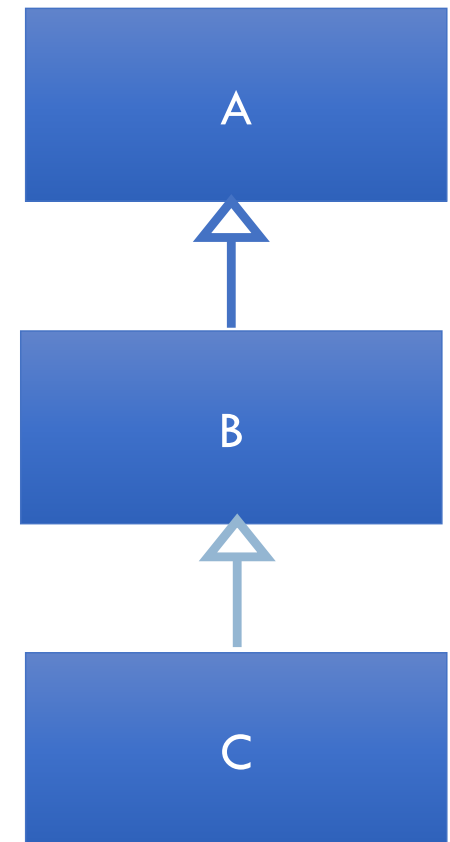
```
public class Main
{
    public static void main(String args[])
    {
        System.out.println("***** ");
        ContaPoupanca cp = new ContaPoupanca("Poupador",1000,2);
        System.out.println("Saldo inicial: " + cp.getSaldo());
        cp.atualiza(); // mais 20: 1020
        System.out.println("Saldo atual (depois do atualiza): " + cp.getSaldo());
        cp.deposita(100); // mais 100: 1120
        System.out.println("Saldo atual (depois do deposita): " + cp.getSaldo());
        cp.atualiza(); // mais 22.40: 1142.40
        System.out.println("Saldo atual (depois do atualiza): " + cp.getSaldo());
        cp.retira(100); // menos 100: 1042.40
        System.out.println("Saldo atual (depois do retira): " + cp.getSaldo());
    }
}
```

```
*****
Saldo inicial: 1000.0
Saldo atual (depois do atualiza): 1020.0
Saldo atual (depois do deposita): 1120.0
Saldo atual (depois do atualiza): 1142.4
Saldo atual (depois do retira): 1042.4
```

# Herança: Circularidade de Herança



- Circularidade de Herança
  - Dada a hierarquia ao lado:
  - Uma classe A não pode herdar de C devido à circularidade na herança de classes, ou seja, a classe C é subclasse e superclasse de A.





- O modificador de acesso `protected` oferece uma visibilidade intermediária entre `public` e `private`.
  - As entidades (métodos, atributos e classes internas) de uma superclasse com acesso `protected` podem ser acessados pela superclasse, subclasses e classes do mesmo pacote que a superclasse ( $\leq$  ruim)
  - Uma subclasse mantém os modificadores de acesso das entidades da superclasse
  - Atributos `protected` tem um acesso mais eficiente nas subclasses por não precisarem de métodos de acesso. No entanto, na maior parte dos casos, é recomendável o uso de atributos `private`, promovendo as boas práticas de encapsulamento que tornam o código mais fácil de manter, modificar e depurar.



- Redução de visibilidade
  - Não é possível reduzir a visibilidade de um método em uma sobrescrita
    - Um método definido como public na superclasse só pode ser sobrescrito como public
    - A promoção da visibilidade porém, pode ocorrer
      - Um método definido como protected em uma superclasse pode ser sobrescrito como public





- A classe Object é superclasse de todas as classes, exceto dela mesma, dado que a classe Object não é subclasse de ninguém
  - Ao criar uma nova classe, se não for especificado uma superclasse, implicitamente a nova classe herdará da classe Object, o que equivale a incluir `extends Object` na declaração da nova classe.
  - Podemos sobrescrever métodos da classe Object:
    - `clone`: Esse método realiza uma cópia do objeto (Não é usado diretamente).
    - `equals`: Esse método compara se dois objetos são iguais.
    - `getClass`: Todo objeto em Java tem acesso à própria classe a que pertence (propriedade conhecida como reflexão).
    - `hashCode`: são valores `int` que representam uma chave do objeto,
    - `toString`: Esse método retorna uma representação `String` de um objeto.

# Herança: classe Object



```
import java.lang.CloneNotSupportedException;

class Teste implements Cloneable {
    int t;

    public Teste(int n) {
        n = t;
    }

    public static void main(String[] args) throws CloneNotSupportedException {
        Teste t = new Teste(10);
        System.out.println(t);
        Teste t1 = (Teste)t.clone();
        System.out.println(t1);
        System.out.println(t1.equals(t));
    }
}
```

```
Teste@eb42cbf
Teste@56e5b723
false
```





- A palavra final também pode ser usada, em diferentes circunstâncias:
  - Quando usada na definição de uma variável, significa que a variável não pode assumir outro valor, tornando-se uma constante.
  - Quando usada na definição de um método, significa que o método não poderá ser sobrescrito.
  - Quando usada na definição de uma classe, significa que a classe não vai admitir herança.

# A palavra final



- Variáveis de instância:
  - Podem ser inicializadas, mesmo não sendo de classe. Entretanto, não pode ser mais alteradas.

```
public class TesteFinal {  
    public static void main(String args[]) {  
        MinhaClasse mc = new MinhaClasse();  
        System.out.println(mc);  
    }  
}  
  
public class MinhaClasse {  
  
    private final int teste;  
  
    public MinhaClasse() {  
        teste = 10;    // Ok, pois inicializamos o valor no construtor.  
    }  
  
    public void meuMetodo() {  
        teste++;    // Erro de compilação  
    }  
  
    public String toString() {  
        String out = teste + "\n";  
        return out;  
    }  
}
```

MinhaClasse.java:10: cannot assign a value to final variable teste  
 teste++; // Erro de compilação  
 ^  
1 error

```
public class MinhaClasse {  
  
    private final int teste=10; // Ok, mesmo sem ser estática  
  
    public MinhaClasse() {  
        //teste = 10;    // Ok, pois inicializamos o valor no construtor.  
    }  
  
    public void meuMetodo() {  
        //teste++;    // Erro de compilação  
    }  
  
    public String toString() {  
        String out = teste + "\n";  
        return out;  
    }  
}
```

# A palavra final



- Variáveis de instância:

```
public class TesteFinal {  
    public static void main(String args[]) {  
        MinhaClasse mc = new MinhaClasse();  
        System.out.println(mc);  
    }  
}
```

- Variáveis de classe: também podem ser declaradas como constantes.

```
public class MinhaClasse {  
    private final int teste=10; // Ok, mesmo sem ser estática  
  
    public MinhaClasse() {  
        teste = 20; // Problema, pois já foi inicializada  
    }  
  
    public void meuMetodo() {  
        //teste++; // Erro de compilação  
    }  
  
    public String toString() {  
        String out = teste + "\n";  
        return out;  
    }  
}
```

# A palavra final



- Métodos de instância:

```
public class MinhaClasse {  
    public final void meuMetodo() {}  
}  
  
public class MinhaNovaClasse extends MinhaClasse {  
    public void meuMetodo() {} // Não é permitido sobrescrever o método  
}
```

# A palavra final



- Classes:

```
public final class MinhaClasse {}  
  
// Isso não é permitido  
public class MinhaNovaClasse extends MinhaClasse {}  
.
```



# Relacionamentos



- Relacionamentos
- Tipos
- Multiplicidade
- Relacionamentos unidirecionais
- Relacionamentos bidirecionais

# Relacionamentos: Conceitos



- Representam relações entre objetos
  - Associação
  - Agregação
  - Composição
- Nem sempre a distinção é clara!



