

# Estrutura de Dados

ES-01



## Alocação de Memória

---

Ao carregar um programa, o sistema operacional disponibiliza um espaço de endereçamento

---

Esse espaço é a memória disponível para o programa

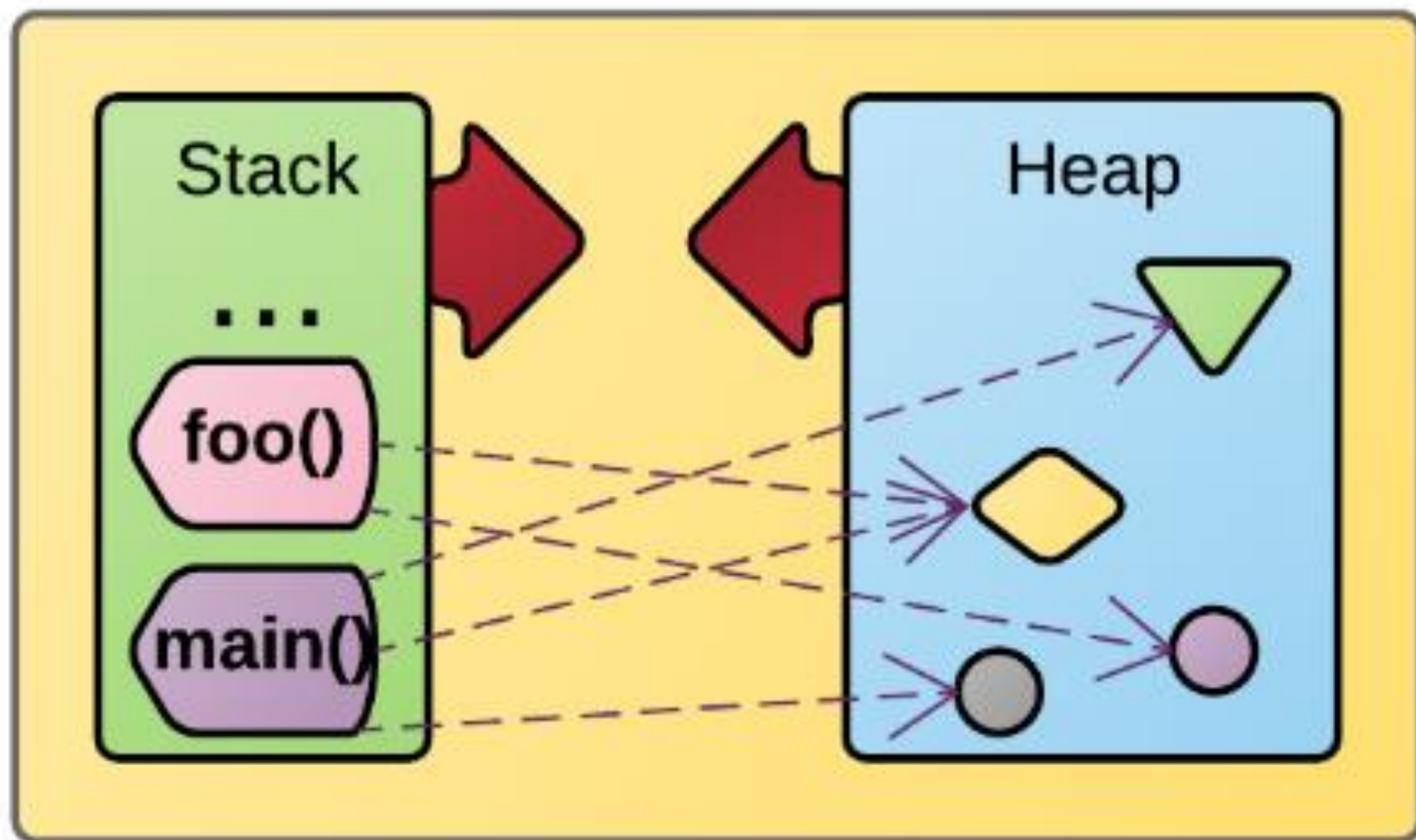
# Tipos de Alocações disponíveis ao programador

- O Heap, ou área de alocação dinâmica
  - Espaço reservado para variáveis e dados criados em tempo de execução;
  - É a memória global do programa.
- Pilha de funções (stack)
  - Área da memória que aloca dados/variáveis ou ponteiros quando uma função é chamada e desalocada quando a função termina
  - Representa a memória local à aquela função/programa
  - Todos os dados são alocados no início da execução da aplicação/função.

