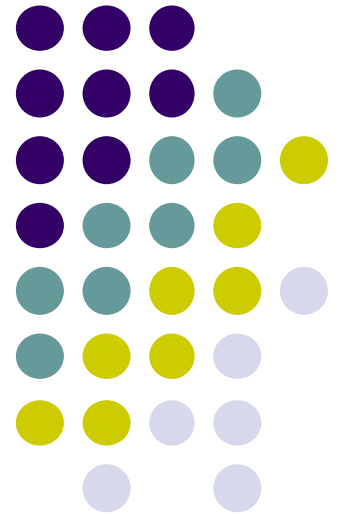


# Programação OO com Java

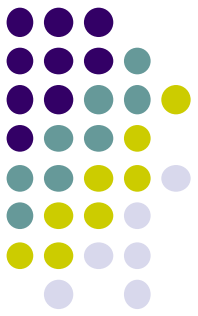
---

MAT A55

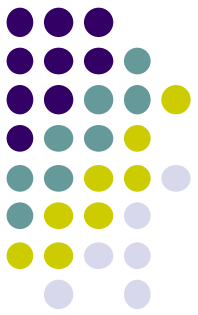
Professora Rita Suzana



# Linguagem Java



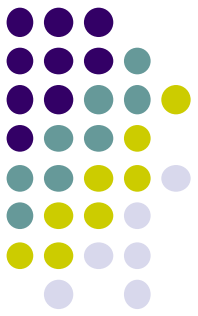
- Linguagens de alto nível orientadas a objetos (programação OO):
  - Iniciou-se com a linguagem SmallTalk-80 (década de 80)
  - Uma linguagem puramente OO: tudo é objeto
  - Criada no laboratório da Xerox
  - Uma nova maneira de raciocínio para criação de programas
  - Classes, objetos, atributos, métodos, mensagens etc.
  - Um novo paradigma de programação
    - Programação estruturada/imperativa => Programação OO
  - **Reuso**: grande promessa da POO
- Na década de 90 surge a linguagem JAVA



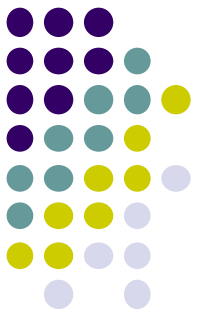
# Linguagem Java

- Iniciada por um grupo de engenheiros da **SUN**
  - Patrick Naughton, Sun Fellow e James Gosling em 1991.
- Inicialmente pretendia-se uma linguagem simples para aparelhos eletrodomésticos (TVs, VCRs, torradeiras, geladeiras, etc).
  - Rápida;
  - eficiente; e
  - portátil para diversas arquiteturas (diversos fabricantes e tipos de aparelhos distintos).
  - Utilizou-se então o conceito de máquinas virtuais
    - JVM (Java virtual machine)

# Linguagem Java

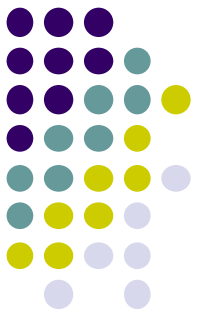


- Engenheiros da Sun evoluíram o *Green*
  - Com influências do C++ criaram a linguagem *Oak*
  - Houve dificuldade na venda da tecnologia
  - Até 1994, os engenheiros não conseguiram comercializar a tecnologia.
  - O avanço da internet foi a luz no fim do túnel.
  - Em 1995, os engs. lançaram um browser em Java para mostrar o poder da linguagem: o HotJava.
  - Além disso, permitiram a execução de código Java dentro das páginas no lado cliente (browser): o que chamamos de *applets*



# Linguagem Java

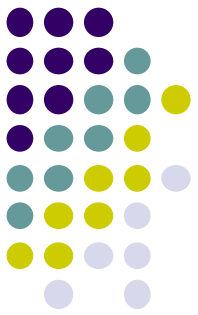
- Finalmente em 1996 a Sun lançou a primeira versão da linguagem Java – Java 1.0
  - Ainda era uma linguagem imatura.
  - havia poucos recursos.
- Em 1998, na JavaOne Conference, a SUN disponibilizou o Java 1.2
- Em 2004 lançou a versão 5
- Final de 2006 saiu a versão 6.



