

# Variáveis Compostas e Subprogramação

Orientação a Objetos – DCC025

Gleiph Ghiotto Lima de Menezes

[gleiph.ghiotto@ufjf.br](mailto:gleiph.ghiotto@ufjf.br)

# Aula de hoje

- Veremos os diferentes tipos de variáveis compostas (*arrays*)
  - Com uma dimensão (vetores)
  - Com duas ou mais dimensões (matrizes)
- Estudaremos a estrutura mais básica de encapsulamento da Orientação a Objetos
  - Métodos

# Exemplo Motivacional

- Programa para auxiliar a escrever “Parabéns!” nas melhores provas de uma disciplina com 3 alunos
  - Ler os nomes e as notas de 3 alunos
  - Calcular a média da turma
  - Listar os alunos tiveram nota acima da média

# Exemplo Motivacional

```
import java.util.Scanner;

public class Notas {

    public static void main(String[] args) {
        Scanner teclado = new Scanner(System.in);
        String nome1, nome2, nome3;
        float nota1, nota2, nota3, media;

        System.out.print("Informe o nome do aluno 1: ");
        nome1 = teclado.nextLine();
        System.out.print("Informe o nome do aluno 2: ");
        nome2 = teclado.nextLine();
        System.out.print("Informe o nome do aluno 3: ");
        nome3 = teclado.nextLine();
    }
}
```



# Exemplo Motivacional

```

System.out.print("Informe a nota de " + nome1 + ": ");
nota1 = teclado.nextFloat();
System.out.print("Informe a nota de " + nome2 + ": ");
nota2 = teclado.nextFloat();
System.out.print("Informe a nota de " + nome3 + ": ");
nota3 = teclado.nextFloat();
media = (nota1 + nota2 + nota3)/3;

if (nota1 > media)
    System.out.println("Parabéns " + nome1);
if (nota2 > media)
    System.out.println("Parabéns " + nome2);
if (nota3 > media)
    System.out.println("Parabéns " + nome3);
    }
}

```

E se fossem 40 alunos?

- É possível definir variáveis que guardam mais de um valor de um mesmo tipo
- Essas variáveis são conhecidas como variáveis compostas, variáveis subscritas, variáveis indexáveis ou arranjos (*array*)
- Existem dois tipos principais de variáveis compostas:
  - Vetores
  - Matrizes

# Vetores

- Variável composta **unidimensional**
  - Contém espaço para armazenar diversos valores de um mesmo tipo
  - É acessada via um índice
- A ideia de vetor é comum na matemática, com o nome de variável subscrita
  - Exemplo:  $x_1, x_2, \dots, x_n$
- O que vimos até agora são variáveis com somente um valor
  - Exemplo:  $x = 123$
- No caso de vetores, uma mesma variável guarda ao mesmo tempo múltiplos valores
  - Exemplo:  $x_1 = 123, x_2 = 456, \dots$

# Recapitulando: variáveis que contêm tipos primitivos

- Até agora, variáveis que contêm tipos primitivos (byte, short, int, long, float, double, char, boolean) sempre ocupam diretamente uma posição na memória

nota1	8.2
flaTheBest	true
nota2	6.0
nota3	7.1
bloco	'A'



# Retomando: Vetores

	0	8.2	
notas	1	6.0	
	2	7.1	
flaTheBest		true	
	0	"João"	
nomes	1	"Pedro"	
bloco		'B'	

# Declaração de vetores

- Forma geral

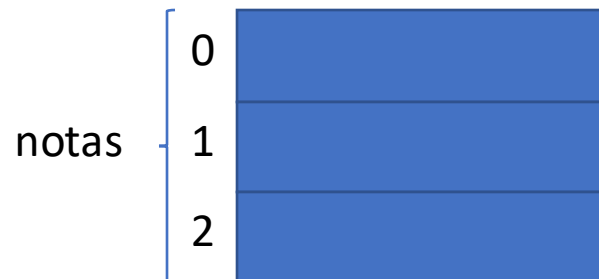
```
TIPO[] NOME = new TIPO[TAMANHO];  
ou  
TIPO[] NOME;  
...  
NOME = new TIPO[TAMANHO];
```

- Exemplos

```
String[] nomes = new String[40];  
float[] notas = new float[40];  
boolean[] presenca;  
presenca = new boolean[5];
```

# Declaração de vetores

- É possível saber o tamanho de um vetor acessando a propriedade *length*
  - Exemplo: `notas.length` → 40
- No Java, todo vetor inicia na posição 0 (zero) e termina na posição *length - 1*
  - Exemplo: `float[] notas = new float[3];`



# Utilização de vetores

- Para acessar (ler ou escrever) uma posição do vetor, basta informar a posição entre colchetes

```
notas[0] = 8;
notas[1] = 5.5f;
notas[2] = 1.5f;
media = (notas[0] + notas[1] + notas[2]) / 3;
```

notas	0	8.0
	1	5.5
	2	1.5
media		5.0

# Utilização de vetores

- Também é possível iniciar os valores de vetores diretamente no código, colocando-os entre chaves ({}), separados por vírgula