# Uso da memória

- Existem 3 maneiras de reservar espaço da memória:
  - Variáveis globais (estáticas)
    - Espaços que existem enquanto o programa estiver executando
  - Variáveis locais
    - Espaço existe enquanto a função que declarou estiver executando
  - Espaços dinâmicos
    - Espaços que existe até ser explicitamente liberado

# Espaço de endereçamento

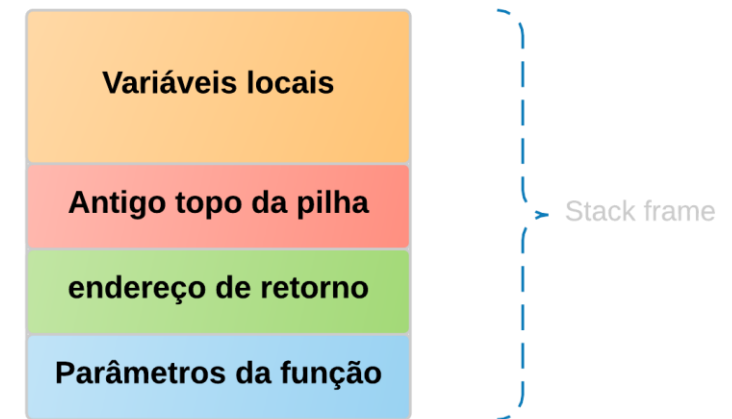


# Stack

- É uma área disponibilizada dentro do espaço de endereçamento do processo;
- Quando uma função é chamada durante a execução de um programa, um bloco de memória é empilhado no topo da pilha de funções;
- Nesse bloco existem referências para todas as variáveis criadas ou apontadas dentro da função chamada;
- Ao término da execução da função, esse bloco é desempilhado/desalocado.

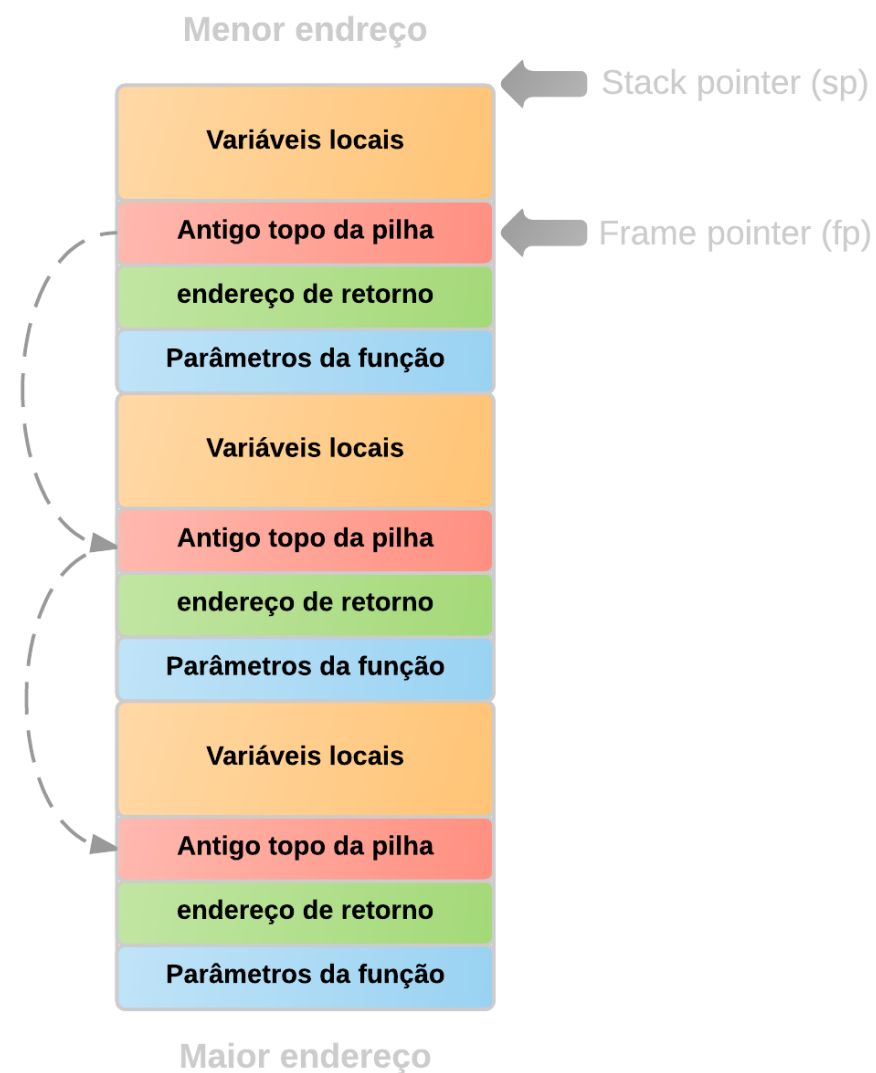
# Funcionamento da Pilha

- Um bloco alocado na pilha representa o conjunto de variáveis locais àquela função
- Cada bloco empilhado chamando de Stack Frame
- Dentro de cada frame/bloco temos
  - variáveis para os parâmetros passados para a função, o
  - endereço de retorno (para onde a instrução *return* aponta)