# Relacionamento: Associação



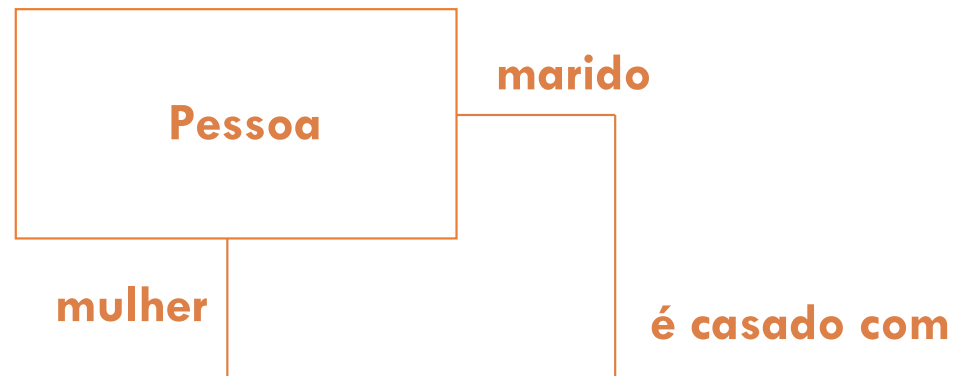
- Associação existente entre duas entidades
  - Cliente possui Pedidos. Um pedido é referente a um cliente.



# Relacionamento: Associação - tipos



- Associação unária

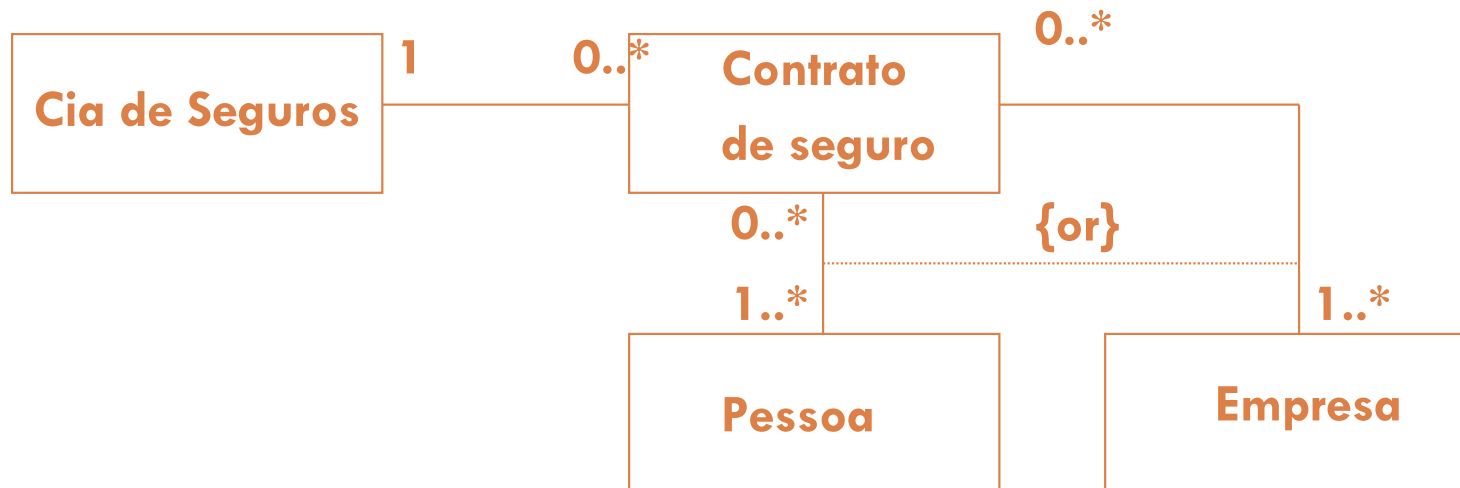


- Associação binária



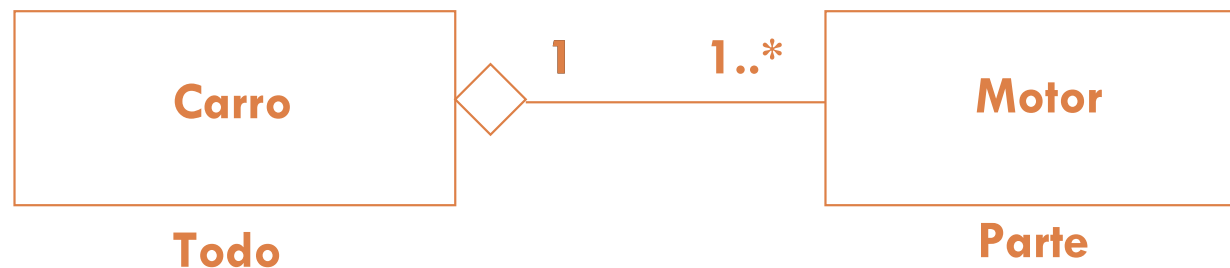


- Associação-ou: indica que somente uma das associações é válida no tempo



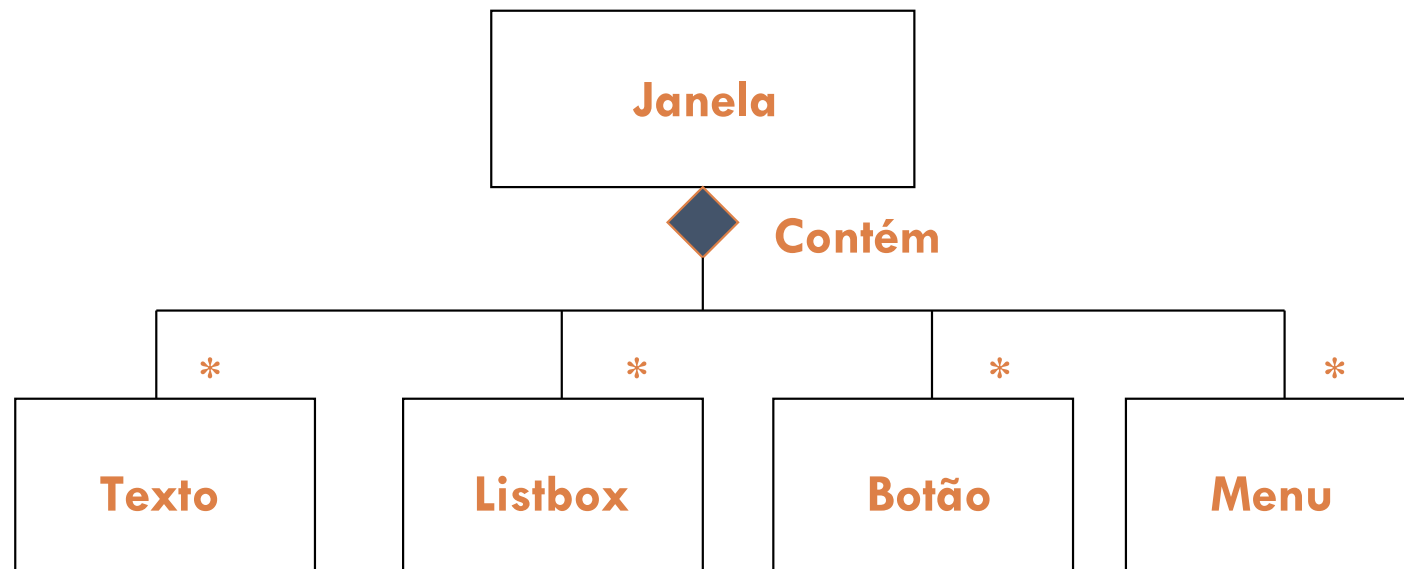


- Considerar algo maior pensando na relação todo-parte. Indica que um objeto parte é um atributo do objeto todo.
  - Um Carro C possui um Motor M. Se o carro for removido do sistema, o motor que estava sendo desenvolvido pode ser reaproveitado em outro carro.