# Linguagem Java

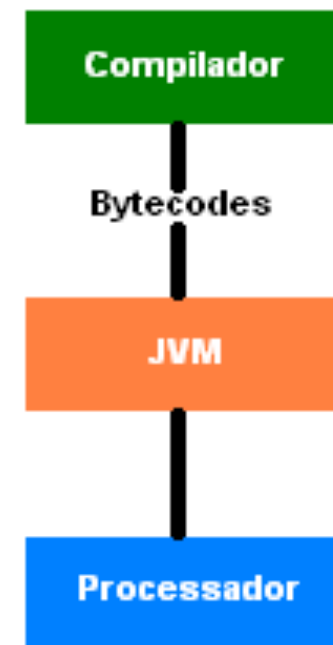
- Características
  - Simples
  - Orientada a Objeto
  - Portável e independente de plataforma
  - Interpretada
  - Diferente de C++ (combinação de objetos e funções)
    - Java : cada elemento é um objeto
    - Aproveita alguns conceitos e sintaxe de C++

# Linguagem Java

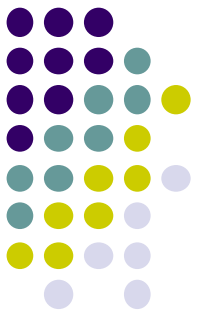


- Características

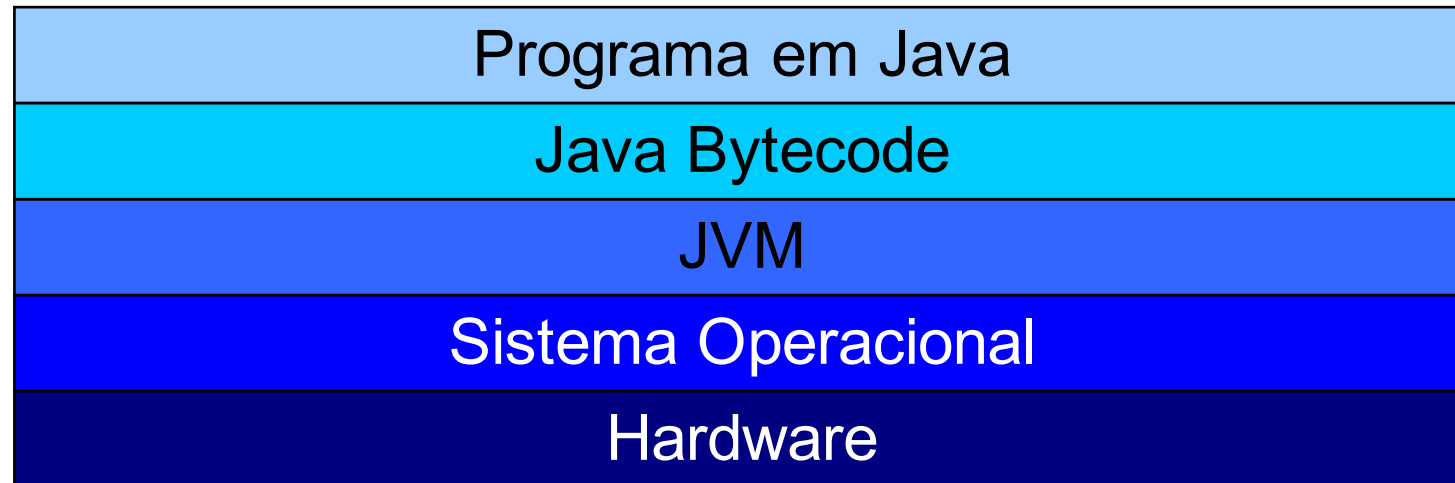
- Interpretada
  - Compilador gera bytecodes
  - Bytecodes se baseiam em um Máquina Virtual Java (MVJ ou JVM- Java Virtual Machine)
    - Máquina implementada em software
    - Uma camada de abstração entre o seu código e o código de máquina
    - Semelhante a uma CPU
    - Máquina que compreende os bytecodes
    - MVJ executa as chamadas de funções do SO.