# Funcionamento da Pilha

- para cada função chamada empilhamos um *stack frame* no topo da pilha de funções
- a pilha sempre começa no maior endereço reservado no espaço de endereçamento para a pilha e vai crescendo no sentido dos menores endereços.





# Heap



Área de alocação dinâmica



Ao desalocar memória do heap, a área volta a está disponível para novas alocações

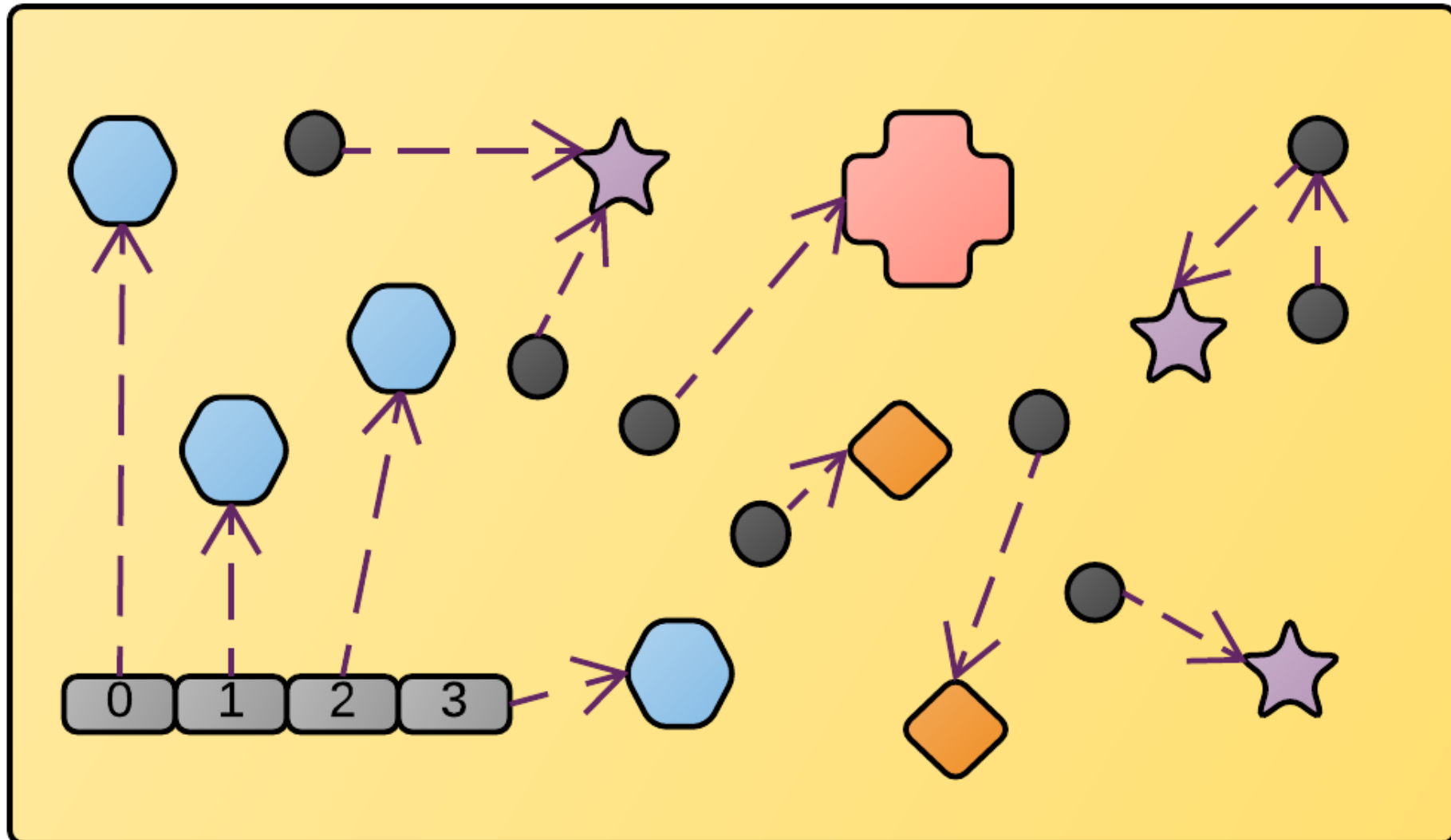



À medida que muitas alocações/desalocações ocorrem no Heap ele sofre muita fragmentação