- Também baseado na relação todo-parte. Neste caso, se o objeto maior for removido, as suas partes filhas serão removidas também.
  - Imagine o caso de um Carro C que possui uma Placa P registrada. Se o carro deixa de existir, a placa não tem mais utilidade dentro do sistema.



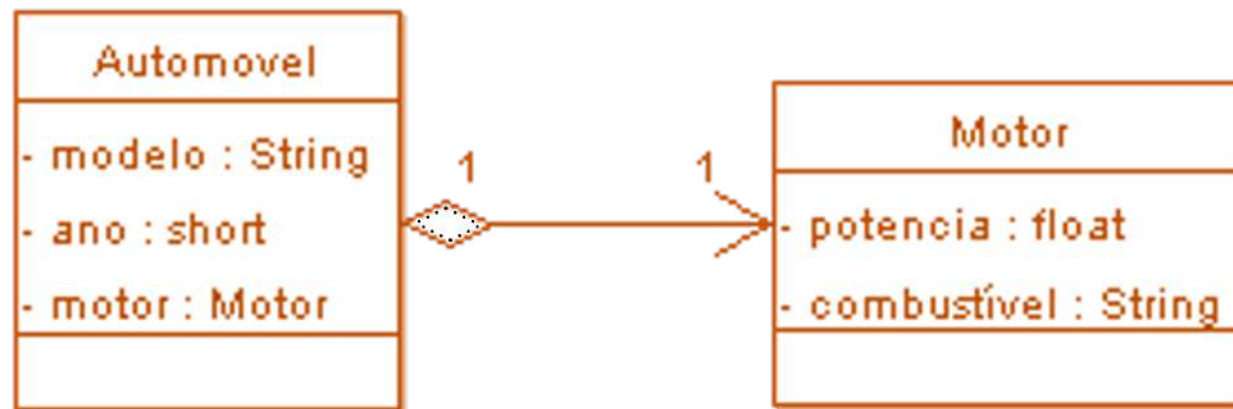


- Para finalidades deste curso:
  - Associação, agregação e composição (só) são diferentes semanticamente.
  - Na implementação em Java, associação, agregação e composição são similares!
  - Poderíamos diferenciar Agregação e Composição delegando, na Composição, a responsabilidade de criar as partes a classe forte do relacionamento.
- Diferenças na implementação:
  - Multiplicidade de associações: 1 para 1, 1 para \*, \* para \* (\* = muitos).
  - Implementação diferente para \* fixo e variável.

# Exemplo 1



- Um automóvel deve ter um motor instalado.
  - Uma instância de motor só pode ser associada a uma instância de automóvel a qualquer momento, e um automóvel só pode possuir um motor a cada instante: relação 1 para 1.



# Exemplo 1: Classe Motor



```
public class Motor
{
    private float potencia;
    private String combustivel;

    public Motor(float pot,String comb)
    {
        potencia = pot;
        combustivel = comb;
    }
    public float getPotencia()
    {
        return potencia;
    }
    public String getCombustivel()
    {
        return combustivel;
    }
    // @Override
    public String toString()
    {
        String out = " ";
        out += "Potência: " + getPotencia();
        out += ", Combustível:" + getCombustivel();
        return out;
    }
}
```

Classe de construção tradicional



# Exemplo 1: Classe Automovel



```
public class Automovel
{
    private String modelo;
    private short ano;
    private Motor motor;

    public Automovel(String mod, short a, Motor mot)
    {
        modelo = mod;
        ano = a;
        motor = mot;
    }

    public String getModelo()
    {
        return modelo;
    }

    public short getAno()
    {
        return ano;
    }
}
```

Indica a possibilidade de relacionar  
Automovel + Motor

Onde acontece o  
relacionamento

```
public Motor getMotor()
{
    return motor;
}

// @Override
public String toString()
{
    String out = " ";
    out += "Modelo: " + getModelo();
    out += ", Ano: " + getAno();
    out += motor;
    return out;
}
}
```

# Exemplo 1: Classe TesteAutomovel



```
public class TesteAutomovel
{
    public static void main(String[] args)
    {
        Motor motFusca = new Motor(47f,"gasolina");
        1 Automovel fusca66 = new Automovel("Fusca", (short)1966, motFusca);
        System.out.println(fusca66); // Invoca toString

        2 Automovel beetle2002 = new Automovel("New Beetle", (short)2002, new Motor(150f,"gasolina"));
        System.out.println(beetle2002); // Invoca toString
    }
}
```

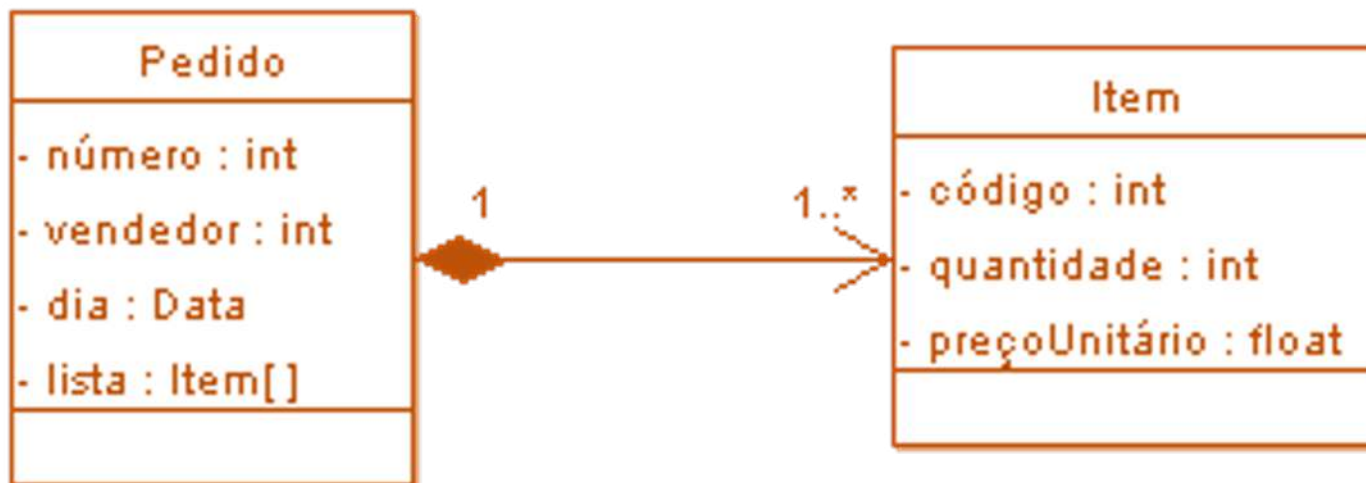
```
Modelo: Fusca, Ano: 1966 Potência: 47.0, Combustível:gasolina
Modelo: New Beetle, Ano: 2002 Potência: 150.0, Combustível:gasolina
```

Qual a diferença entre as duas instâncias (1 e 2)?

