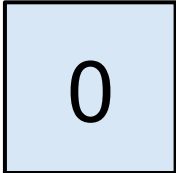


# PROCEDIMENTOS E REFERÊNCIAS

# REFERÊNCIAS

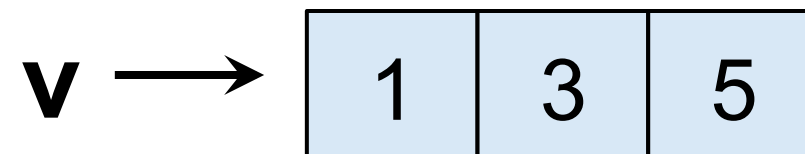
- As variáveis de tipo **primitivo** (i.e., `int`, `double`, `boolean`, ...) guardam valores

```
int i = 0;
```

**i** 

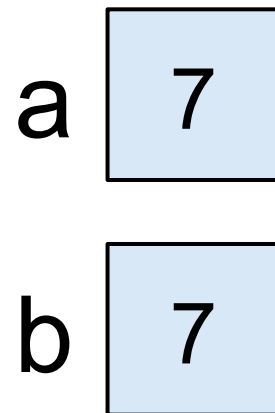
- As variáveis de tipo de **referência** guardam um apontador para onde a instância deste tipo está na memória do computador (p.e., um vetor)

```
int[] v = new int[3];  
v[0] = 1;  
v[1] = 3;  
v[2] = 5;
```



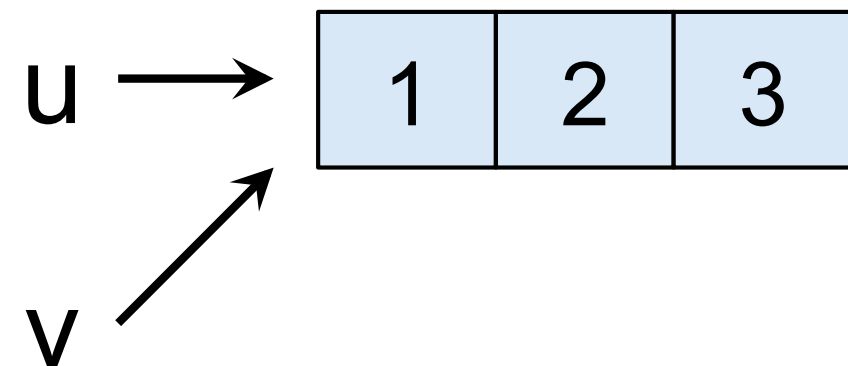
# ATRIBUIÇÃO: VALOR VS REFERÊNCIA

```
int a = 7;  
int b = a;
```



```
int[] u = new int[3];  
u[0] = 1;  
u[1] = 2;  
u[2] = 3;
```

```
int[] v = u;
```



# REFERÊNCIAS INDEFINIDAS (*NULL*)

- Ao contrário das variáveis de tipo primitivo, as quais têm necessariamente um valor atribuído, as referências podem estar indefinidas

```
int[] z = null;
```

z



- Caso um programa execute uma instrução que manipula instâncias de tipos de referência (p.e., vetores) através de uma referência a `null` irá terminar devido a um erro de `NullPointerException`

```
z[0] = 7;
```



`NullPointerException!`

# PROCEDIMENTOS

- Um procedimento potencialmente **altera** as instâncias de **tipos de referência** passadas como argumentos (p.e., vetores)

```
static void procedure(int[] vetor) {  
    vetor[0] = 512;  
  
    ...  
}
```

- Tipicamente, os procedimentos **não devolvem nada**, e logo, o tipo de devolução é definido como sendo vazio (`void`)
- Apesar de não ser devolvido nenhum valor, é possível utilizar a **instrução return** (sem qualquer valor à frente), o que faz com que a **execução termine**