Isso gera impacto na performance e na eficiência de como o programa aloca memória

# Heap





# Comparação: Velocidade de Acesso

- Stack:
  - O acesso a variáveis alocadas são extremamente rápidos
  - dependem apenas de um deslocamento de ponteiros,
- Heap:
  - acesso é relativamente baixo
  - depende muito do *runtime* da linguagem e da biblioteca que faz alocação



# Comparação: Escopo

## Heap

- Variáveis alocadas são desalocadas através de uma instrução explícita do programa

## Stack

- variáveis alocadas são desalocadas quando a função retorna

# Comparação: Gerenciamento de memória

## Stack:

- Eficientes e não fragmentados

## Heap:

- Fragmenta a memória e pode levar um mal aproveitamento dos espaço de enredoçamento

# Comparação: Limite

## Stack:

- Possui limite de crescimento
- Varia de acordo com a linguagem na qual foi escrito
- Chamadas recursivas podem rapidamente estourar a pilha de funções de uma linguagem

## Heap:

- Crescimento dinâmico
- Quando não há mais espaço alocável no *Heap*, é solicitado ao sistema operacional através de uma chamada de sistema ([\*system call\*](#))
- o crescimento do *Heap* fica à cargo dos limites impostos e controlados pelo sistema operacional.

# Leitura da Comunidade de Devs...

<https://pt.stackoverflow.com/questions/3797/o-que-s%C3%A3o-e-onde-est%C3%A3o-a-stack-e-heap>

# Ponteiros

- Ponteiros são variáveis utilizadas para apoiar a alocação dinâmica.
- Elas armazenam um endereço de memória que é geralmente ocupado por um dado (variável) de um determinado tipo;
- Ponteiro tem um tipo associado que determina o tipo do dado que ele é capaz de apontar, isto é, o tipo da variável que seria encontrada no endereço apontado
- Quando utilizado para variáveis um ponteiro pode ser um ponteiro para um tipo pré-definido da linguagem ou um tipo definido pelo programador



# Ponteiros

- As variáveis do tipo ponteiro são armazenadas no segmento de dados junto com outras variáveis estáticas do programa.
- Como um ponteiro armazena apenas um endereço de memória, o seu tamanho em bytes é o tamanho necessário para armazenar tal endereço: em geral são usados 4 bytes (o tamanho de um endereço de memória no computador)
- Quando um ponteiro contém o endereço de uma variável, dizemos que o ponteiro está “apontando” para essa variável
- Além de apontar (referenciar) para variáveis, ponteiros podem ser usados para criar, controlar e destruir estruturas dinâmicas de um determinado tipo.



Exemplos...

# Ponteiros(operadores e funções)

---

Operação/Linguagem	Pascal	C
Declarando ponteiro	<code>p : ^tipo;</code>	<code>tipo *p;</code>
Apontando para uma variável(estática)	<code>p := @v;</code>	<code>p = &amp;v;</code>
Acessando estrutura apontada	<code>p^:=2;</code>	<code>*p=2;</code>
Criando e apontando estrutura dinâmica	<code>new(p);</code>	<code>p=(*int)malloc( sizeof(tipo) );</code>
Destruindo estrutura alocada dinamicamente	<code>dispose(p);</code>	<code>free(p);</code>
Ponteiro recebendo referencia de outro ponteiro	<code>p1:=p2;</code>	<code>p1=p2;</code>

- Objetivo: Usar ponteiros para passar parâmetros para funções. Complete o programa 1. Este programa usa a função void troca (int \*a, int \*b). Esta função troca os valores apontados por a e b.

```
#include <stdio.h>

void troca (int *a, int *b)
{
    int temp;

    temp = *a;
    *a = *b;
    /* ***** Falta um comando aqui */

} /* Fim de troca */

int main (void)
{
    int x, y;

    scanf("%d %d", &x, &y);
    troca(&x, &y);
    printf("Troquei ----> %d %d\n", x, y);
    return 0;
}
```



# Ponteiros em Java ?

# Ponteiros em Java ?

- As variáveis de referência a objetos – esse tipo de variável armazena o endereço de memória onde foi instanciado um objeto, ou seja, é um ponteiro para um objeto.
- A dúvida se a linguagem Java utiliza, ou não, ponteiros vêm do fato da mesma não possuir um recurso conhecido da Linguagem C/Pascal: A Aritmética de Ponteiros.
  - Em C, um programador pode acessar livremente a área de memória que quiser, basta informar ao ponteiro o endereço inteiro que representa uma determinada área.