# Exemplo 2



- Um pedido de compras deve ter vários itens.
  - Não podemos ter pedidos vazios e um item deve pertencer a um único pedido: relação 1 para 1..\*.



# Exemplo 2: Relação 1 para 1..\*



- Como representar esta relação ?

```
public class Pedido
{
    private int vendedor;
    private Data dia;
    private Item lista1;
    private Item lista2;
    private Item lista3;
    private Item lista4;
    private Item lista5;
}
```

- Como //saber que instâncias estão em uso?
- E se precisar de mais instâncias de Item do que as declaradas?

# Exemplo 2: Classe Data



```
public class Data
{
    private byte dia, mes;
    private short ano;

    public Data(byte d, byte m, short a)
    {
        dia = d; mes = m; ano = a;
    }

    public byte getDia()
    {
        return dia;
    }

    public byte getMes()
    {
        return mes;
    }
}
```

Classe de construção tradicional

```
    public short getAno()
    {
        return ano;
    }

    // @Override
    public String toString()
    {
        String out = " ";
        out += "Data: " + getDia();
        out += "/" + getMes();
        out += "/" + getAno();
        return out;
    }
}
```

# Exemplo 2: Classe Item



```
public class Item
{
    private int codigo;           // propriedade codigo
    private int quantidade;       // propriedade quantidade em estoque
    private float precoUnitario;  // propriedade preço por unidade

    public Item(int cod, int quant, float preco) {
        codigo = cod;
        quantidade = quant;
        precoUnitario = preco;
    }

    public int getCodigo() {
        return codigo;
    }

    public int getQuantidade() {
        return quantidade;
    }

    public float getPreco() {
        return precoUnitario;
    }

    /* @Override */
    public String toString() {
        String out = " ";
        out += "Item: \n Código: " + getCodigo();
        out += ", " + getQuantidade() + " unidades a R$" + getPreco() + " cada.";
        return out;
    }

    public float custoTotal() {
        return getQuantidade()* getPreco();
    }
}
```

Classe de construção tradicional



# Exemplo 2: Classe Pedido



```
import java.util.ArrayList;

public class Pedido
{
    private int numero;
    private int vendedor;
    private Data dia;
    ArrayList<Item> lista;

    public Pedido(int n,int v,Data d)
    {
        numero = n;
        vendedor = v;
        dia = d;
        lista = new ArrayList<Item>();
    }

    public int getNumero()
    {
        return numero;
    }

    public int getVendedor()
    {
        return vendedor;
    }
}
```

Classe que concentra os dois relacionamentos:  
Pedido -> Data  
Pedido -> Item

Pedido -> Data

Pedido -> Item (usando ArrayList)

Cria o ArrayList para armazenar cada Item do Pedido

# Exemplo 2: Classe Pedido



```
public Data getData()
{
    return dia;
}
public void adicionaItem(Item i)
{
    lista.add(i);
}
public float calculaTotal()
{
    float total = 0f;
    for(int i=0; i<lista.size(); i++)
    {
        Item umItem = lista.get(i);
        total = total + umItem.custoTotal();
    }
    return total;
}
// @Override
public String toString()
{
    String out = "";
    out += "Pedido # " + getNumero();
    out += " do vendedor " + getVendedor() + " " + dia;
    out += "\nItens:\n";
    for(int i=0; i<lista.size(); i++)
    {
        Item umItem = lista.get(i);
        out += " * " + umItem + "\n";
    }
    out += "Total do pedido: " + calculaTotal();
    return out;
}
}
```

Adiciona novo Item ao Pedido

Verifica cada Item do Pedido

Monta a String com todos os dados do Pedido



# Exemplo 2: Classe TestePedido



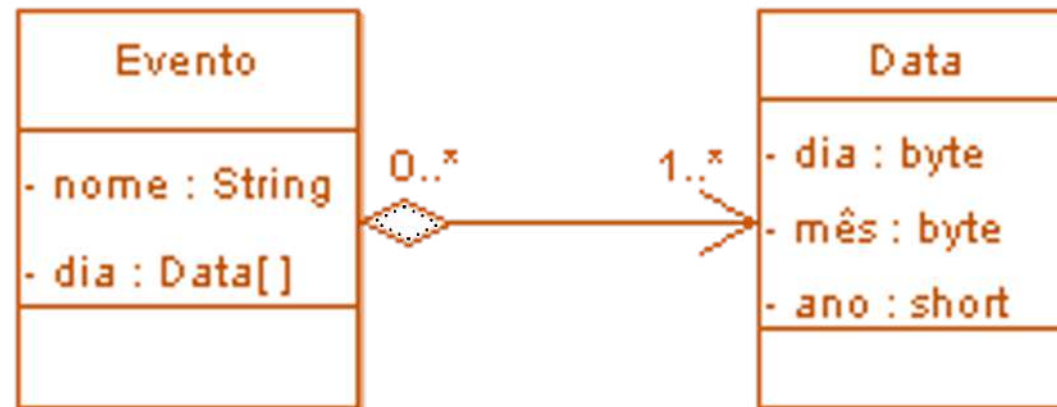
```
public class TestePedido {  
    public static void main(String[] args) {  
  
        /* Cria uma nova data */  
        Data hoje = new Data((byte)19, (byte)9, (short)2016);  
        /* Indica a data como sendo a data do pedido */  
        Pedido p = new Pedido(1, 3, hoje);  
  
        /* Cria 4 objetos do tipo Item */  
        Item i1 = new Item(1215, 10, 9.45f);  
        Item i2 = new Item(1217, 1, 21.00f);  
        Item i3 = new Item(1223, 1, 22.05f);  
        Item i4 = new Item(1249, 3, 50.95f);  
  
        /* Adiciona os itens ao pedido */  
        p.adicionaItem(i1);  
        p.adicionaItem(i2);  
        p.adicionaItem(i3);  
        p.adicionaItem(i4);  
  
        /* Invoca o método toString para o pedido */  
        System.out.println(p);  
    }  
}
```

```
Pedido # 1 do vendedor 3   Data: 19/9/2016  
Itens:  
  * Item:  
    Código: 1215, 10 unidades a R$9.45 cada.  
  * Item:  
    Código: 1217, 1 unidades a R$21.0 cada.  
  * Item:  
    Código: 1223, 1 unidades a R$22.05 cada.  
  * Item:  
    Código: 1249, 3 unidades a R$50.95 cada.  
Total do pedido: 290.40002
```

# Exemplo 3



- Um evento ocorre em uma determinada data.
  - Uma instância de data pode ser usada por várias instâncias de evento ou não usada por nenhuma. Um evento pode ter mais de uma data: relação 0..\* para 1..\*.