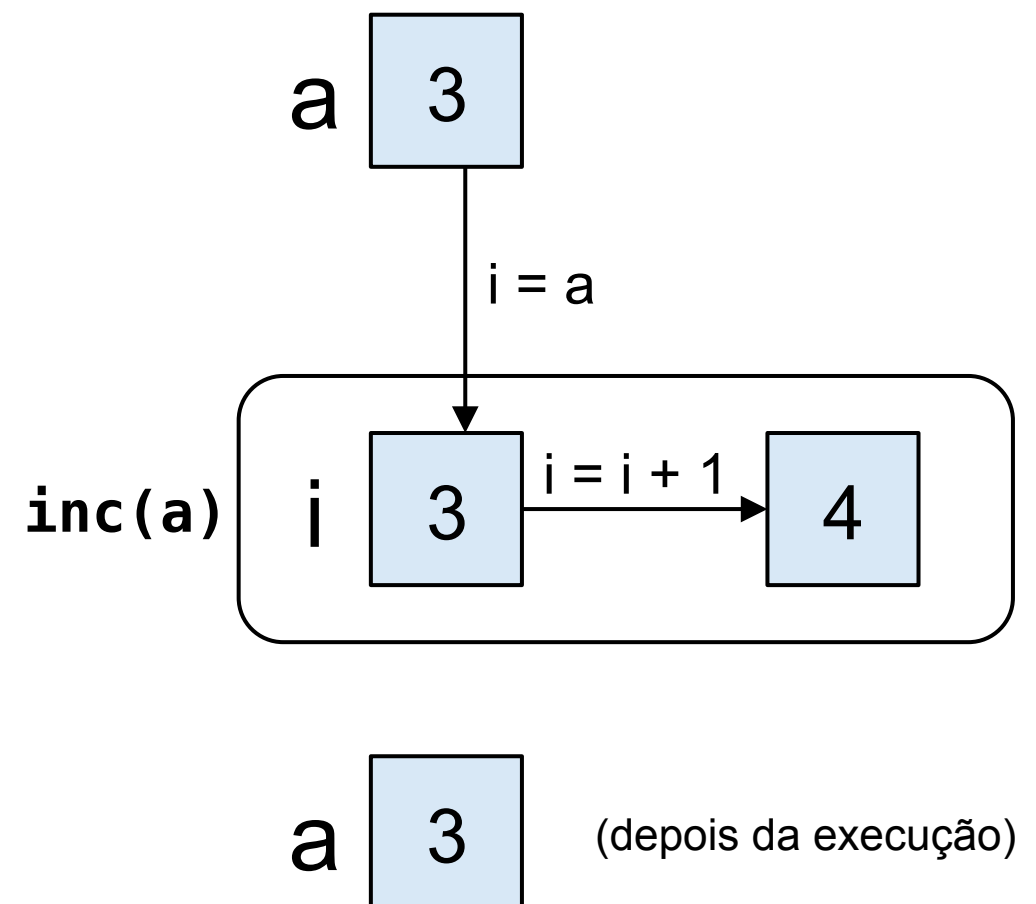
# INVOCÇÃO: PASSAGEM POR VALOR

```
static void inc(int i) {  
    i = i + 1;  
}
```

...

```
int a = 3;  
inc(a);
```

...



# INVOCÇÃO: PASSAGEM DE REFERÊNCIAS

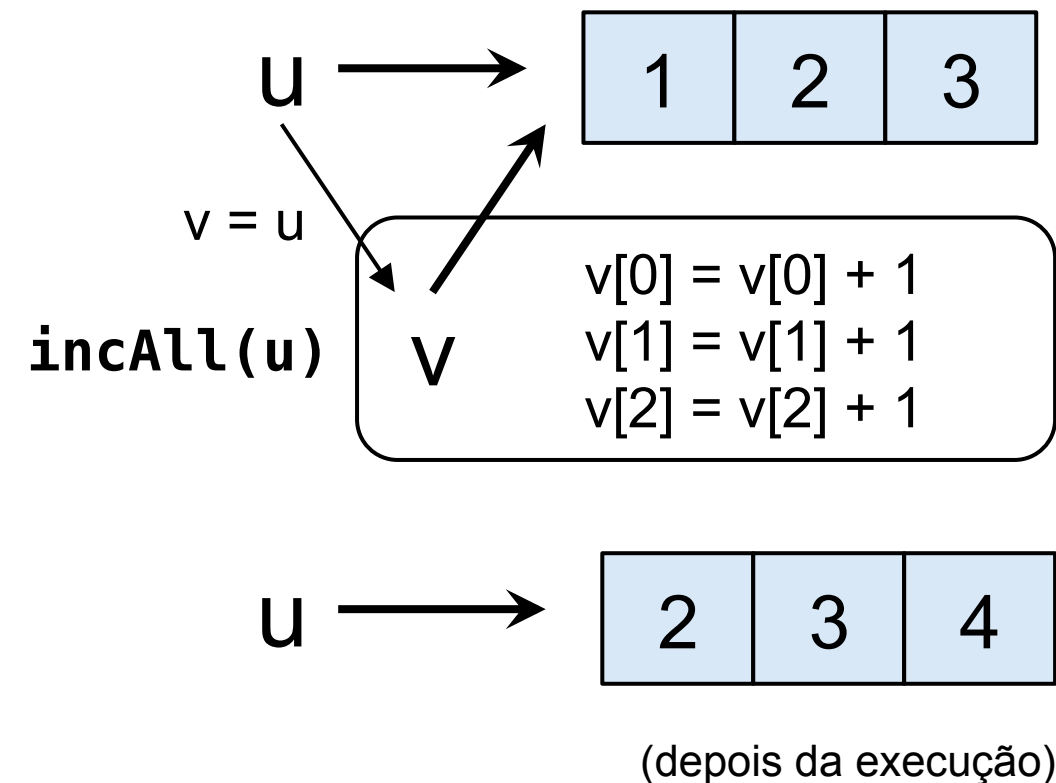
```
static void incAll(int[] v) {  
    int i = 0;  
    while(i != v.length) {  
        v[i] = v[i] + 1;  
        i = i + 1;  
    }  
}
```

...

```
int[] u = new int[3];  
u[0] = 1;  
u[1] = 2;  
u[2] = 3;
```

```
incAll(u);
```

...