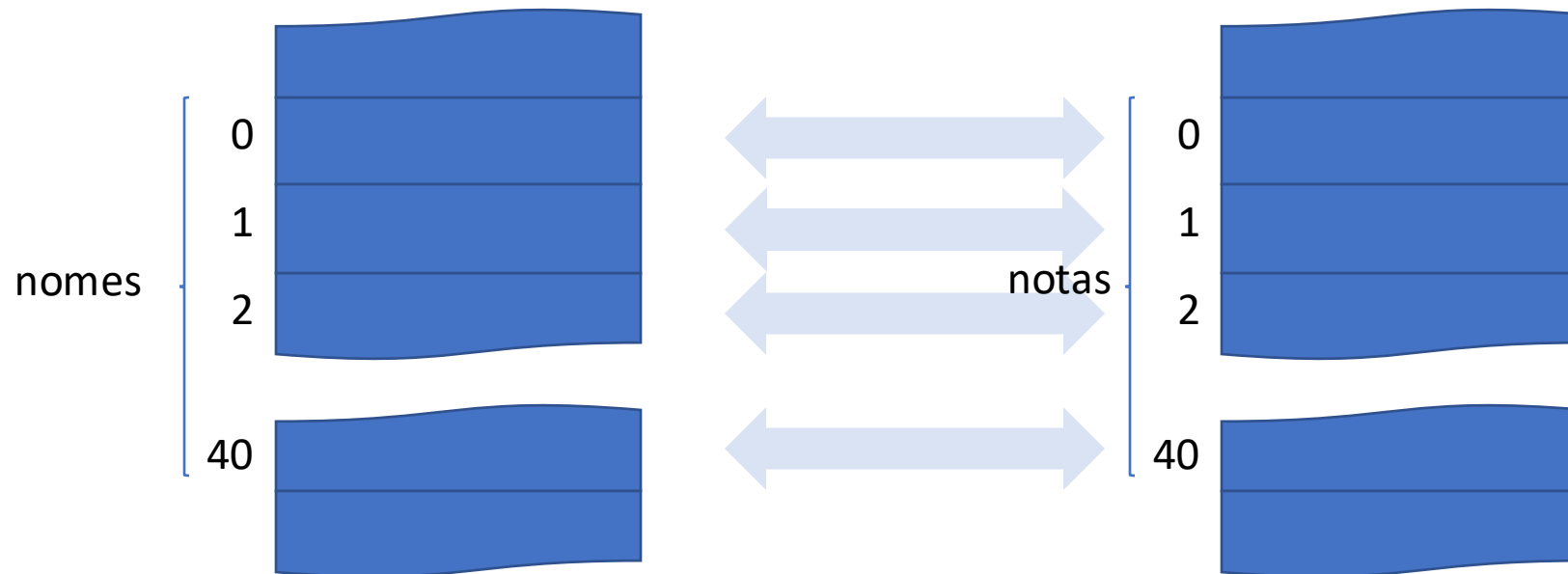
```
notas = { 8, 5.5f, 1.5f};
media = (notas[0] + notas[1] + notas[2]) / 3;
```

- Outra possibilidade é de iterar por todos os seus valores

```
for (int i = 0; i < notas.length; i++) {
    System.out.print(notas[i]);
}
```

## Retomando: E se fossem 40 alunos?

- Criaríamos dois vetores (nomes e notas) de 40 posições
- Vincularíamos a posição  $N$  do vetor de nomes à posição  $N$  do vetor de notas



# Retomando: E se fossem 40 alunos?

```
import java.util.Scanner;

public class Notas {

    public static void main(String[] args) {
        final int NUMERO_ALUNOS = 40;
        Scanner teclado = new Scanner(System.in);
        String[] nomes = new String[NUMERO_ALUNOS];
        float[] notas = new float[NUMERO_ALUNOS];
        float media = 0;

        for (int i = 0; i < NUMERO_ALUNOS; i++) {
            System.out.print("Informe o nome do aluno " + (i+1) + ": ");
            nomes[i] = teclado.nextLine();
        }
    }
}
```



## Retomando: E se fossem 40 alunos?



```

for (int i = 0; i < NUMERO_ALUNOS; i++) {
    System.out.print("Informe a nota de " + nomes[i] + ": ");
    notas[i] = teclado.nextFloat();
    media += notas[i];
}
media /= NUMERO_ALUNOS;

for (int i = 0; i < NUMERO_ALUNOS; i++) {
    if (notas[i] > media)
        System.out.println("Parabéns " + nomes[i]);
}
}

```



# Matrizes

- Variável composta **multidimensional**
  - É equivalente a um vetor, contudo permite a utilização de diversas dimensões acessadas via diferentes índices
  - Pode ser pensada como um vetor cujo tipo é outro vetor, recursivamente
  - Em diversas situações matrizes são necessárias para correlacionar informações

## Exemplo motivacional

- Assumindo que **um aluno é avaliado com três notas**, seria necessário um vetor de três posições para guardar as notas de um aluno...

notas	0	4.5
	1	6.5
	2	7.0

# Exemplo motivacional

- Contudo, assumindo que **uma turma tem cinco alunos**, seria necessária uma matriz bidimensional para guardar as notas de todos os alunos de uma turma...

		alunos				
		0	1	2	3	4
notas	0		4.5			
	1		6.5			
	2		7.0			

```
float[][] notas = new float[5][3]; // Declaração
System.out.println(notas[1][0]);
```