# Exemplo 3: Classe Evento



```
import java.util.ArrayList;

public class Evento
{
    private String nome;
    private ArrayList<Data> dia;

    public Evento(String n, Data umDia)
    {
        nome = n;
        dia = new ArrayList<Data>();
        dia.add(umDia);
    }

    public String getNome()
    {
        return nome;
    }

    public void marcaDiaAdicional(Data d)
    {
        dia.add(d);
    }

    // @Override
    public String toString()
    {
        String out = "";
        out += "Evento # " + getNome() + " ocorrerá nos dias:\n";
        for(int i=0; i<dia.size(); i++)
        {
            Data umDia = dia.get(i);
            out += " * " + umDia + "\n";
        }
        return out;
    }
}
```

→ Cria o ArrayList para armazenar os dias já com uma data inicial (obrigatório)

→ Adiciona nova data ao evento

# Exemplo 3: Classe TesteEvento



```
public class TesteEvento
{
    public static void main(String[] args)
    {
        Evento encontro = new Evento("MDCVI Encontro de Pessoas que Realmente "+
            "Usam Todos os Diagramas de UML",
            new Data((byte)1,(byte)4,(short)2016));
        encontro.marcaDiaAdicional(new Data((byte)2,(byte)4,(short)2016));
        encontro.marcaDiaAdicional(new Data((byte)3,(byte)4,(short)2016));
        encontro.marcaDiaAdicional(new Data((byte)4,(byte)4,(short)2016));
        System.out.println(encontro);
    }
}
```

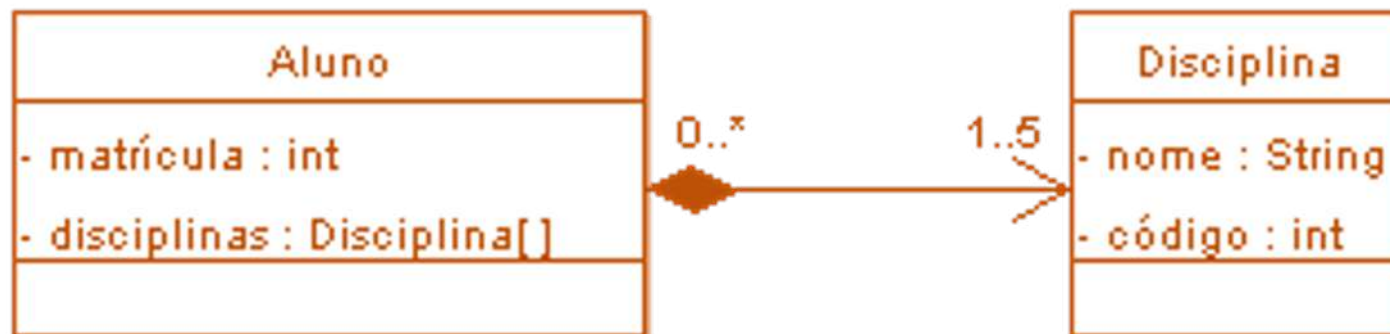
Evento # MDCVI Encontro de Pessoas que Realmente Usam Todos os Diagramas de UML ocorrerá nos dias:

- \* Data: 1/4/2016
- \* Data: 2/4/2016
- \* Data: 3/4/2016
- \* Data: 4/4/2016

# Exemplo 4



- Um aluno está matriculado em algumas disciplinas.
  - Um aluno pode estar matriculado em 1, 2, 3, 4 ou 5 disciplinas. Cada disciplina pode ter 0 ou mais alunos matriculados: relação 0..\* para 1..5.





# Exemplo 4: Classe Disciplina



```
public class Disciplina
{
    private String nome;
    private int codigo;

    public Disciplina(String n,int c)
    {
        nome = n; codigo = c;
    }
    public String getNome()
    {
        return nome;
    }
    public int getCodigo()
    {
        return codigo;
    }
    // @Override
    public String toString()
    {
        String out = "";
        out += "Disciplina código "+ getCodigo() + ": " +getNome();
        return out;
    }
}
```

# Exemplo 4: Classe Aluno



```
public class Aluno
{
    private int matricula;
    private ArrayList<Disciplina> disciplinas;

    public Aluno(int m)
    {
        matricula = m;
        disciplinas = new ArrayList<Disciplina>();
    }

    public int getMatricula()
    {
        return matricula;
    }

    public void matricula(Disciplina d)
    {
        if (disciplinas.size() < 5)
            disciplinas.add(d);
    }

    /* @Override */
    public String toString()
    {
        String out = "";
        out += "Aluno com matrícula # " + getMatricula() + " está matriculado nas disciplinas:\n";
        for(int i=0; i<disciplinas.size(); i++) {
            Disciplina umaDisciplina = disciplinas.get(i);
            out += " * " + umaDisciplina + "\n";
        }
        return out;
    }
}
```

# Exemplo 4: Classe TesteAluno



```
public class TesteAluno {  
    public static void main(String[] args) {  
  
        Disciplina pp = new Disciplina("Programação em Prolog", 11001);  
        Disciplina pl = new Disciplina("Programação em Lisp", 11002);  
        Disciplina ia = new Disciplina("Inteligência Artificial", 11201);  
        Disciplina ln = new Disciplina("Lógica Nebulosa", 11205);  
        Disciplina ag = new Disciplina("Algoritmos Genéticos", 11760);  
  
        Aluno m = new Aluno(34030001);  
        m.matricula(pp); m.matricula(ia); m.matricula(ln);  
        System.out.println(m);  
  
        Aluno n = new Aluno(34030029);  
        n.matricula(pl); n.matricula(ln); n.matricula(ag);  
        System.out.println(n);  
    }  
}
```

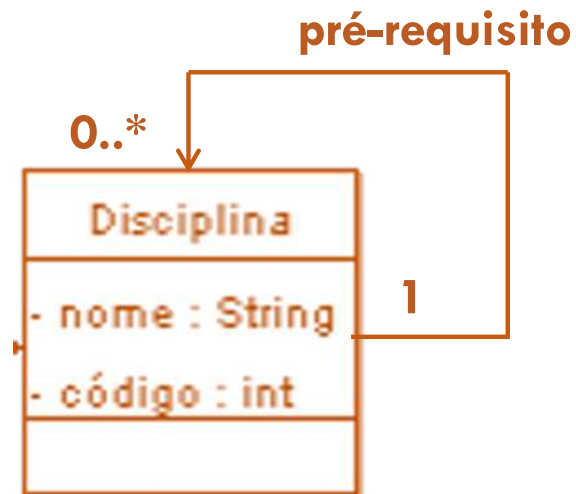
Aluno com matrícula # 34030001 está matriculado nas disciplinas:  
\* Disciplina código 11001: Programação em Prolog  
\* Disciplina código 11201: Inteligência Artificial  
\* Disciplina código 11205: Lógica Nebulosa

Aluno com matrícula # 34030029 está matriculado nas disciplinas:  
\* Disciplina código 11002: Programação em Lisp  
\* Disciplina código 11205: Lógica Nebulosa  
\* Disciplina código 11760: Algoritmos Genéticos





- Considere o exemplo de uma disciplina que tem várias disciplinas como pré-requisito.