# Linguagem Java

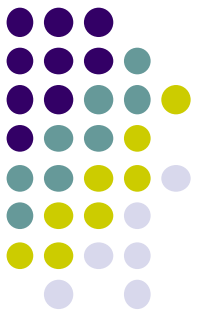


- Camadas de Abstração





# Linguagem Java



- A máquina virtual Java é um interpretador que transforma as instruções *bytecodes* em linguagem de máquina.
- Portabilidade
  - Existem implementações da JVM para Solaris, Windows, Apple, Unix e Linux

## Exemplo.java

```
public class Exemplo
{
    public static void main()
    {
        System.out.println("Primeiro programa!");
    }
}
```

## COMPILADOR

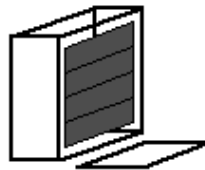
## Exemplo.class

```
???
// Source... ??
?|?|?... ??
...<Exemplo.java>...?
..|| ? || ?? ? ... ||?|
.....
BYTECODES
```

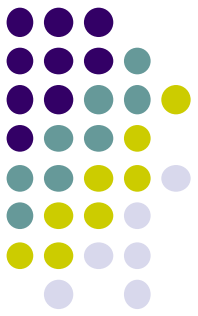
## Linguagem de Máquina

```
11110100100100000111000010101010
1001010011111000001010101010101
001110011001001010101010000001
1101010010101111100001110101011
100101001000101010100100101111
101010100101010010110001011010
101010101001010101011001101010
```














## INTERPRETADOR



# Programação OO com Java

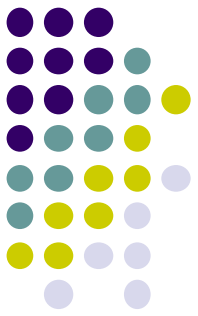


- Revisando aula passada...
  - Vimos os conceitos de: classes, objetos/instâncias, estado de objetos, atributos e métodos.

Avião	
	capacidadePassageiros
	modelo
	comprimento
	largura
	ano
	ultimaManutencao
	pesoMaxDecolagem
	decolar()
	pousar()
	embarcarPassageiros()
	desembarcarPassageiros()
	abastecer()
	realizarManutencao()

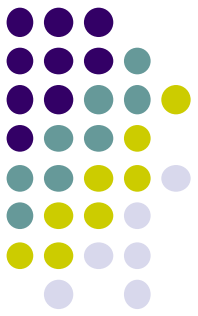
**Um objeto Avião e seu estado:**  
capacidadePassageiros = 200  
modelo = Airbus A320  
comprimento = 37,57 metros  
largura = 11,76 metros  
ano = 1999  
ultimaManutencao = 12/01/2008  
pesoMaxDecolagem = 77.000 kg

# Programação OO com Java



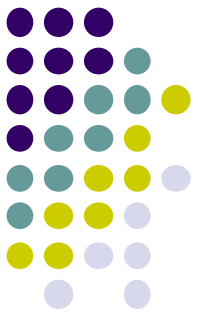
- Encapsulamento:
  - Combinação de dados e comportamentos em uma classe, escondendo do usuário do objeto os detalhes de implementação.
  - Dizemos que um objeto em um programa “encapsula” todo o estado e o comportamento, de modo que podemos tratar o objeto como uma coisa só.
  - É por isso que um programa em Java costuma ser formado por vários objetos em vez de apenas um.

