Programação III

Aula 3 – Orientação a Objetos com Java

Definição de Classes

```
class Box {
    private double width;
    private double height;
    private double depth;

    public Box(double width, double height, double depth) {
        this.width = w;
        this.height = h;
        this.depth = d;
    }
    public Box() {
        width = -1;
        height = -1;
        depth = -1;
    }
    public Box(double len) {
        width = height = depth = len;
    }
    public double volume() {
        return width * height * depth;
    }
}
```

Uso das classes

```
class BoxDemo {
    public static void main(String[] args) {
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box();
        Box mycube = new Box(7);
        double vol;
        vol = mybox1.volume();
        System.out.println("Volume of mybox1 is " + vol);

        vol = mybox2.volume();
        System.out.println("Volume of mybox2 is " + vol);

        vol = mycube.volume();
        System.out.println("Volume of mycube is " + vol);
    }
}
```

Objetos como parâmetros

```
class Test {
    int a, b;
    Test(int i, int j) {
        a = i;
        b = j;
    }

    boolean equalTo(Test o) {
        if(o.a == a && o.b == b) return true;
        else return false;
    }
}
```

```
class Box {
    /* ... */
    Box(Box ob) {
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
    }
    /* ... */
}
```

Passagem por cópia ou referência

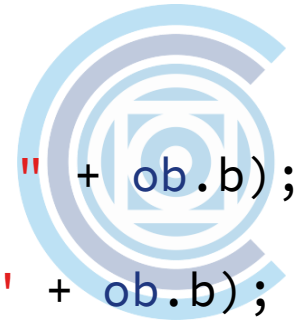
```
// Tipos primitivos são passados por cópia.
class Test {
    void meth(int i, int j) {
        i *= 2;
        j /= 2;
    }
}
class CallByValue {
    public static void main(String[] args) {
        Test ob = new Test();
        int a = 15, b = 20;
        System.out.println("a and b before call: " + a + " " + b);
        ob.meth(a, b);
        System.out.println("a and b after call: " + a + " " + b);
    }
}
```

Passagem por cópia ou referência

- Objetos são sempre passados por referência

```
class Test {
    /* … */
    void meth(Test o) {
        o.a *= 2;
        o.b /= 2;
    }
}

class PassObjRef {
    public static void main(String[] args) {
        Test ob = new Test(15, 20);
        System.out.println("ob.a and ob.b before call: " + ob.a + " " + ob.b);
        ob.meth(ob);
        System.out.println("ob.a and ob.b after call: " + ob.a + " " + ob.b);
    }
}
```



Uso do Static

- Independem de instâncias da classe.
- Declarados com a palavra-chave static.
- Acessíveis antes da criação de objetos.
- **Características:**
  - **Variáveis Estáticas:** Compartilhadas por todas as instâncias, acessíveis diretamente pela classe.
  - **Métodos Estáticos:**
    - Chamam apenas métodos estáticos.
    - Acessam apenas variáveis estáticas.
    - Não usam this ou super.
  - **Blocos Estáticos:** Executados **uma única vez** ao carregar a classe

Uso do Static

```
class StaticDemo {
    static int a = 42;
    static void callme() {
        System.out.println("a = " + a);
    }
    static {
        System.out.println("Bloco estático inicializado.");
    }
    public static void main(String[] args) {
        StaticDemo.callme();
    }
}
```

Saída:

```
Bloco estático inicializado.
a = 42
```

Classes internas e aninhadas

- **Definição:**
  - **Classes Aninhadas:** Definidas dentro de outra classe, limitadas ao escopo da classe externa.
  - **Tipos:**
    - **Estática:** Usa o modificador static e acessa membros não estáticos da classe externa somente via objeto.
    - **Interna:** Não estática, acessa diretamente todos os membros da classe externa.
- **Características:**
  - A classe interna pode acessar os membros da classe externa, inclusive privados.
  - A classe externa **não pode** acessar membros da classe interna diretamente.
  - Classes internas podem ser declaradas em qualquer bloco, como métodos ou laços.