# Relacionamento Unário



```
import java.util.ArrayList;

public class Disciplina {
    private String codigo;
    private String nome;
    private ArrayList<Disciplina> preRequisitos;

    public Disciplina(String nome, String codigo) {
        this(); // chamada para o construtor sem parâmetros
        setNome(nome);
        setCodigo(codigo);
    }

    public Disciplina() {
        preRequisitos = new ArrayList<Disciplina>();
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public void setCodigo(String codigo) {
        this.codigo = codigo;
    }

    public void addPreRequisitos(Disciplina d) {
        preRequisitos.add(d);
    }

    public void removePreRequisitos(Disciplina d) {
        preRequisitos.remove(d);
    }
}
```

```
@Override
public String toString() {
    String out = "";
    out += "Disciplina código " + codigo + ": " + nome;
    if (preRequisitos.size() > 0) {
        out += "\nPre-requisitos: ";
        for (Disciplina p:preRequisitos) {
            out += "* " + p;
        }
    }
    out += "\n";
    return out;
}

public static void main(String[] args) {

    Disciplina d1 = new Disciplina("Cálculo 1","11001");
    Disciplina d2 = new Disciplina("Cálculo 2","11002");
    Disciplina d3 = new Disciplina("Cálculo 3","11003");

    d2.addPreRequisitos(d1);
    d3.addPreRequisitos(d2);
    System.out.println(d1);
    System.out.println(d2);
    System.out.println(d3);
}
```

Disciplina código 11001: Cálculo 1

Disciplina código 11002: Cálculo 2

Pre-requisitos: \* Disciplina código 11001: Cálculo 1

Disciplina código 11003: Cálculo 3

Pre-requisitos: \* Disciplina código 11002: Cálculo 2

Pre-requisitos: \* Disciplina código 11001: Cálculo 1



- Unidirecional:
  - Apenas um das classes contém a referência para a outra
- Bi-direcional
  - Em uma relação bidirecional, cada entidade tem um campo de relacionamento ou propriedade que se refere à outra entidade. Por exemplo, se Pedido sabe quais são as instâncias do Item e se Item sabe a que Pedido pertence, eles têm um relacionamento bidirecional.
  - Cuidado com relacionamentos bi-direcionais!
  - Eles devem ser evitados sempre que possível!
  - Só faz sentido se ambos os lados necessitarem de informação sobre o outro!
- Até o momento, tratamos todos como unidirecionais!

# Revisitando: Exemplo Automóvel-Motor (1-1)



```
public class Motor
{
    private float potencia;           // propriedade potência do motor
    private String combustivel;       // propriedade combustível do motor
    private Automovel auto;           // propriedade combustível do motor

    public Motor(float pot,String comb) {
        potencia = pot;
        combustivel = comb;
    }

    public float getPotencia() {
        return potencia;
    }

    public String getCombustivel() {
        return combustivel;
    }

    public void setAutomovel(Automovel auto) {
        this.auto = auto;
    }

    /* @Override */
    public String toString()
    {
        String out = " ";
        out += "Potência: " + getPotencia();
        out += ", Combustível:" + getCombustivel(); |
        return out;
    }
}

public class Automovel
{
    private String modelo;
    private short ano;
    private Motor motor;

    public Automovel(String mod, short a, Motor mot) {
        modelo = mod;
        ano = a;
        motor = mot;
        motor.setAutomovel(this);
    }

    public String getModelo() {
        return modelo;
    }

    public short getAno() {
        return ano;
    }

    public Motor getMotor() {
        return motor;
    }

    /* @Override */
    public String toString() {
        String out = " ";
        out += "Modelo: " + getModelo();
        out += ", Ano: " + getAno();
        out += motor;
        return out;
    }
}
```



# Revisitando: Exemplo Automóvel-Motor



```
public class TesteAutomovel {  
    public static void main(String[] args) {  
  
        /* Cria um novo objeto motor */  
        Motor motFusca = new Motor(47f,"gasolina");  
        /* Cria um novo objeto carro, associando ao carro o motor anteriormente criado */  
        Automovel fusca66 = new Automovel("Fusca", (short)1966, motFusca);  
        /* Invoca o método toString do automóvel */  
        System.out.println(fusca66);  
  
        /* Cria um novo objeto carro, associando ao carro o objeto motor implicitamente criado */  
        /* Qual a principal diferença entre estas duas declarações? */  
        Automovel beetle2002 = new Automovel("New Beetle", (short)2002, new Motor(150f,"gasolina"));  
        /* Invoca o método toString do automóvel */  
        System.out.println(beetle2002);  
    }  
}
```

Modelo: Fusca, Ano: 1966 Potência: 47.0, Combustível:gasolina

Modelo: New Beetle, Ano: 2002 Potência: 150.0, Combustível:gasolina

- ```

/* @Override */
public String toString()
{
    String out = " ";
    out += "Potência: " + getPotencia();
    out += ", Combustível:" + getCombustivel();
    out += auto;
    return out;
}

```

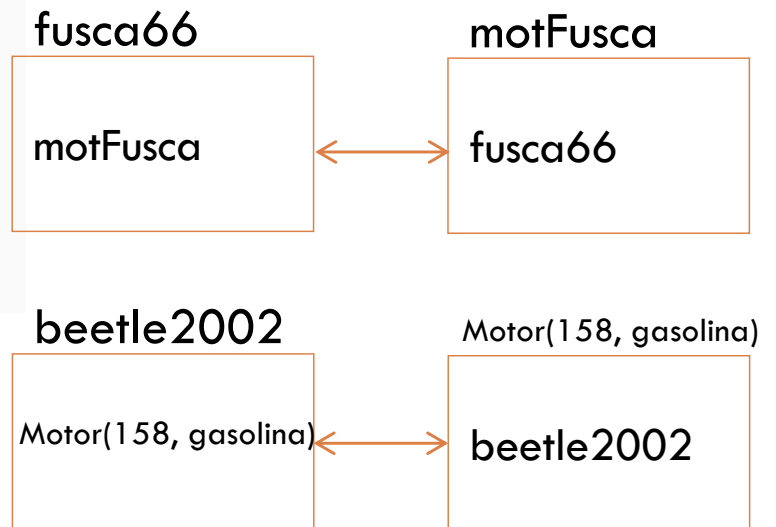
158

# Revisitando: Exemplo Automóvel-Motor



- E se gerarmos inconsistência?

```
public class TesteAutomovel {  
    public static void main(String[] args) {  
  
        /* Cria um novo objeto motor */  
        Motor motFusca = new Motor(47f,"gasolina");  
        /* Cria um novo objeto carro, associando ao carro o motor anteriormente criado */  
        Automovel fusca66 = new Automovel("Fusca", (short)1966, motFusca);  
        /* Invoca o método toString do automóvel */  
        System.out.println("Dados do fusca66");  
        System.out.println(fusca66);  
        /* Cria um novo objeto carro, associando ao carro o objeto motor implicitamente criado */  
        /* Qual a principal diferença entre estas duas declarações? */  
        Automovel beetle2002 = new Automovel("New Beetle", (short)2002, new Motor(150f,"gasolina"));  
        /* Invoca o método toString do automóvel */  
        System.out.println("Dados do beetle2002");  
        System.out.println(beetle2002);  
    }  
}
```



```
Dados do fusca66  
Modelo: Fusca, Ano: 1966 Potência: 47.0, Combustível:gasolina  
Dados do beetle2002  
Modelo: New Beetle, Ano: 2002 Potência: 150.0, Combustível:gasolina
```

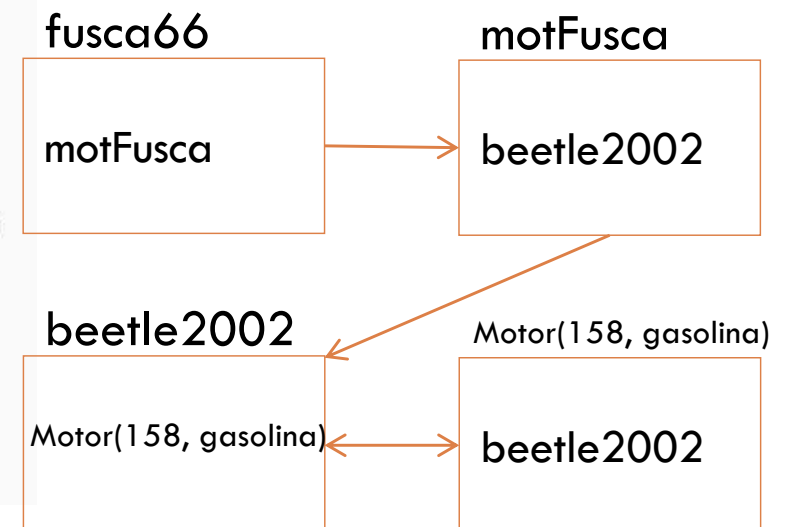
# Revisitando: Exemplo Automóvel-Motor



- E se gerarmos inconsistência?