# Programação OO com Java



- Encapsulamento (exemplos):
  - Impressora:
    - Não sabemos como a impressora faz para imprimir as páginas internamente. Uma série de operações são realizadas, mas apenas solicitamos a impressão e esperamos pelo resultado.
    - Não precisamos abrir a impressora e medir o nível de tinta do cartucho. Apenas solicitamos a informação do status do nível de tinta.
  - Celular:
    - Não sabemos o que o aparelho celular faz para se comunicar com a operadora e realizar as chamadas, apenas solicitamos a ligação através do número desejado. Os sub-passos estão encapsulados.
  - Conta Bancária
    - Não sabemos os passos operacionais que o banco realiza para registrar um saque. Apenas solicitamos e recebemos o dinheiro.

# Programação OO com Java



- Exercício:

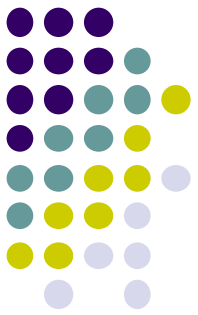
Em um sistema para locadora de filmes precisamos representar os filmes da locadora respondendo perguntas, tais como:

- Qual o ator principal do filme?
- Quem dirigiu o filme?
- Há quantos anos o filme foi lançado?
- Quantas cópias existem disponíveis para o filme?
- É um filme nacional?

Como seria representado um cliente da locadora?

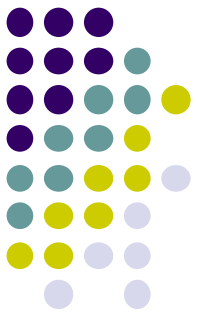
Como um cliente poderia realizar a locação/devolução de um filme?

# Programação OO com Java



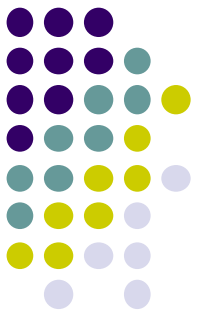
- Encapsulamento (continuação):
  - A manipulação dos atributos das classes deve ocorrer através de métodos acessores (*gets* e *sets*). Não se deve acessar os atributos diretamente fora da classe.
  - Comportamentos atribuídos à classe devem estar implementados através de métodos dentro da classe e jamais fora dela - Alta *coesão*.
  - Objetos se comportam como uma caixa-preta e se relacionam através de *troca de mensagens*.

# Programação OO com Java



- Como faríamos para instanciar um objeto a partir de uma classe?
- Quais valores teriam os atributos do objeto instanciado?
  - Utilizamos o conceito de **construtores**.
  - Construtores (ou métodos construtores) são responsáveis por instanciar objetos.
    - São métodos que podem possuir parâmetros de entrada e que sempre devolvem uma instância da classe como retorno.

# Programação OO com Java



- Introduzindo a sintaxe do Java:

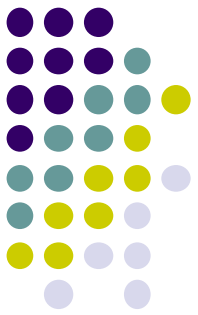
```
class NomeDaClasse  
{  
    corpo_da_classe...  
}
```

- No corpo da classe temos: atributos, métodos e construtores.

```
class NomeDaClasse  
{  
    atributos...  
  
    construtores...  
  
    métodos...  
}
```



# Programação OO com Java



## ● Introduzindo a sintaxe do Java:

```
class NomeDaClasse
{
    atributo1DaClasse;
    atributo2DaClasse;
    ...
    atributoNDaClasse;
}
```

Nome da classe e do construtor são iguais

Nos construtores não definimos parâmetros de retorno/saída.