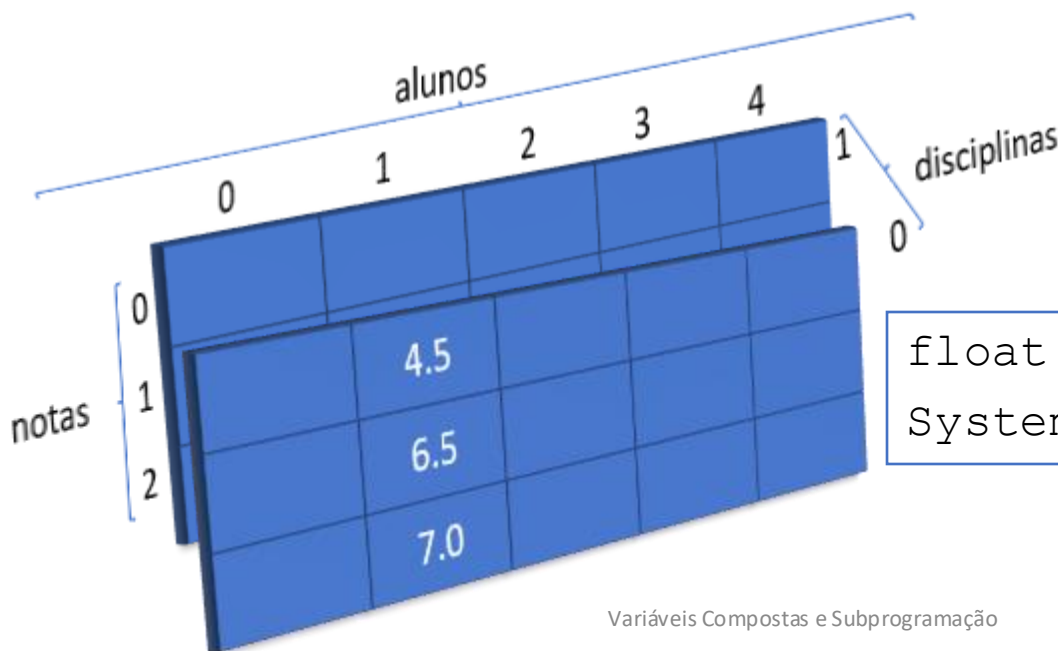
# Exemplo motivacional

- Na memória, podemos imaginar que seria algo assim...



## Exemplo motivacional

- Ainda, assumindo que **uma disciplina tem duas turmas**, seria necessária uma matriz tridimensional para guardar as notas de todos os alunos de todas as turmas da disciplina...



```
float[][][] notas = new float[2][5][3];  
System.out.println(notas[0][1][0]);
```

# Subprogramação

# Exemplo

```
import java.util.Scanner;
public class IMC {
    public static void main(String[] args) {
        Scanner teclado = new Scanner(System.in);

        System.out.print("Entre com a sua altura em metros: ");
        double altura = teclado.nextDouble();

        System.out.print("Entre com a sua massa em kg: ");
        double massa = teclado.nextDouble();

        double imc = massa / Math.pow(altura, 2);
        System.out.println("Seu IMC é " + imc);
    }
}
```

*Parecidos!*

# Exemplo usando método

```
import java.util.Scanner;
public class IMC {
    public static double leia(String mensagem) {
        Scanner teclado = new Scanner(System.in);
        System.out.print(mensagem);
        return teclado.nextDouble();
    }
}
```

*Declaração  
do método*

```
public static void main(String[] args) {
    double altura = leia("Entre com a sua altura em metros: ");
    double massa = leia("Entre com a sua massa em kg: ");

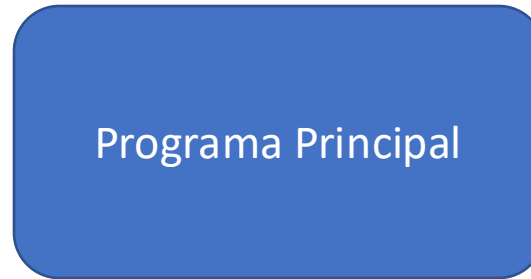
    double imc = massa / Math.pow(altura, 2);
    System.out.println("Seu IMC é " + imc);
}
}
```

