

**Bacharelado em Ciências da Computação**

*BC1501 – Programação Orientada a Objetos*



**Aula 02**

**Material adaptado do Prof.  
André G. R. Balan (CMCC)**

# Aula passada

## ▶ Variáveis de referência (Java)

- *Só por meio delas criamos e acessamos os objetos*
- *Podem aparecer dentro de um trecho do programa:*

```
void main() {  
    Mesa mesa01;  
    ...  
}
```

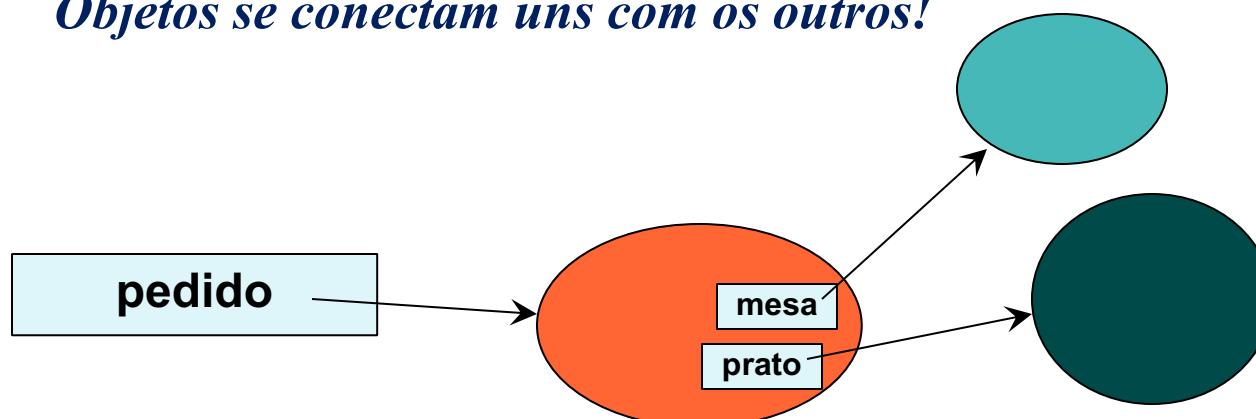
- *Podem aparecer como propriedade de uma classe:*

```
class Pedido {  
    private Mesa mesa;  
    ...  
}
```

# Aula passada

## ▶ Objetos

- *Um objeto é um pedaço de memória. Ele armazena dados*
- *Operações específicas sobre objetos são chamados métodos*
- ***Declarar uma variável de referência não significa criar um objeto***
- *Criar um objeto: operador **new***
- ***Objetos se conectam uns com os outros!***



# Aula passada

## ▶ Classes

- *É um modelo a partir do qual criam-se objetos*
  - **Propriedades** : *as informações que irão compor os objetos desta classe*
  - **Métodos**: *as operações específicas sobre os objetos desta classe*
- *Acesso às propriedades e métodos:*
  - **Public**
  - **Private**
  - **Protected** (*tem relação com herança*)

# Aula passada

## ► Classes - exemplos:

```
class Mesa {  
    int numero;  
}
```

```
class Prato {  
    String nome;  
}
```

```
class Garcom {  
    String nome;  
}
```

```
class Pedido {  
    int numero;  
    Mesa mesa;  
    Prato prato;  
    Garcom garcom;  
    void imprimir();  
}
```

# Aula passada

- ▶ Um simples programa de teste:

```
public static void main(String[] args) {  
    Mesa m1; // declaração de variáveis de referência  
    Prato p1;  
    Garcon g1;  
  
    m1 = new Mesa(); // novo objeto  
    m1.numero = 10;  
  
    p1 = new Prato(); // novo objeto  
    p1.nome = "Risoto";  
  
    g1 = new Garcon();  
    g1.nome = "Pedro Pedroso";  
  
    Pedido ped1 = new Pedido();  
    ped1.numero = 100;  
    ped1.mesa = m1;  
    ped1.prato = p1;  
    ped1.garcon = g1;  
    ped1.imprime();  
}
```

# Aula passada

- ▶ O método imprimir da classe **Pedido**:

```
class Pedido {  
    int numero;  
    Mesa mesa;  
    Prato prato;  
    Garcon garcon;  
  
    void imprimir() {  
        System.out.println("Pedido #" + numero);  
        System.out.println("Mesa #" + mesa.numero);  
        System.out.println("Prato: #" + prato.nome);  
        System.out.println("Garçon #" + garcon.nome);  
    }  
}
```

E se essas variáveis de referência não apontassem para nenhum objeto?