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

        /* Cria um novo objeto motor */
        Motor motFusca = new Motor(47f,"gasolina");
        /* Cria um novo objeto carro, associando ao carro o motor anteriormente criado */
        Automovel fusca66 = new Automovel("Fusca", (short)1966, motFusca);
        /* Invoca o método toString do automóvel */
        System.out.println("Dados do fusca66");
        System.out.println(fusca66);
        /* Cria um novo objeto carro, associando ao carro o objeto motor implicitamente criado */
        /* Qual a principal diferença entre estas duas declarações? */
        Automovel beetle2002 = new Automovel("New Beetle", (short)2002, new Motor(150f,"gasolina"));
        /* Invoca o método toString do automóvel */
        System.out.println("Dados do beetle2002");
        System.out.println(beetle2002);
        motFusca.setAutomovel(beetle2002);
        System.out.println("Dados do fusca66");
        System.out.println(fusca66);
        System.out.println("Dados do carro relacionado a motFusca");
        System.out.println(motFusca.getAutomovel());
    }
}
```



```
Dados do fusca66
Modelo: Fusca, Ano: 1966 Potência: 47.0, Combustível:gasolina
Dados do beetle2002
Modelo: New Beetle, Ano: 2002 Potência: 150.0, Combustível:gasolina
Dados do fusca66
Modelo: Fusca, Ano: 1966 Potência: 47.0, Combustível:gasolina
Dados do carro relacionado a motFusca
Modelo: New Beetle, Ano: 2002 Potência: 150.0, Combustível:gasolina
```



# Revisitando: Exemplo Aluno-Disciplina (\*-\*)



```
public class Disciplina
{
    private String nome;           // propriedade nome
    private int codigo;            // propriedade codigo
    private ArrayList<Aluno> alunos;

    public Disciplina(String n,int c) {
        nome = n; codigo = c;
        alunos = new ArrayList<Aluno>();
    }

    public String getNome() {
        return nome;
    }

    public int getCodigo() {
        return codigo;
    }

    public void addAluno(Aluno a) {
        alunos.add(a);
    }

    public String listaAlunos() {
        String out = "";
        out += "Alunos matriculados na disciplina" + nome + " :\n";
        for(int i=0; i<alunos.size(); i++) {
            out += " * " + (alunos.get(i)).getMatricula() + "\n";
        }
        return out;
    }

    /* @Override */
    public String toString() {
        String out = "";
        out += "Disciplina código " + getCodigo() + ": " + getNome();
        for(int i=0; i<alunos.size(); i++) {
            out += " * " + (alunos.get(i)) + "\n";
        }
        return out;
    }
}
```

```
public class Aluno
{
    private int matricula;
    private ArrayList<Disciplina> disciplinas;

    public Aluno(int m)
    {
        matricula = m;
        disciplinas = new ArrayList<Disciplina>();
    }

    public int getMatricula()
    {
        return matricula;
    }

    public void matricula(Disciplina d)
    {
        if (disciplinas.size() < 5) {
            disciplinas.add(d);
            d.addAluno(this);
        }
    }

    /* @Override */
    public String toString()
    {
        String out = "";
        out += "Aluno com matrícula # " + getMatricula() + " está matriculado nas disciplinas:\n";
        for(int i=0; i<disciplinas.size(); i++) {
            Disciplina umaDisciplina = disciplinas.get(i);
            out += " * " + umaDisciplina + "\n";
        }
        return out;
    }
}
```

# Revisitando: Exemplo Aluno-Disciplina



```
public class TesteAluno {  
    public static void main(String[] args) {  
  
        Disciplina pp = new Disciplina("Programação em Prolog", 11001);  
        Disciplina pl = new Disciplina("Programação em Lisp", 11002);  
        Disciplina ia = new Disciplina("Inteligência Artificial", 11201);  
        Disciplina ln = new Disciplina("Lógica Nebulosa", 11205);  
        Disciplina ag = new Disciplina("Algoritmos Genéticos", 11760);  
  
        Aluno m = new Aluno(34030001);  
        m.matricula(pp); m.matricula(ia); m.matricula(ln);  
        System.out.println(m);  
  
        Aluno n = new Aluno(34030029);  
        n.matricula(pl); n.matricula(ln); n.matricula(ag);  
        System.out.println(n);  
  
        System.out.println(pp.listaAlunos());  
        System.out.println(pl.listaAlunos());  
    }  
}
```

Note que agora não faz mais sentido listar os dados dos alunos na disciplina... ou teremos dados replicados sempre!

```
Aluno com matrícula # 34030001 está matriculado nas disciplinas:  
* Disciplina código 11001: Programação em Prolog * 34030001  
  
* Disciplina código 11201: Inteligência Artificial * 34030001  
  
* Disciplina código 11205: Lógica Nebulosa * 34030001
```

```
Aluno com matrícula # 34030029 está matriculado nas disciplinas:  
* Disciplina código 11002: Programação em Lisp * 34030029  
  