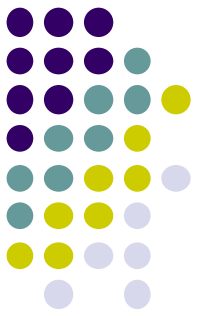
```
public NomeDaClasse(parametros_de_entrada_do_construtor)
{
    ...
    corpo do construtor
    ...
}
```

Construtores sempre retornam, implicitamente, uma instância da classe

```
public void umMetodoDaClasse(int c)
{
    ...
    corpo do método
    ...
}
```

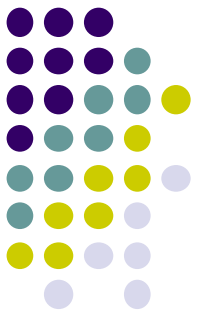
Métodos podem ter nenhum ou vários parâmetros de entrada. Porém, no máximo um parâmetro de saída

```
public int umOutroMetodoDaClasse()
{
    ...
    corpo do método
    ...
}
```



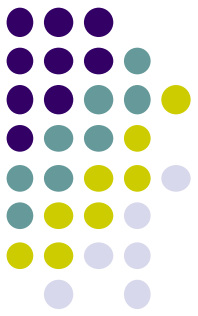
# Criando classes em Java

- Sintaxe básica
  - Uma classe em Java será declarada com a palavra-chave **class** seguida do nome da classe.
    - O nome não pode conter espaços
    - Deve começar com uma letra
    - Deve ser diferente das palavras reservadas
    - Caracteres maiúsculos e minúsculos são diferenciados
    - Conteúdo da classe limitado pelas chaves { }



# Criando classes em Java

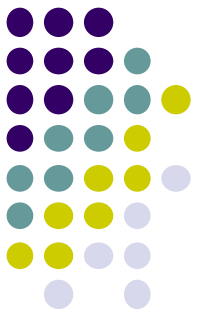
- Java provê tipos primitivos divididos em quatro grandes categorias:
  - *Inteiros* (nos discretos): **byte** (8 bits), **short** (16 bits), **int** (32 bits) e **long** (64 bits).
  - *Floating Point* (nos contínuos): **float** (32 bits) e **double** (64 bits)
  - *Character*: **char** (16 bits)
  - *Boolean*: **boolean**
- A classe `String` é usada para representar cadeias de caracteres.
  - Não são dados nativos, sendo instâncias da classe `String`



# Criando classes em Java

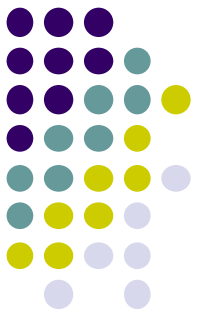
## Dados nativos em Java

- Valores numéricos podem ser comparados com operadores que retornam um valor do tipo boolean. Os operadores são:
  - < (menor)
  - > (maior)
  - <= (menor ou igual)
  - >= (maior ou igual)
  - == (igual)
  - != (diferente)



# Criando classes em Java

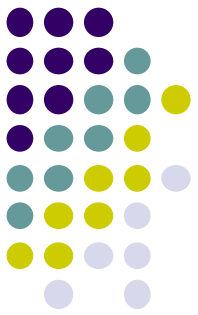
- Valores booleanos podem ser combinados com três operadores lógicos. Que são:
  - && (E lógico)
  - || (OU lógico)
  - ! (NÃO lógico)
- Operação com instâncias da classe String.
  - + (concatenação)
- Strings não podem ser comparados com os operadores >, <, ==.



# Criando classes em Java

- Campos de classes em Java
  - Os campos de classes em Java devem ser declarados dentro do corpo da classe.
    - Cada campo deve ser representado por um determinado tipo de dado.
    - Em linguagens POO, é possível declarar campos como referências a instâncias de outras classes já existentes.