# Referencia em java (analogia)

```
Dog d = new Dog( );  
d.bark( );
```

considere isso

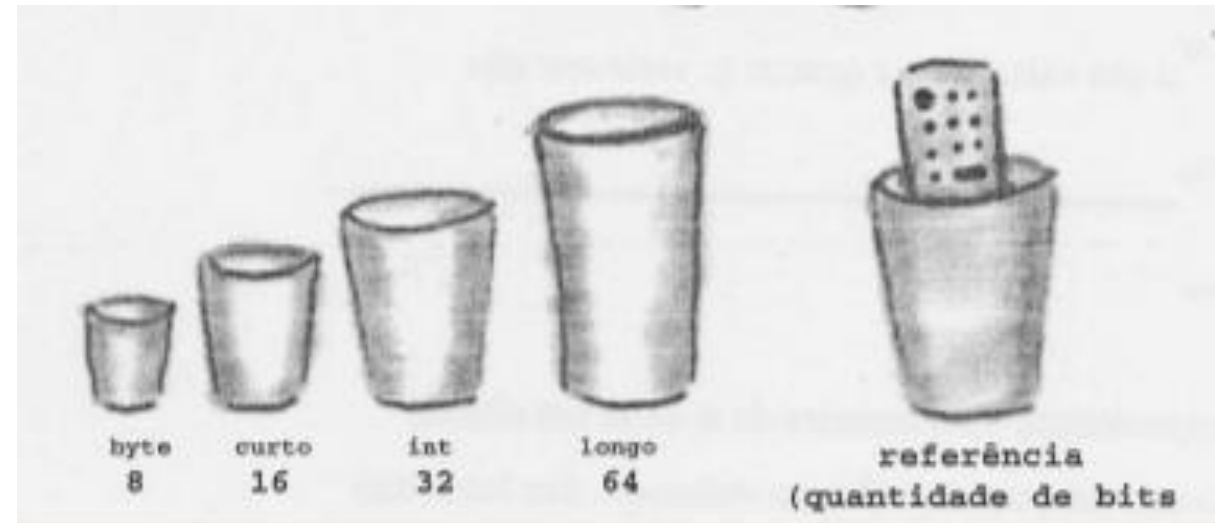
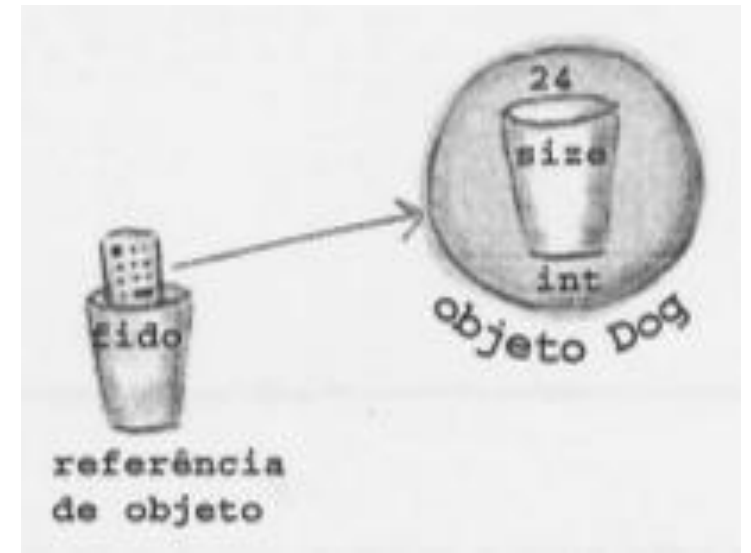
como se fosse isso



Pense na variável de referência de Dog como o controle remoto de um objeto Dog.

Você a usará para acessar o objeto e fazer algo (chamar métodos).