Classes internas e aninhadas

```
class Outer {
    int outer_x = 100;
    void test() {
        Inner inner = new Inner();
        inner.display();
    }
    class Inner { // Classe Interna
        void display() {
            System.out.println("display: outer_x = " + outer_x);
        }
    }
}
class InnerClassDemo {
    public static void main(String[] args) {
        Outer outer = new Outer();
        outer.test();
    }
}
```

Saída:

```
display: outer_x = 100
```

Exemplo de Uso

```
class Window {
    private String title;

    Window(String title) {
        this.title = title;
    }

    class Button {
        private String label;

        Button(String label) {
            this.label = label;
        }

        void click() {
            System.out.println("Botão '" + label + "' clicado na janela '" + title + "'.");
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Window window = new Window("Configurações");
        Window.Button button = window.new Button("Salvar");
        button.click();
    }
}
```

Classes internas locais

```
class Outer {
    int outer_x = 100;
    void test() {
        for (int i = 0; i < 3; i++) {
            class Inner {
                void display() {
                        System.out.println("display: outer_x = " + outer_x);
                }
             }
            Inner inner = new Inner();
            inner.display();
        }
    }
}
```

Saída:

```
display: outer_x = 100 (3 vezes)
```

- **Uso Prático:**
  - Especialmente útil para tratar eventos ou simplificar o código.
  - Classes internas anônimas serão abordadas posteriormente.

Exemplo de Uso

```
class Product {
    String name;
    double price;

    Product(String name, double price) {
        this.name = name;
        this.price = price;
    }

    @Override
    public String toString() {
        return name + " - $" + price;
    }
}

class Shop {
    void displayProductsSortedByPrice(List<Product> products) {
        class PriceComparator implements java.util.Comparator<Product> {
            @Override
            public int compare(Product p1, Product p2) {
                return Double.compare(p1.price, p2.price);
            }
        }

        Collections.sort(products, new PriceComparator());
        products.forEach(System.out::println);
    }
}
```

Varargs

- **Definição**
  - Abreviação de "variable-length arguments".
  - Permite métodos que recebem número variável de argumentos, simplificando chamadas.
- **Antigas Soluções:**
  - **Sobrecarga:** Criar múltiplas versões do método.
  - **Array:** Passar argumentos manualmente dentro de um array.
    - **Desvantagem:** Tedioso e sujeito a erros.

- **Sintaxe:**

```
static void vaTest(int ... v) {
    // v é tratado como um array do tipo int[].
}
```

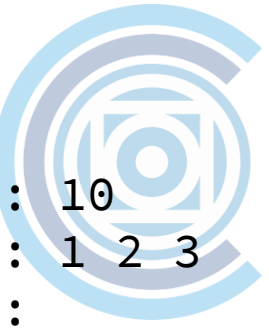
- **Vantagens:**
  - Argumentos são automaticamente colocados em um array.
  - Chamadas mais simples e diretas.

Varargs - Exemplo

```
class VarArgs {
    static void vaTest(int ... v) {
        System.out.print("Número de args: " + v.length + " Conteúdo: ");
        for (int x : v) System.out.print(x + " ");
        System.out.println();
    }
    public static void main(String[] args) {
        vaTest(10); // 1 argumento
        vaTest(1, 2, 3); // 3 argumentos
        vaTest(); // nenhum argumento
    }
}
```

Saída:

```
Número de args: 1 Conteúdo: 10
Número de args: 3 Conteúdo: 1 2 3
Número de args: 0 Conteúdo:
```

Varags - Overloading

- Um método varargs pode coexistir com métodos normais.
- Métodos normais têm preferência sobre métodos varargs

```
class OverloadVarargs {
    static void test(int a) {
        System.out.println("Método com 1 parâmetro int: " + a);
    }
    static void test(int ... a) {
        System.out.println("Método varargs com int[]. Número de args: " + a.length);
    }
    public static void main(String[] args) {
        test(5); // Chama o método normal
        test(1, 2, 3); // Chama o método varargs
    }
}
```

Varargs - Ambiguidade

- Dois métodos podem ser igualmente válidos para uma chamada.

```
class AmbiguityExample {
    static void test(int ... a) {
        System.out.println("Varargs com int[]");
    }
    static void test(long ... a) {
        System.out.println("Varargs com long[]");
    }
    public static void main(String[] args) {
        // test(10); // Erro: Ambiguidade!
    }
}
```

Varargs - Ambiguidade

- Soluções: evitar sobrecarga ambígua ou realizar chamadas explícitas:

```
class ResolveAmbiguity {
    static void test(int ... a) {
        System.out.println("Varargs com int[]");
    }
    static void test(long ... a) {
        System.out.println("Varargs com long[]");
    }
    public static void main(String[] args) {
        test((int) 10); // Chama o método com int
        test((long) 10); // Chama o método com long
    }
}
```