# Aula passada

- ▶ Este projeto tem vários arquivos. **Por onde devo começar?!!**
- ▶ Você não pode fazer um bolo sem ter primeiro os ingredientes, correto?
  - Nesse projeto, as classes são os ingredientes...
  - O programa principal é o bolo

# Nesta aula

## ▶ POO em Java

- Tipos nativos
- Wrappers
- Pacotes
- Construtores
- Sobrecarga
- Vetores de Objetos
- Classe ArrayList

# Tipos nativos em JAVA

## ▶ Tipos nativos/primitivos

Tipo	Faixa de valores
<b>boolean</b>	false ou true (8 bits)
<b>char</b>	0 a 65535 (16 bits)
<b>byte</b>	-128 a 127 (8 bits)
<b>short</b>	-32.768 a 32.767 (16 bits)
<b>int</b>	-2.147.483.648 a 2.147.483.647 (32 bits)
<b>long</b>	-9 *10 <sup>18</sup> a 9 * 10 <sup>18</sup> (64 bits)
<b>float</b>	32 bits
<b>double</b>	64 bits

Preste atenção nos tipos: Letras minúsculas!!!

# Tipos nativos em JAVA

- ▶ Tipos primitivos podem aparecer como **propriedades** de classes ou **variáveis simples**
- ▶ Exemplo: como **propriedade**

```
class Triangulo {  
  
    float lado1, lado2, lado3;  
  
    boolean eEquilatero { .... }  
  
    float calculaPerimetro { .... }  
}
```

# Tipos nativos em JAVA

- Exemplo: como variáveis simples (locais)

```
Boolean eEquilatero    {
    boolean igualdade12, igualdade23, resultado;

    igualdade12 = (lado1 == lado2);

    igualdade23 = (lado2 == lado3);

    resultado = (igualdade12 && igualdade23)

    return resultado;
}
```

# Wrappers

- ▶ Em Java, você também encontrará os seguintes tipos de variáveis:
  - **Integer , Long, Float, Boolean, Byte, Void , Double, Short,**
- ▶ Estes não são tipos primitivos!
- ▶ São classes chamadas Wrappers (empacotadores)
- ▶ Elas possuem funções úteis. Exemplo:

*// conversão de uma String em um Inteiro*

```
int a = Integer.parseInt("10")
```

# Wrappers

- ▶ De maneira muito interessante, para se criar um objeto dessas classes Wrappers em Java, não é preciso utilizar o operador **new!** Exemplo:

**Integer i = 10;**

é o mesmo que

**Integer i = new Integer(10);**

- ▶ Isto permite tratar objetos das classes Wrappers como uma simples variável primitiva!
- ▶ Este recurso da linguagem JAVA se chama “**autoboxing**” e só está presente a partir da versão 5 da linguagem (2004)

# Wrappers

- ▶ Qual o real motivo de existir as classes Wrappers?

# Wrappers

- ▶ Os desenvolvedores JAVA desenvolveram toda uma coleção de estruturas de dados (Listas, Pilhas, Filas, etc.), **para trabalharem unicamente com objetos.**
  - Motivo: uniformizar todas operações sobre as coleções de dados: Inserção, Remoção, Busca, Ordenação
  - Neste caso, ficou-se decidido que os tipos primitivos seriam **empacotados**

# Pacotes (Java)

- ▶ Os Pacotes são uma funcionalidade da linguagem Java que permite o agrupamento de classes com propósitos similares, em uma estrutura hierárquica
- ▶ Alguns pacotes fornecidos com a linguagem Java
  - java.util :: import java.util.\*
  - java.applets ::
- ▶ A linguagem C++ oferece uma funcionalidade parecida: os *Namespaces*. Um *namespace*, entretanto, pode agrupar muito mais que classes: variáveis, objetos, constantes, vetores, etc...