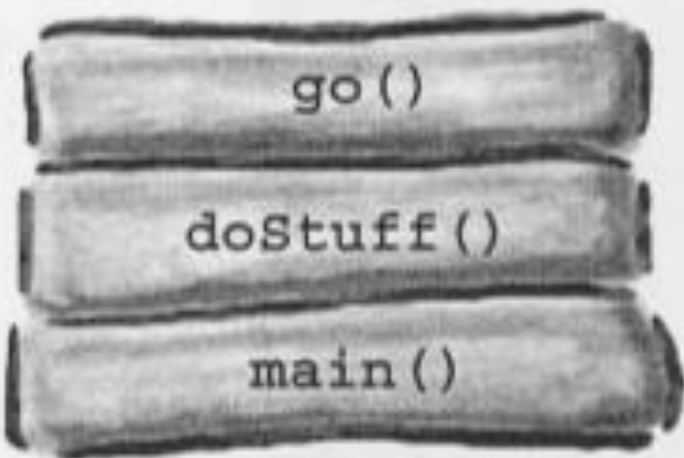
# Referencia em Java (analogia)





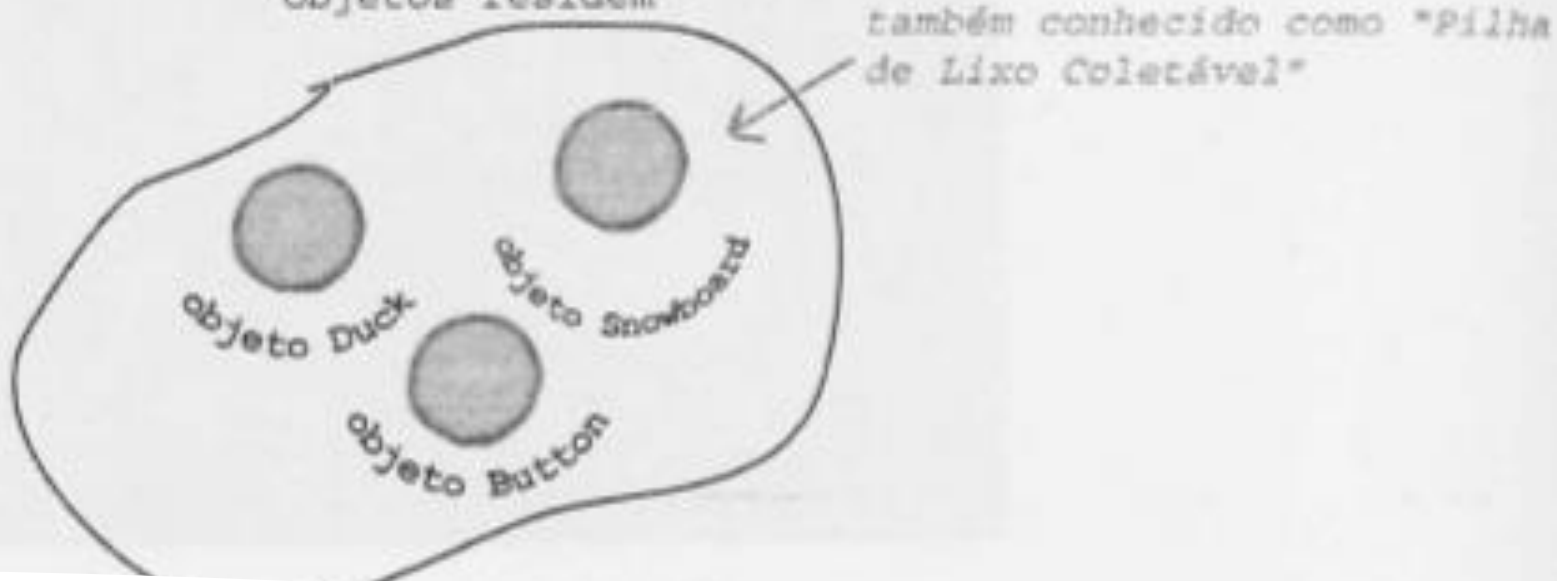
## A Pilha

Onde as chamadas de método e as variáveis locais residem



## O Heap

Onde todos os objetos residem



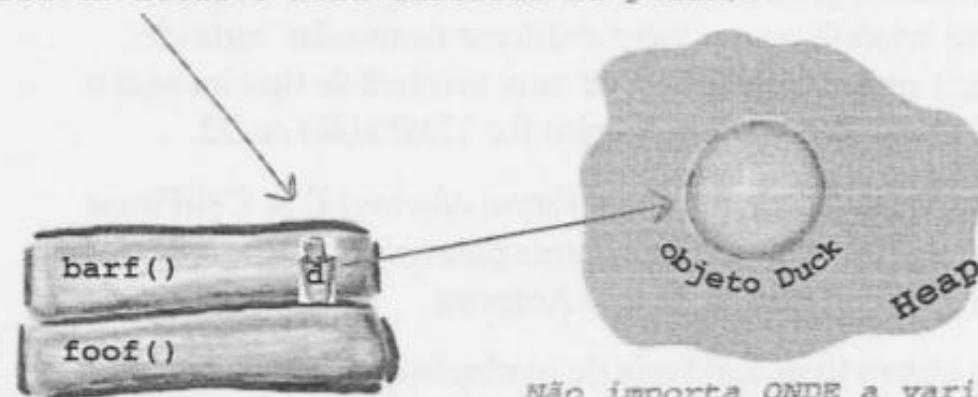
# Variáveis em Memória no Java

- Variáveis locais
  - Declaradas dentro de um método, inclusive parâmetros
  - Existem enquanto o método está na pilha (stack)
- Variáveis de Instância
  - Declaradas dentro de uma classe mas não de um método
  - Residem dentro do objeto que a pertencem (heap)

# E se a variável local for a referencia a um objeto?

```
public class StackRef {  
    public void foof() {  
        barf();  
    }  
  
    public void barf() {  
        Duck d = new Duck(24);  
    }  
}
```