# Criando classes em Java



- Exemplo:

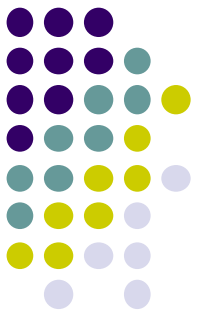
```
class Casa
{
    int numero;
    String cor;
    int qtdQuartos;

    public Casa(int num, String c, int qtdQ)
    {
        inicializa os atributos da casa de acordo com os parâmetros de entrada do construtor...
    }

    public void pintarCasa(String c)
    {
        mudando a cor da casa...
    }

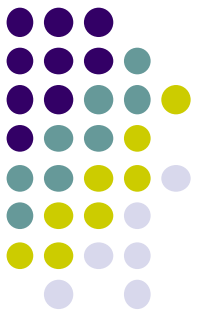
    public String obterInformacoesCasa()
    {
        retorna um texto com as informações contidas nos atributos da casa...
    }
}
```

# Programação OO com Java



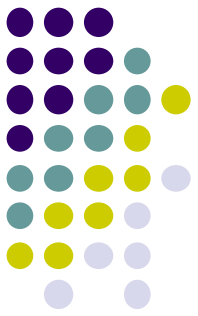
- Exercício:
  - Escrever, de acordo com a sintaxe apresentada, a classe Filme do exercício da locadora.
  - Atributos
    - Titulo, ator, lancamento, qtd\_copias
  - Métodos
    - Construtor
    - Locar ()
    - Devolver( ) só considerando o filme

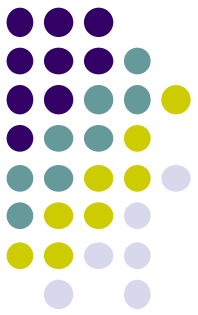




# Criando classes em Java

- **if – else                    if (expressão\_booleana )**  
                                  **{ bloco de comandos do if;**  
                                  **}**  
                                  **[else]**  
                                  **{ bloco de comandos do else;**  
                                  **}**
- **Else é opcional**

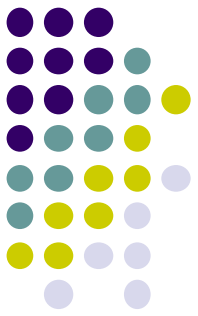




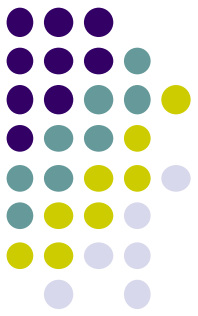
# Criando classes em Java

- O *escopo* dos campos e variáveis dentro de uma classe determina a sua visibilidade.
  - Campos declarados em uma classe são válidos por toda a classe, mesmo que os campos estejam declarados depois dos métodos que usam.
  - Variáveis e instâncias declaradas dentro de métodos só serão válidas dentro desse método.
  - Dentro de métodos e blocos de comandos, a ordem de declaração de variáveis e referências a instâncias é considerada.
    - devem ser declarados antes de serem utilizadas
  - Variáveis passadas como argumentos para métodos só são válidas dentro dos métodos.
  - Ex: *class Triangulo*

# Criando classes em Java

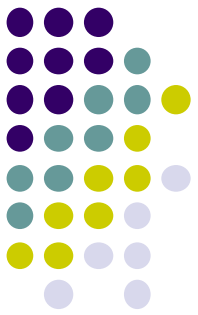


- Exercício
  - Criar uma classe triângulo
    - lado1, lado2, lado3
    - Método éEquilatero()



# Criando classes em Java

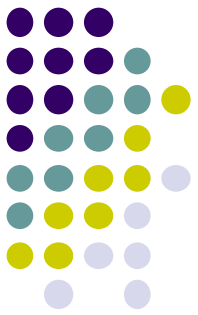
- Exemplo da Classe Triângulo
  - lado1, lado2, lado 3
    - Válidos por toda classe
  - Igualdade12, igualdade23
    - Válidas dentro do método éEquilátero.
    - E se declarar no final do método?
  - Resultado???