* Disciplina código 11205: Lógica Nebulosa * 34030001  
* 34030029  
  
* Disciplina código 11760: Algoritmos Genéticos * 34030029
```

```
Alunos matriculados na disciplina Programação em Prolog:  
* 34030001
```

```
Alunos matriculados na disciplina Programação em Lisp:  
* 34030029
```

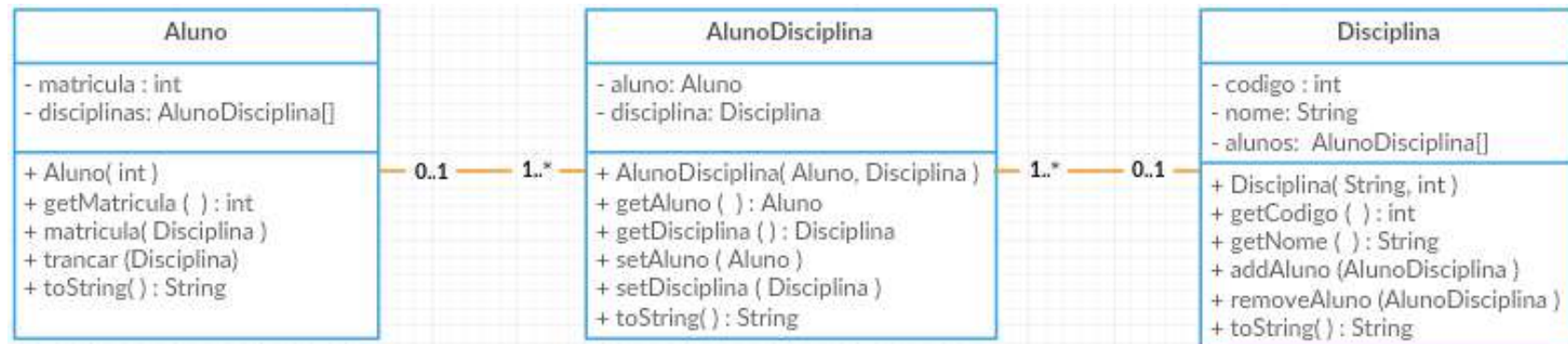


- Usando uma classe Associativa
  - Quando a associação entre duas classes possui multiplicidade muitos (\*) em ambas as extremidades, uma forma de se manter esta estrutura ocorre por meio de uma classe associativa.
  - Esta classe é necessária para armazenar atributos transmitidos pela associação de ambas as classes, podendo a mesma, inclusive, possuir atributos próprios.

# Revisitando: Exemplo Aluno-Disciplina



- Usando uma classe Associativa
  - 1 aluno pode cursar várias disciplinas



# Revisitando: Exemplo Aluno-Disciplina



```
public class Aluno
{
    private int matricula;
    private ArrayList<AlunoDisciplina> disciplinas;

    public Aluno(int m)
    {
        matricula = m;
        disciplinas = new ArrayList<AlunoDisciplina>();
    }

    public int getMatricula()
    {
        return matricula;
    }

    public void matricula(Disciplina d)
    {
        disciplinas.add(new AlunoDisciplina(this,d));
        d.addAluno(disciplinas.get(disciplinas.size()-1));
    }

    public void trancar(Disciplina d)
    {
        for (int i=0; i<disciplinas.size(); ++i) {
            // Achei a inscrição do aluno na disciplina
            if ((disciplinas.get(i)).getDisciplina() == d) {
                d.removeAluno((disciplinas.get(i)));
                disciplinas.remove(i);
            }
        }
    }

    /* @Override */
    public String toString()
    {
        String out = "";
        out += "Aluno com matrícula # " + getMatricula() + " está matriculado nas disciplinas:\n";
        for(int i=0; i<disciplinas.size(); i++) {
            out += " * " + ((disciplinas.get(i)).getDisciplina()).getNome() + "\n";
        }
        return out;
    }
}
```

Optamos por criar a inscrição do aluno da disciplina no momento em que o aluno solicita a matrícula na mesma!

Seremos os responsáveis por remover o link entre o aluno e a disciplina



# Revisitando: Exemplo Aluno-Disciplina



```
import java.util.ArrayList;

public class Disciplina
{
    private String nome;        // propriedade nome
    private int codigo;         // propriedade codigo
    private ArrayList<AlunoDisciplina> alunos;

    public Disciplina(String n,int c) {
        nome = n; codigo = c;
        alunos = new ArrayList<AlunoDisciplina>();
    }

    public String getNome() {
        return nome;
    }

    public int getCodigo() {
        return codigo;
    }

    public void addAluno(AlunoDisciplina a) {
        alunos.add(a);
    }

    public void removeAluno(AlunoDisciplina a) {
        alunos.remove(a);
    }

    /* @Override */
    public String toString() {
        String out = "";
        out += "Alunos matriculados na disciplina código " + getCodigo() + " " + getNome() + ":\n";
        for(int i=0; i<alunos.size(); i++) {
            out += " * " + ((alunos.get(i)).getAluno()).getMatricula() + "\n";
        }
        return out;
    }
}
```

# Revisitando: Exemplo Aluno-Disciplina



```
public class AlunoDisciplina
{
    private Aluno aluno;
    private Disciplina disciplina;

    public AlunoDisciplina(Aluno aluno, Disciplina disciplina) {
        this.aluno = aluno;
        this.disciplina = disciplina;
    }

    public Aluno getAluno() {
        return aluno;
    }

    public Disciplina getDisciplina() {
        return disciplina;
    }

    public void setAluno(Aluno aluno) {
        this.aluno = aluno;
    }

    public void setDisciplina(Disciplina disciplina) {
        this.disciplina = disciplina;
    }

    /* @Override */
    public String toString() {
        String out = "";
        out += "Aluno " + aluno.getMatricula() + " está matriculado na disciplina " + disciplina.getNome() + "\n";
        return out;
    }
}
```

## Classe associativa!

Ela poderia, inclusive, ter dados próprios como as notas do aluno naquela disciplina!

# Revisitando: Exemplo Aluno-Disciplina



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

        Disciplina pp = new Disciplina("Programação em Prolog",11001);
        Disciplina pl = new Disciplina("Programação em Lisp",11002);
        Disciplina ia = new Disciplina("Inteligência Artificial",11201);
        Disciplina ln = new Disciplina("Lógica Nebulosa",11205);
        Disciplina ag = new Disciplina("Algoritmos Genéticos",11760);

        Aluno m = new Aluno(34030001);
        m.matricula(pp);
        m.matricula(ia);
        m.matricula(ln);
        System.out.println(m);

        Aluno n = new Aluno(34030029);
        n.matricula(pl);
        n.matricula(ln);
        n.matricula(ag);
        System.out.println(n);

        Aluno o = new Aluno(34030088);
        o.matricula(pp);
        o.matricula(ia);
        o.matricula(ag);
        System.out.println(o);

        System.out.println(pp);
        System.out.println(pl);
        System.out.println(ia);
        System.out.println(ln);
        System.out.println(ag);

        System.out.println("***** Depois do trancamento *****\n");
        o.trancar(pp);
        System.out.println(o);
        System.out.println(pp);
    }
}
```

Aluno com matrícula # 34030001 está matriculado nas disciplinas:

- \* Programação em Prolog
- \* Inteligência Artificial
- \* Lógica Nebulosa

Aluno com matrícula # 34030029 está matriculado nas disciplinas:

- \* Programação em Lisp
- \* Lógica Nebulosa
- \* Algoritmos Genéticos

Aluno com matrícula # 34030088 está matriculado nas disciplinas:

- \* Programação em Prolog
- \* Inteligência Artificial
- \* Algoritmos Genéticos

Alunos matriculados na disciplina código 11001 Programação em Prolog:

- \* 34030001
- \* 34030088

Alunos matriculados na disciplina código 11002 Programação em Lisp:

- \* 34030029

Alunos matriculados na disciplina código 11201 Inteligência Artificial:

- \* 34030001
- \* 34030088

Alunos matriculados na disciplina código 11205 Lógica Nebulosa:

- \* 34030001
- \* 34030029

Alunos matriculados na disciplina código 11760 Algoritmos Genéticos:

- \* 34030029
- \* 34030088

\*\*\*\*\* Depois do trancamento \*\*\*\*\*

Aluno com matrícula # 34030088 está matriculado nas disciplinas:

- \* Inteligência Artificial
- \* Algoritmos Genéticos

Alunos matriculados na disciplina código 11001 Programação em Prolog:

- \* 34030001





- Object-Oriented Programming with Java: An Introduction, David J. Barnes; Prentice Hall, 2000.
- ALBAHARI, J. C# 10 in a Nutshell: The Definitive Reference. O'Reilly Media, 2022.
- BUDD, T. An Introduction to Object-Oriented Programming 3rd Edition. Addison-Wesley. 2001.