*Chamadas  
ao método*

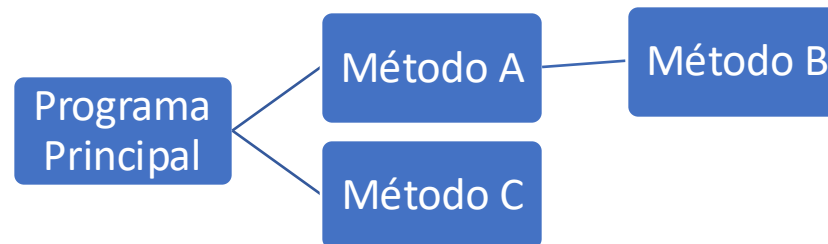


# Dividir para conquistar

- Antes: um programa gigante

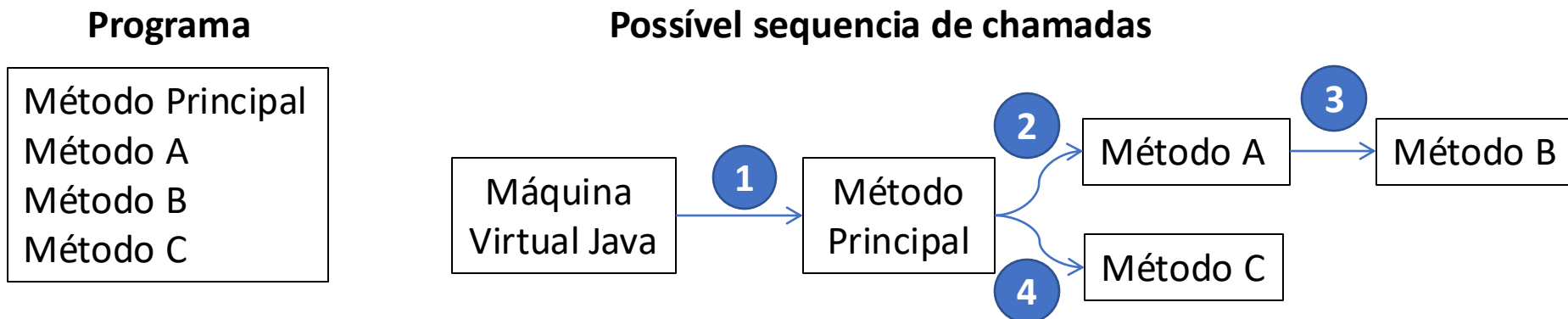


- Depois: vários programas menores

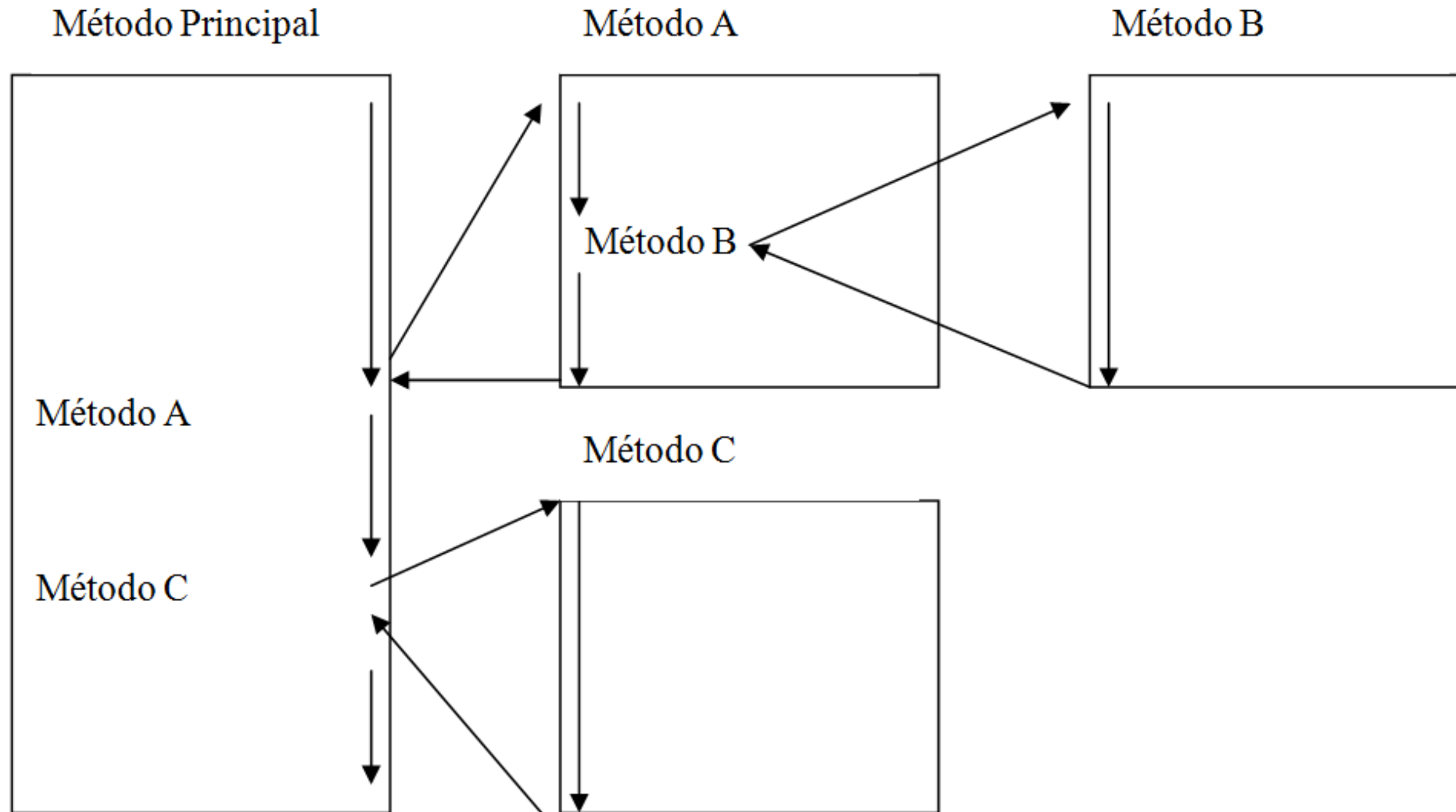


# Fluxo de execução

- O programa tem início em um método principal (no caso do Java é o método *main*)
- O método principal chama outros métodos
- Estes métodos podem chamar outros métodos, sucessivamente
- Ao fim da execução de um método, o programa retorna para a instrução seguinte à da chamada ao método