*barf()* declara e cria uma nova variável de referência 'd' do objeto Duck (já que ela é declarada dentro do método, é uma variável local e será inserida na pilha).

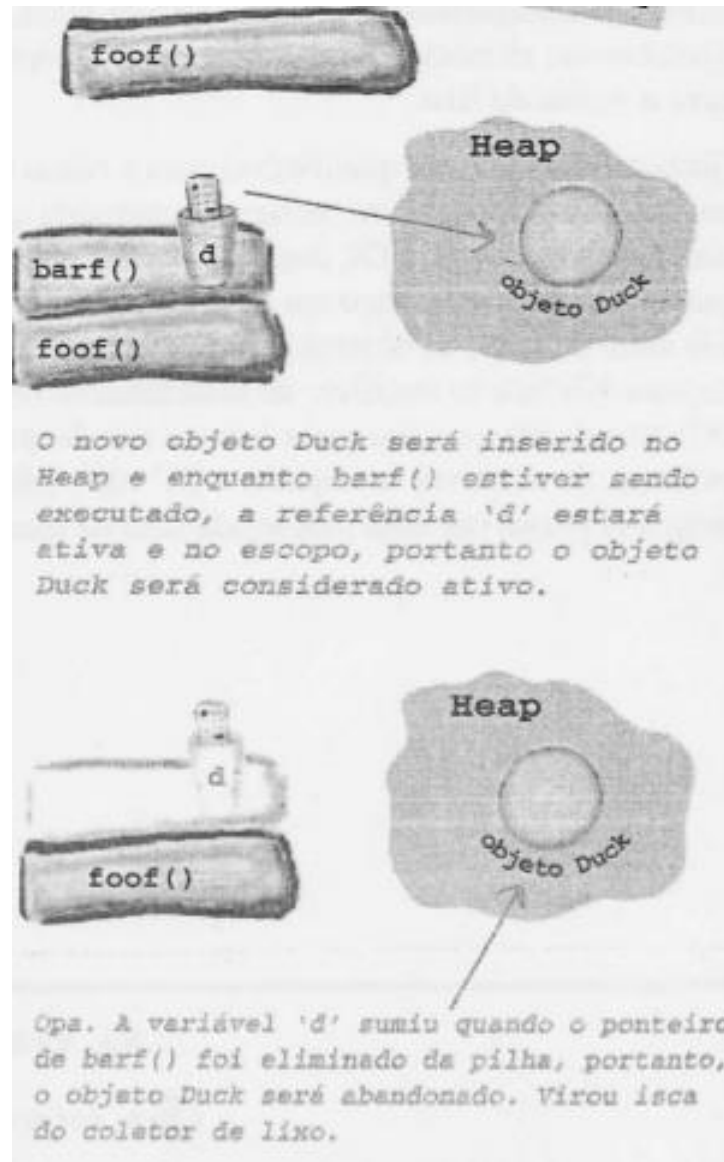


Não importa ONDE a variável de referência do objeto for declarada (dentro de um método ou como a variável de instância de uma classe) o objeto sempre será inserido no heap.

# Coletor de Lixo (Garbage Collector)

- é um processo usado para a automação do gerenciamento de memória.
- Com ele é possível recuperar uma área de memória inutilizada por um programa, o que pode evitar problemas de vazamento de memória, resultando no esgotamento da memória livre para alocação.
- Esse sistema contrasta com o gerenciamento manual de memória, em que o programador deve especificar explicitamente quando e quais objetos devem ser desalocados e retornados ao sistema.
- Entretanto, muitos sistemas usam uma combinação das duas abordagens(C++).