# Pacotes (Java)

- ▶ Em Java, ao criar novas classes, basta definirmos o nome do pacote no início do arquivo. Exemplo:

```
Package universidade.biblioteca
```

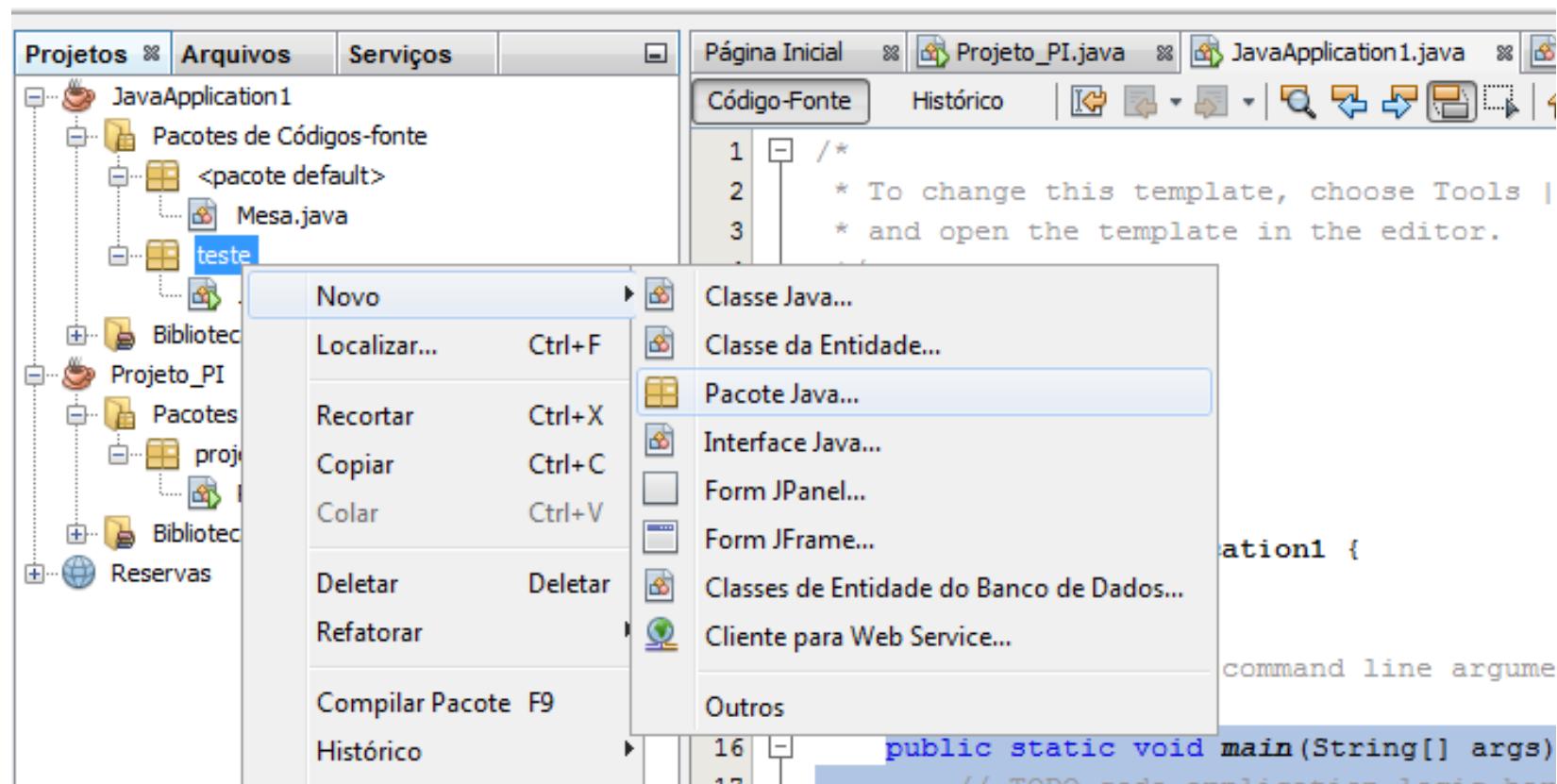
```
public class Livro {
```

```
}
```

- ▶ A palavra public garante que a classe ficará visível fora do pacote. Sem a palavra public, somente as classes daquele pacote podem utilizar a classe.