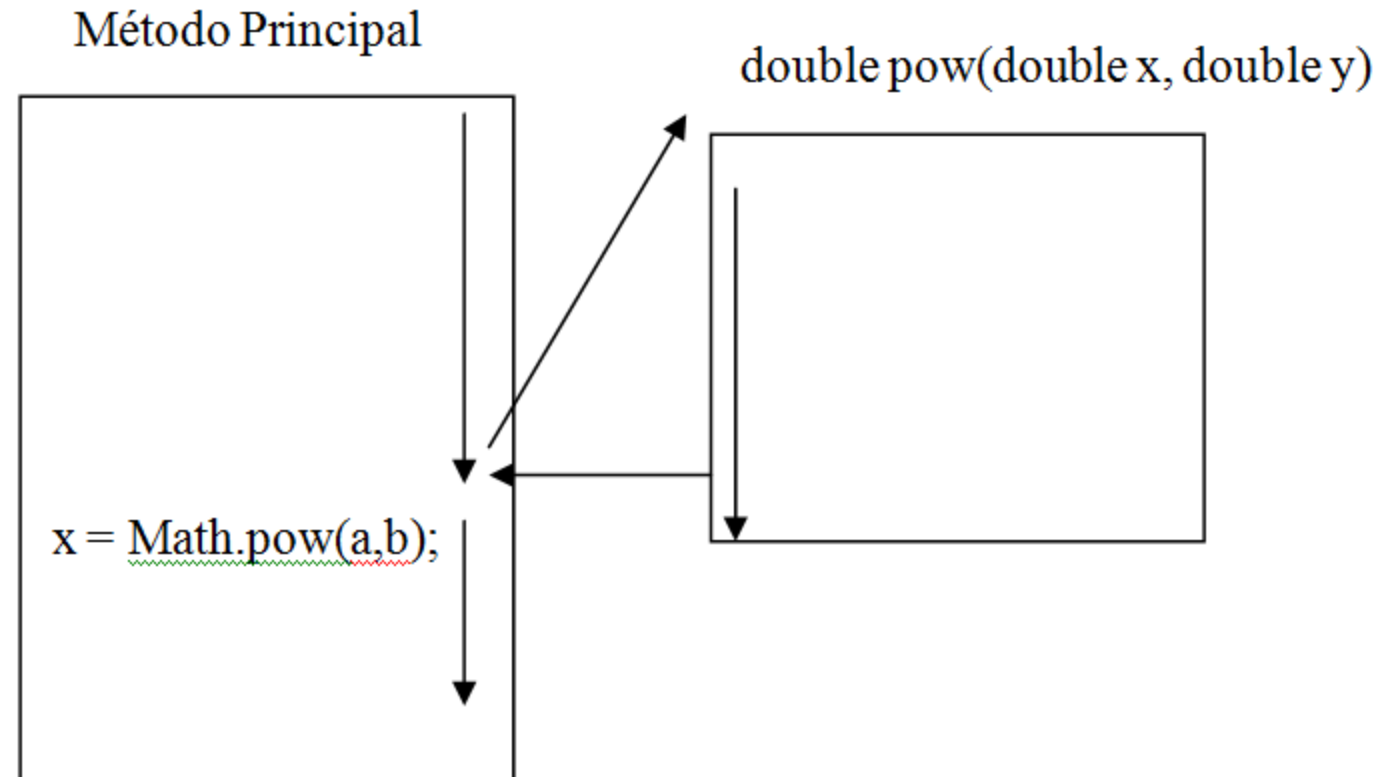


# Fluxo de execução



# Fluxo de execução

- É equivalente ao que acontece quando chamamos um método predefinido do Java



# Vantagens

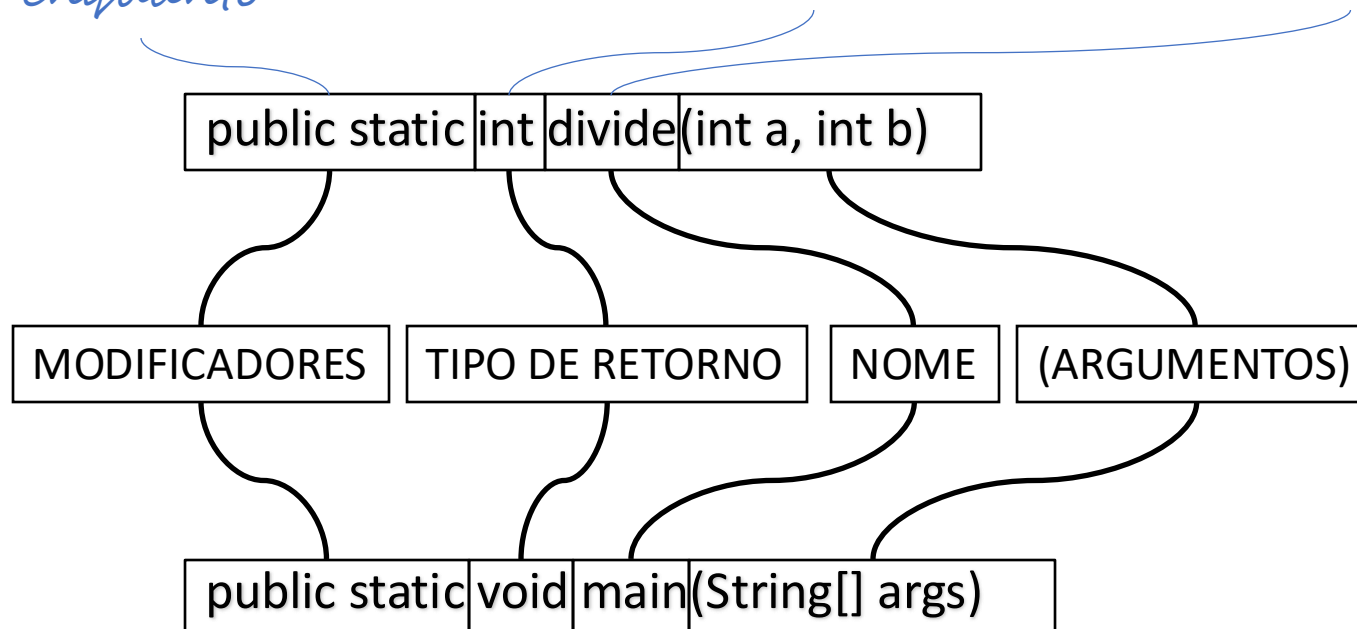
- Economia de código
  - Quanto mais repetição, mais economia
- Facilidade na correção de defeitos
  - Corrigir o defeito em um único local
- Legibilidade do código
  - Podemos dar nomes mais intuitivos a blocos de código
  - É como se criássemos nossos próprios comandos
- Melhor tratamento de complexidade
  - Estratégia de “dividir para conquistar” nos permite lidar melhor com a complexidade de programas grandes
  - Abordagem *top-down* ajuda a pensar!

# Sintaxe de um método

*Vamos usar esses modificadores por enquanto*

*Qualquer tipo da linguagem*

*Mesma regra de nome de variável*



*Significa que não tem retorno*

*Mesma regra de declaração de variáveis, separando por vírgula cada argumento*

# Acesso a variáveis

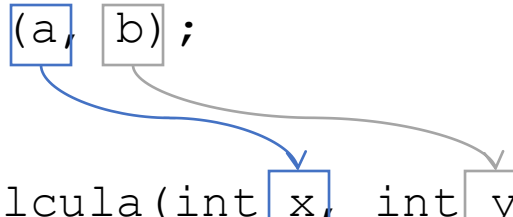
- Um método não consegue acessar as variáveis de outros métodos
  - Cada método pode criar as suas próprias variáveis locais
  - Os parâmetros para a execução de um método devem ser definidos como argumentos do método
- Passagem por valor
  - Java **copiará o valor** de cada argumento para a respectiva variável
  - Os nomes das variáveis podem ser diferentes

```

z = calcula(a, b);

public static double calcula(int x, int y)