# SIMPLIFICAÇÕES SINTÁTICAS

- **Formas sintáticas alternativas** de uma linguagem de programação que visam **facilitar a leitura e escrita** de código (por humanos)
  - Instruções simplificadas, geralmente requerendo **menos texto**
  - Não **acrescentam funcionalidade** à linguagem
  - A melhor opção entre alternativas sintáticas é uma matéria subjectiva



# SIMPLIFICAÇÃO SINTÁTICA: CRIAÇÃO DE VETORES

- É possível criar um vetor fornecendo os seus elementos diretamente

```
int[] v = new int[3];  
v[0] = 0;  
v[1] = 2;  
v[2] = 4;
```

=

```
int[] v = {0, 2, 4};
```

# SIMPLIFICAÇÃO SINTÁTICA: ARITMÉTICAS

```
i = i + 1;
```

=

```
i++;
```

```
j = j - 1;
```

=

```
j--;
```

```
s = s + p;
```

=

```
s += p;
```

Análogo para os operadores

`-=`

`*=`

`/=`

`%=`

# SIMPLIFICAÇÃO SINTÁTICA: O CICLO *FOR*

- O ciclo **for** é uma estrutura de repetição alternativa ao ciclo **while**, consistindo numa simplificação sintática especialmente adequada para **iterações** (p.e. sobre vetores)

```
int i = 0;  
while(i != expr) {  
    ...  
    i++;  
}
```

=

```
for(int i = 0; i != expr; i++) {  
    ...  
}
```

# SIMPLIFICAÇÃO SINTÁTICA:

## *while* ***VS*** *for* — EXEMPLO

- Somar todos os elementos de um vetor de inteiros *v*

```
int soma = 0;
int i = 0;
while(i != v.length) {
    soma += v[i];
    i++;
}
```

=

```
int soma = 0;
for(int i = 0; i != v.length; i++) {
    soma += v[i];
}
```

# SIMPLIFICAÇÃO SINTÁTICA: EXPRESSÃO CONDICIONAL

- Útil quando o valor a atribuir a uma variável depende de determinada condição

```
int max = 0;  
if(a < b) {  
    max = b;  
} else {  
    max = a;  
}
```

=

```
int max = a < b ? b : a;
```

# SIMPLIFICAÇÃO SINTÁTICA: BLOCOS DE INSTRUÇÕES

- No caso de um bloco de instruções conter apenas uma instrução, as chavetas podem ser omitidas

```
while(i != 10) {  
    i++;  
}
```

=

```
while(i != 10)  
    i++;
```

```
if(a < b) {  
    max = b;  
} else {  
    max = a;  
}
```

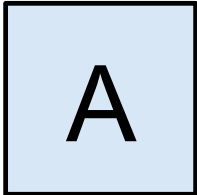
=

```
if(a < b)  
    max = b;  
else  
    max = a;
```

# TIPO PRIMITIVO *CHAR* (CARACTERES)

- Uma variável do tipo char **guarda um carácter**
- Os caracteres são representados em Java **entre plicas** ( ' )

```
char c = 'A';
```

**c** 

# REPRESENTAÇÃO NUMÉRICA DE CARACTERES

- Cada carácter tem um **valor numérico inteiro** correspondente
- Os valores são **consecutivos** de acordo com a ordem alfabética (os conjuntos de minúsculas e maiúsculas estão separados)
- Os caracteres podem ser **manipulados como inteiros** (e.g., usando os operadores  $<$ ,  $>$ ,  $+$ ,  $-$ ,  $++$ ,  $--$ )

CARÁTER	VALOR
...	...
,	96
a	97
b	98
c	99
...	...



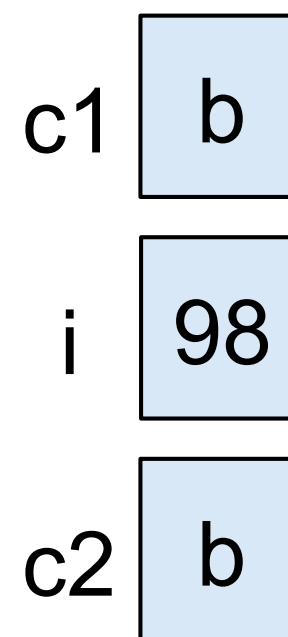
# TIPO PRIMITIVO *CHAR* (CARACTERES)

- Um carácter pode ser convertido para o seu valor numérico, e vice-versa

```
char c1 = 'b';
```

```
int i = (int)c1;
```

```
char c2 = (char)i;
```



# A RETER

- Referências
  - Atribuição: valor vs referência
  - Referências indefinidas
- Procedimentos
  - Passagem por valor
  - Passagem de referência
- Simplificações sintáticas
- O tipo primitivo char