# Criando Classes em Java

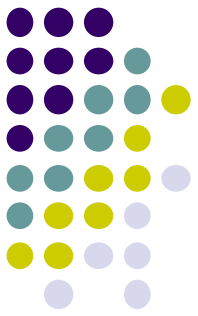
- Uma vantagem do paradigma OO é a possibilidade de *encapsular* campos e métodos capazes de manipular esses campos em uma classe
  - É desejável que campos das classes sejam ocultos.
  - Até agora demonstramos o *encapsulamento*, mas sem colocar os dados ocultos.
- Modificadores de acesso podem ser usados tanto em campos como em métodos de uma classe.
- O objetivo é proteger a integridade e a consistência dos dados e operações que uma determinada classe manipula.

# Classes em Java



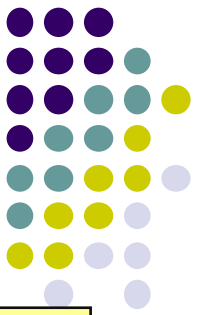
- Modificadores de Acesso
  - **public**: garante que o campo ou método da classe declarado com este modificador poderá ser acessados ou executado a partir de qualquer outra classe.
  - **private**: só podem ser acessados, modificados ou executados por métodos da mesma classe, sendo ocultos para o programador usuário que for usar instâncias desta classe ou criar classes herdeiras ou derivadas.
  - **protected**: funciona como o modificador private, exceto que classes herdeiras ou derivadas também terão acesso ao campo ou método.
  - Finalmente, campos e métodos podem ser declarados sem modificadores. Nesse caso, eles serão considerados como pertencentes à categoria **package**, significando que seus campos e métodos serão visíveis para todas as classes de um mesmo pacote.

# Classes em Java



- Modificadores de Acesso (cont)
  - Todo campo deve ser declarado como **private** ou **protected**.
  - Métodos que devem ser acessíveis devem ser declarados com o modificador **public**. Caso classes não venham a ser agrupadas em pacotes, a omissão não gera problemas.
  - Métodos para controle dos campos devem ser escritos, e estes métodos devem ter o modificador **public**.
  - Se for desejável, métodos podem ser declarados como **private**.

# Exemplo de Escopo e Modificadores



**public** determina que essa classe pode ser instanciada dentro de qualquer outra classe.

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

**private** determina que os dados só podem ser manipulados dentro da própria classe. Como “dia”, “mês” e “ano” são declarados fora dos métodos, são variáveis globais (atributos).

```
public void Data(byte d, byte m, short a)
{
    if (!DataEValida(d, m, a) == true)
```

**public** indica que o método pode ser acessado sem restrições

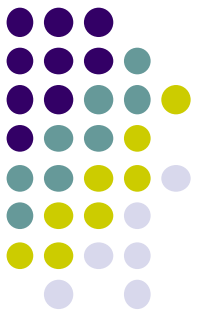
```
        dia = d; mes = m; ano = a;
    }

    dia = 0; mes = 0; ano = 0;
}
```

As variáveis “d”, “m” e “a” são parâmetros (argumentos) e só podem ser manipulados dentro desse método.



# Exemplo de Escopo e Modificadores (cont)



**private** indica que esse método só pode ser acessado através de outros métodos da própria classe.

A variáveis “validade” é uma variável local e só podem ser manipulada dentro desse método.

```
private boolean dataEValida(byte d, byte m, short a)
{
    boolean validade = false;
    if ((d >= 1) && (d <= 31) && (m >= 1) && (m <= 12))
        validade = true;
    return validade;
}
```

**protected** indica que esse método só pode ser acessado através de métodos dessa classe ou de classes herdeiras ou derivadas.

```
protected void mostraData()
{
    System.out.println(dia + "/" + mês + "/" + ano);
}
}
```