# Pacotes (Java)

## ► No Netbeans



# Construtores

## Construtor

- ▶ É um **método especial** da classe que é **executado automaticamente** no momento da criação de um novo objeto
  - Exemplo:      m1 = new **Mesa ()**;
  - *Aqui, estamos invocando o **construtor** da classe Mesa. É uma chamada de função sem parâmetros!*
  - *Se quisermos passar parâmetros para a construção do objeto, basta escrevermos na classe um outro método construtor!*

# Construtores

## Regras para escrever o método construtor na classe

- ▶ Deve ter exatamente o mesmo nome da classe
- ▶ Não tem nenhum retorno
- ▶ Deve ser declarado como **public**

# Construtores

- ▶ Exemplo: um construtor com um único parâmetro

```
class Mesa {  
    int numero;  
  
    public Mesa(int num) {  
        numero = num;  
    }  
}
```

- ▶ Agora, podemos criar um objeto passando um parâmetro

```
m1 = new Mesa(10);
```

# Construtores

- ▶ E se quiséssemos que o parâmetro **num** também se chamassem **numero** (o mesmo nome de uma das propriedades)?

```
class Mesa {  
    int numero;  
  
    public Mesa(int numero) {  
        numero = numero;  
    }  
}
```



# A palavra reservada **this**

- ▶ Usamos a palavra reservada **this** para nos referirmos à propriedade do objeto:

```
class Mesa {  
    int numero;  
  
    public Mesa(int numero) {  
        this.numero = numero;  
    }  
}
```

# Sobrecarga de métodos

- ▶ É possível termos mais de um construtor, desde que eles tenham números diferentes de parâmetros, ou tipos diferentes de parâmetros

```
class Mesa {  
    int numero;  
    int lugares;  
  
    public Mesa(int numero) {  
        this.numero = numero;  
    }  
    public Mesa(int numero, int lugares) {  
        this.numero = numero;  
        this.lugares = lugares;  
    }  
}
```

# Sobrecarga de métodos

- Depois, podemos usar o construtor que preferirmos:

```
Mesa m1 = new Mesa(10);
```

OU

```
Mesa m1 = new Mesa(10, 5);
```

- Vai depender de quais informações teremos antes da criação do objeto

# Sobrecarga de métodos

- ▶ Isto se chama **sobrecarga de construtor**
- ▶ Na verdade, é possível **sobreclarregar qualquer método**
  - Métodos da classe podem ter o mesmo nome, desde que o número ou os tipos dos parâmetros sejam diferentes em cada versão do método
- ▶ Exemplo

```
int     soma(int a, int b);  
  
double soma(double a, double b);
```

# Vetores de Objetos

- ▶ Considere o seguinte código em Java

```
class A {  
    public int n;  
}  
. . .  
// um vetor de 10 variáveis de referências  
A [] vet = new A[10];  
// Tentamos acessar o objeto apontado pela primeira  
variável. O que irá acontecer?  
vet[0].n = 1
```

# Vetores de Objetos

- ▶ Considere o seguinte código em Java

```
class A {  
    public int n;  
}  
  
.  
.  
.  
  
// um vetor de 10 variáveis de referências  
A [] vet = new A[10];  
  
// Tentamos acessar o objeto apontado pela primeira  
variável. O que irá acontecer?  
vet[0].n = 1
```

Exception in thread "main" java.lang.NullPointerException

Não existe objetos no vetor, somente variáveis de referência!