```



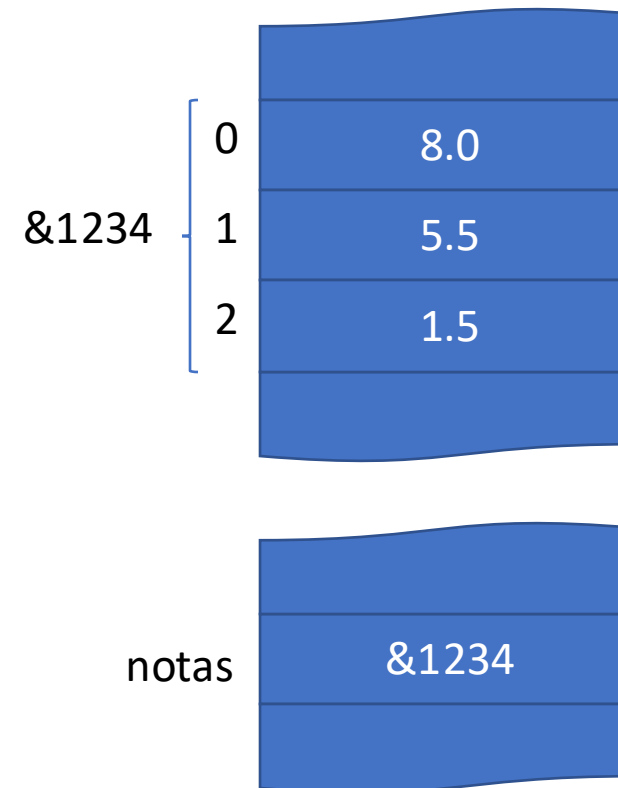
# Exemplo

```
public class Troca {
    public static void troca(int x, int y) {
        int aux = x;
        x = y;
        y = aux;
    }
    public static float media(int x, int y) {
        return (x + y) / 2f;
    }
    public static void main(String[] args) {
        int a = 5;
        int b = 7;
        troca(a, b);
        System.out.println("a: " + a + ", b: " + b);
        System.out.println("média: " + media(a,b));
    }
}
```



# Passagem de ponteiro por valor

- Variáveis compostas são, na verdade, ponteiros.
- Seus endereços são passados por valor
  - Se criar uma nova variável, o efeito não é notado fora do método
  - Se trocar o valor de uma posição da variável, o efeito é notado fora do método



# Exemplo 1

```
public class Array {
    public static void mostra(int[] array) {
        System.out.println(array[0] + ", " + array[1]);
    }

    public static void troca(int[] array) {
        array = new int[2];
        array[0] = 20;
        array[1] = 10;
    }

    public static void main(String[] args) {
        int[] array = { 10, 20 };
        mostra(array);
        troca(array);
        mostra(array);
    }
}
```

## Exemplo 2

```
public class Array {
    public static void mostra(int[] array) {
        System.out.println(array[0] + ", " + array[1]);
    }

    public static void troca(int[] array) {
        int tmp = array[0];
        array[0] = array[1];
        array[1] = tmp;
    }

    public static void main(String[] args) {
        int[] array = { 10, 20 };
        mostra(array);
        troca(array);
        mostra(array);
    }
}
```

# Sobrecarga de métodos

- Uma classe pode ter **dois ou mais métodos com o mesmo nome**, desde que os tipos de seus argumentos sejam distintos
- Isso é útil quando queremos implementar um método em função de outro
- Exemplo baseado na classe String:

```
public int indexOf(String substring) {
    return indexOf(substring, 0);
}
```

# Métodos sem argumentos

- Não é necessário ter argumentos nos métodos
  - Nestes casos, é obrigatório ter () depois do nome do método
  - A chamada ao método também precisa conter ()

- Exemplo de declaração:

```
public static void pulaLinha() {
    System.out.println();
}
```

- Exemplo de chamada:

```
pulaLinha();
```

# Exercício 1

- Leia o nome e a idade de 10 pessoas e liste as pessoas
  - Em ordem alfabética
  - em ordem crescente de idade

## Exercício 2

- Faça uma calculadora que forneça as seguintes opções para o usuário, usando métodos sempre que possível
- A calculadora deve operar sempre sobre o valor corrente na memória

Estado da memória: 0

Opções:

- (1) Somar
- (2) Subtrair
- (3) Multiplicar
- (4) Dividir
- (5) Limpar memória
- (6) Sair do programa

Qual opção você deseja?

```
package br.ufjf.dcc.oo.desenho.formas;
```

```
import javax.swing.JFrame;
import javax.swing.JOptionPane;
```

```
public class TesteFormas {

    public static void main(String[] args) {
        JFrame tela = new JFrame();

        String valorString = JOptionPane.showInputDialog(
            "Digite "
            + "1 para desenhar retângulos e "
            + "2 para desenhar círculos...");
        int valorInt = Integer.parseInt(valorString);

        FormasPanel formasPanel = new FormasPanel(valorInt);

        tela.add(formasPanel);
        tela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        tela.setSize(500, 500);
        tela.setVisible(true);
    }
}
```

```
package br.ufjf.dcc.oo.desenho.formas;
```

```
import java.awt.Graphics;
import javax.swing.JPanel;
```

```
public class FormasPanel extends JPanel {
```

```
    int forma;
```

```
    public FormasPanel(int forma) {
        super();
        this.forma = forma;
    }
```

```
@Override
```

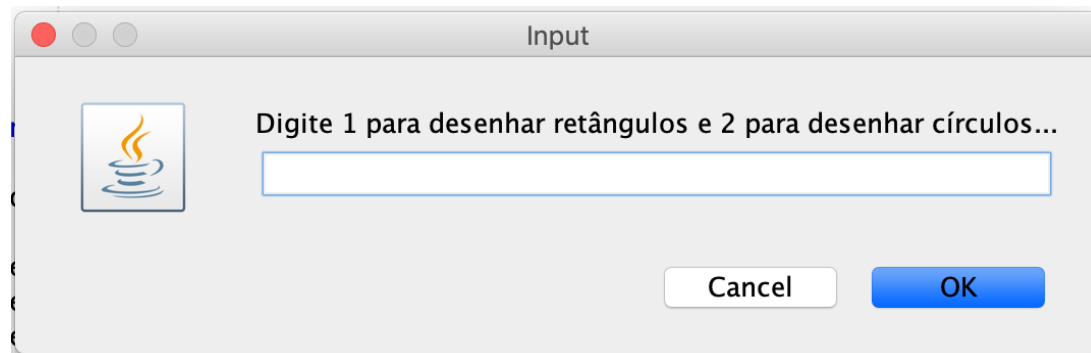
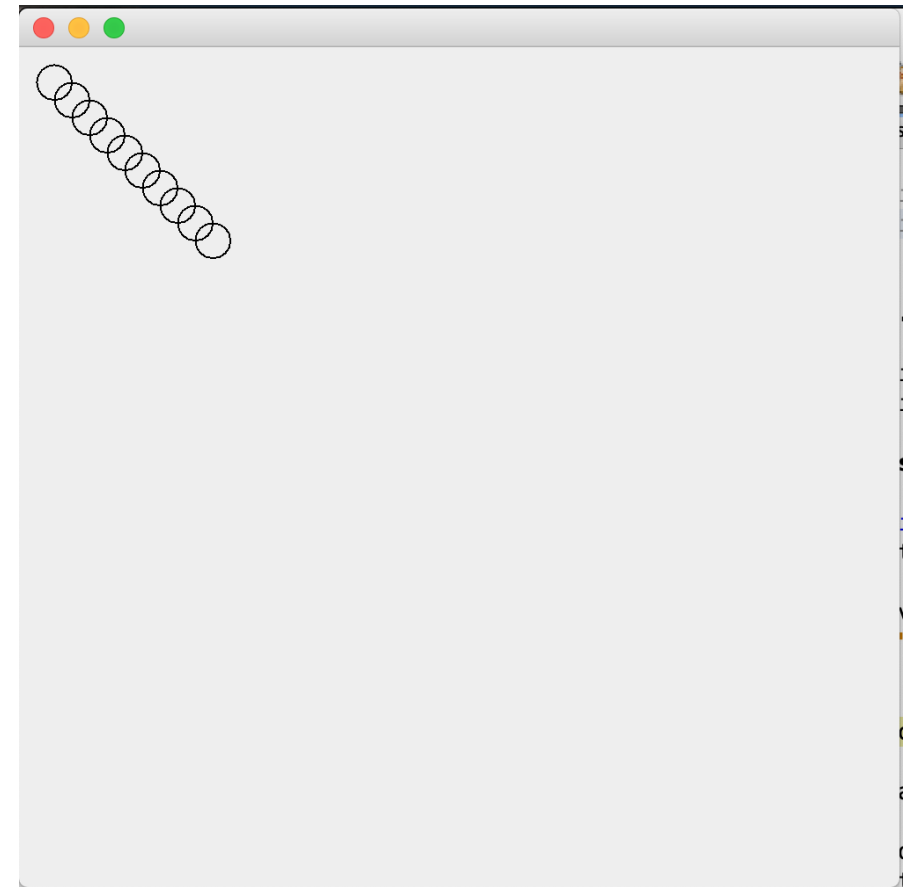
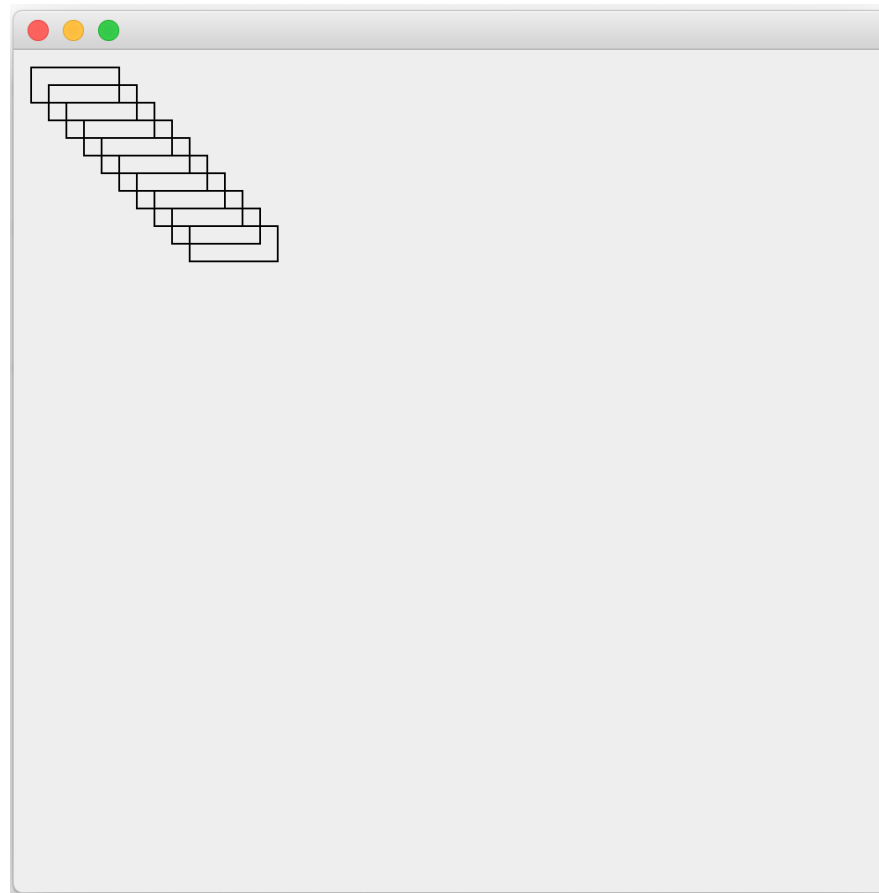
```
protected void paintComponent(Graphics g) {
    super.paintComponent(g);
```

```
    for (int i = 0; i < 10; i++) {
        if (this.forma == 1) {
            g.drawRect(10 + i * 10, 10 + 10 * i, 50, 20);
        } else if (this.forma == 2) {
            g.drawOval(10 + 10 * i, 10 + 10 * i, 20, 20);
        }
    }
```

```
}
```