# Vetores de Objetos

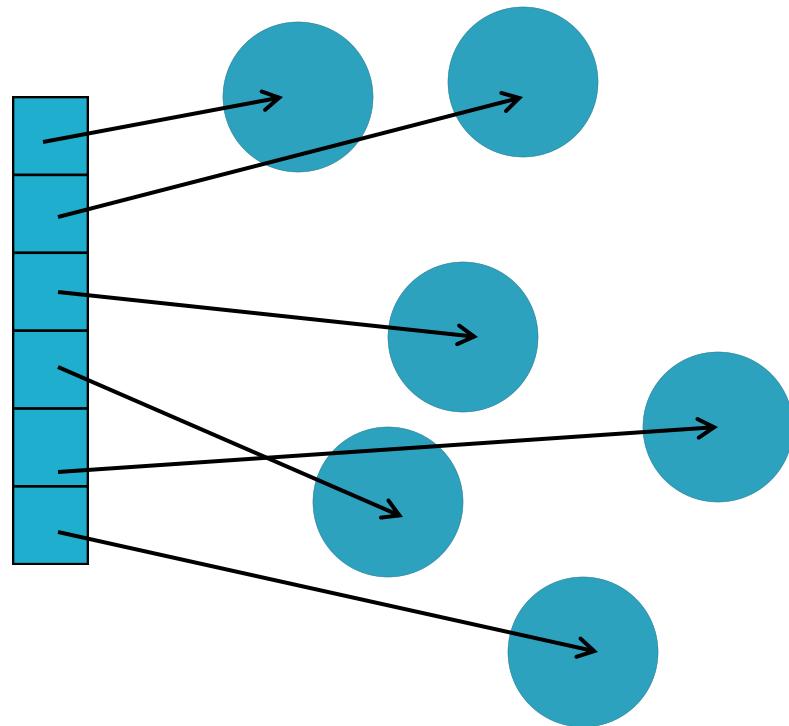
- ▶ Em Java não é possível alocar um vetor de objetos, somente um vetor de variáveis de referência!
- ▶ Para “colocar” um objeto no vetor:

```
vet[0] = new A();
```

```
vet[0].n = 10; // Assim não haverá erro
```

# Vetores de Objetos

- ▶ Veja como fica um “vetor de objetos” em Java



# Vetores de Objetos

- ▶ Em um **verdadeiro vetor de objetos**, o conjunto de todos os objetos deveriam ocupar uma porção contígua de memória, seja na Heap ou na Stack do processo.
- ▶ Em **C++** é possível criar verdadeiros vetores de objeto:

```
A *vet = new A[10]; // cria na Heap
```

```
A vet[10] // cria na stack
```

# A classe ArrayList

- ▶ A classe ArrayList presente no pacote **java.util** é uma importante **estrutura de dados** para lidar com grupos de objetos
- ▶ Uma ED Lista é importante para uma série de aplicações:
  - Lista de usuários em uma sala de Chat
  - Lista de disciplinas de um curso
  - Lista de jogadas de um jogo de Xadrez
  - Etc....
- ▶ Seu mecanismo interno utiliza vetores

# A classe ArrayList

- ▶ Como o próprio nome diz, a classe ArrayList é uma ED lista cujo mecanismo interno utiliza ‘**vetores de objetos**’.
- ▶ Para nós, entretanto, não interessa **como** a Lista funciona internamente, nos interessa quais as funcionalidades que ele nos oferece.
  - **Abstração**

# A classe ArrayList

- ▶ Os principais métodos da classe ArrayList
  - Insere no final - add(Object var)
  - Insere em um posição específica - add(int pos, Object var)
  - Remove de uma posição - remove(int pos)
  - Consulta se um elemento está contido - contains(Objetc var)
  - Verifica se a lista está Vazia - isEmpty()
  - Limpa – remove todos os elementos - clear()

# A classe ArrayList

- ▶ Exemplo: Uma lista de Pratos

```
ArrayList<Prato> pratos;  
  
pratos = new ArrayList<Prato>();  
  
Prato p1 = new Prato("Feijoada");  
  
pratos.add(p1);
```

# A classe ArrayList

- ▶ E se quisermos então que um pedido de restaurante tenha vários pratos?
- ▶ Então, um objeto da classe Pedido deve ser capaz de apontar/referenciar vários objetos da classe prato.
- ▶ Como fazemos isso?
- ▶ Colocamos um ArrayList de pratos dentro de um objeto Pedido!!

# A classe ArrayList

- ▶ A nova classe Pedido

```
class Pedido {  
    int numero;  
    Mesa mesa;  
    Garcon garcon;  
    ArrayList<Prato> pratos;  
  
    public Pedido () {    // construtor  
        pratos = new ArrayList<Prato>();  
    }  
  
    public void adicionaPrato(Prato prato) {  
        pratos.add(prato);  
    }  
}
```

# A classe ArrayList

- ▶ A nova classe Pedido

```
class Pedido {  
    .  
    .  
    .  
  
    public void imprime() {  
        . . .  
        for(Prato p : pratos) { // Imprime a lista de pratos!!  
            System.out.println(p.nome);  
        }  
        . . .  
    }  
}
```