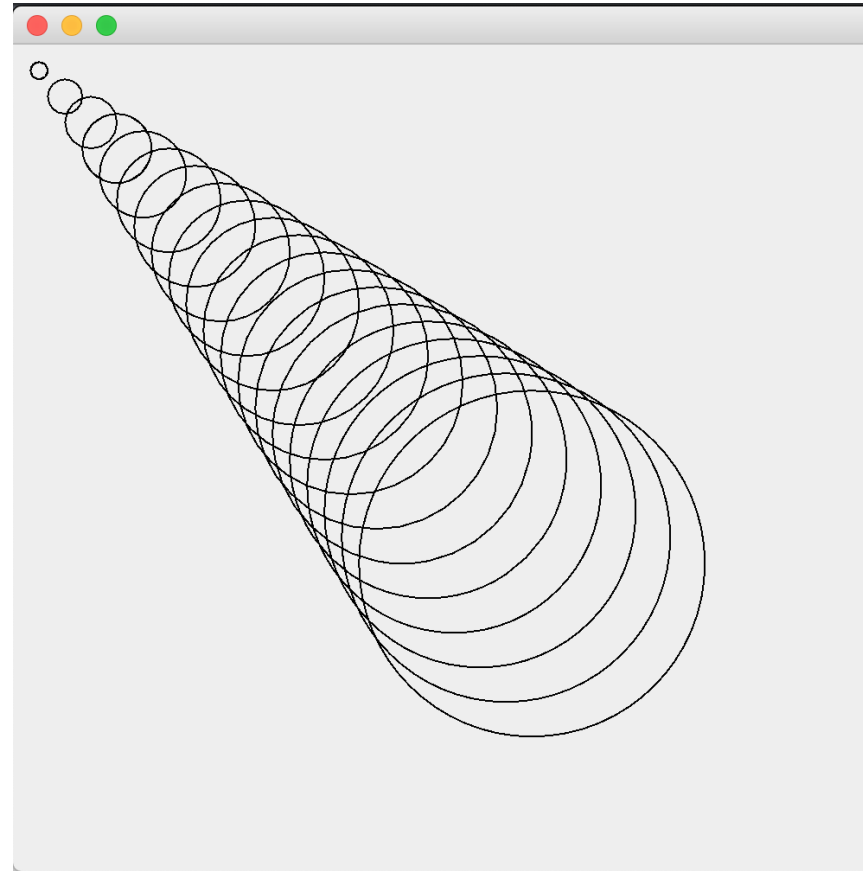
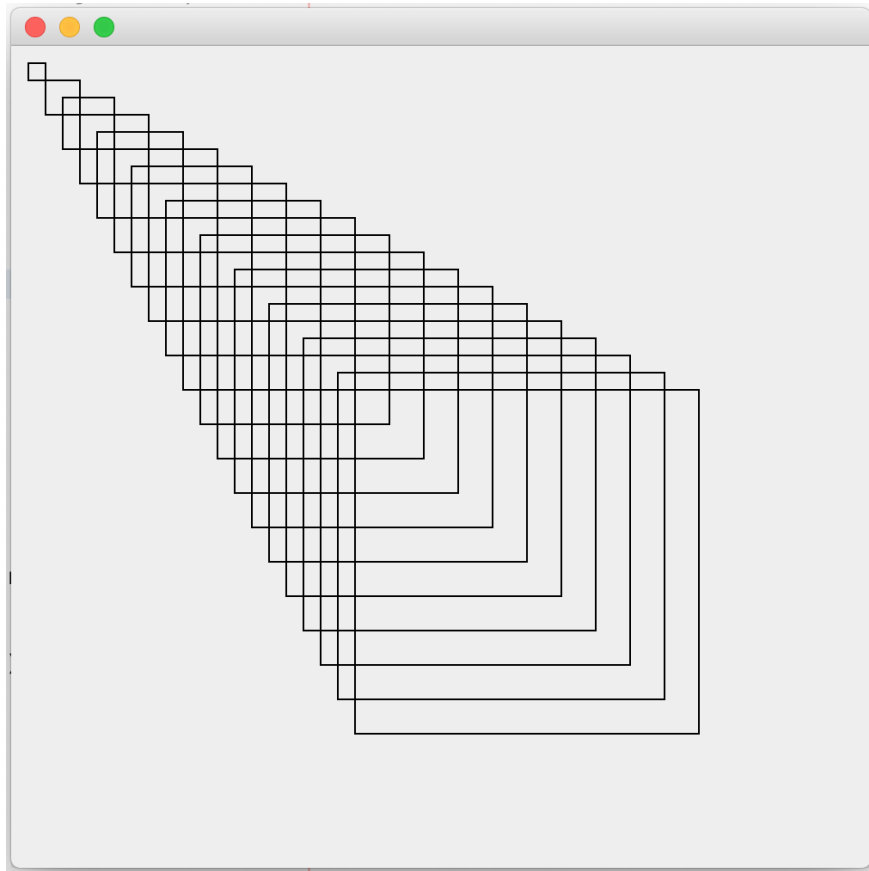
```
}
```





## Exercício 3

- Implemente um programa que gere uma das seguintes saídas



# Variáveis Compostas e Subprogramação

Orientação a Objetos – DCC025

Gleiph Ghiotto Lima de Menezes

[gleiph.ghiotto@ufjf.br](mailto:gleiph.ghiotto@ufjf.br)