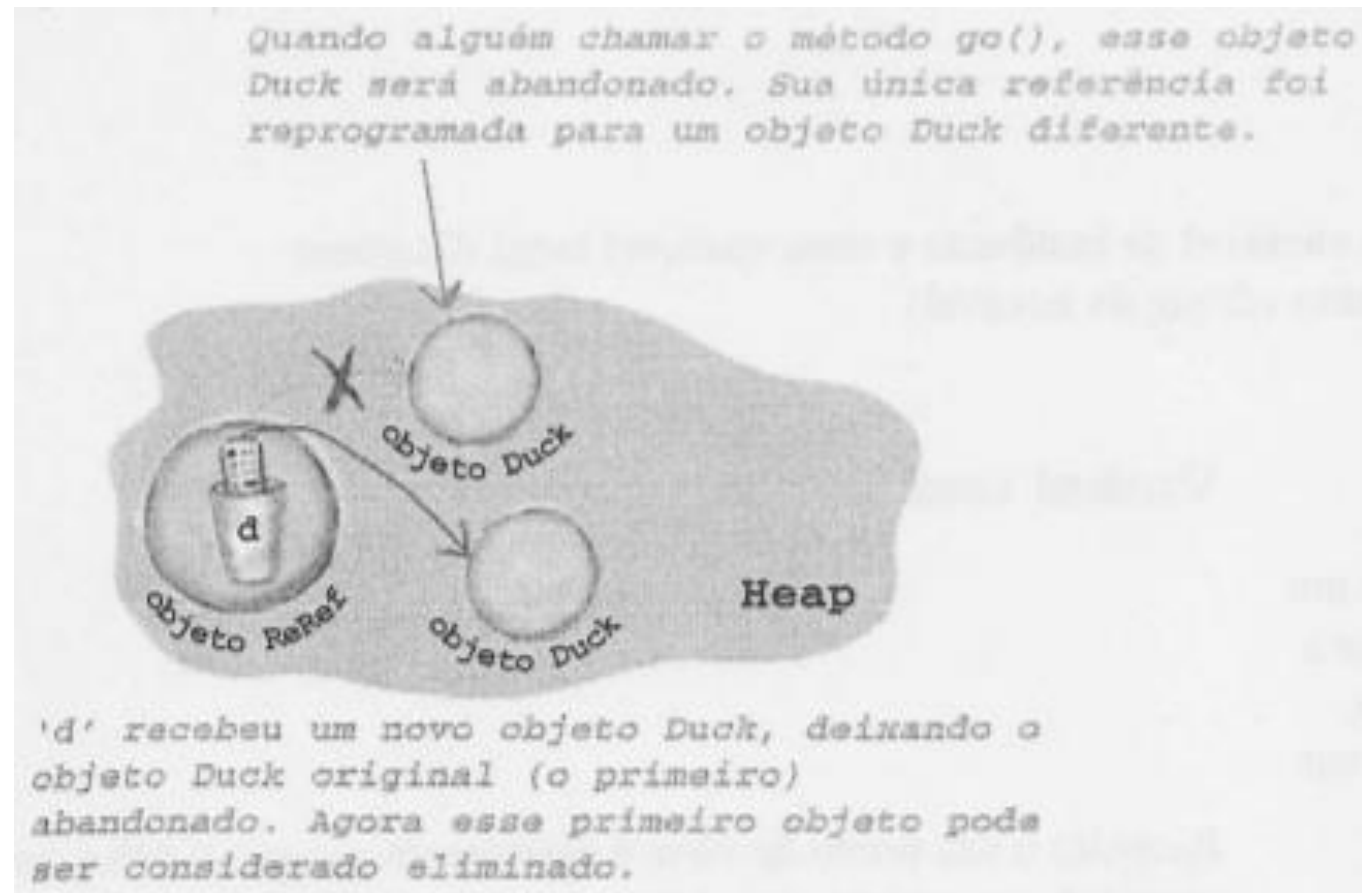
## Onde o Coletor de Lixo atua?



```
public class StackRef {  
    public void foof() {  
        barf();  
    }  
  
    public void barf() {  
        Duck d = new Duck();  
    }  
}
```

- 1º Caso
  - A referência sai do escopo permanentemente
  - Ex: a execução de barf() é concluída e o método é eliminado

## Onde o Coletor de Lixo atua?

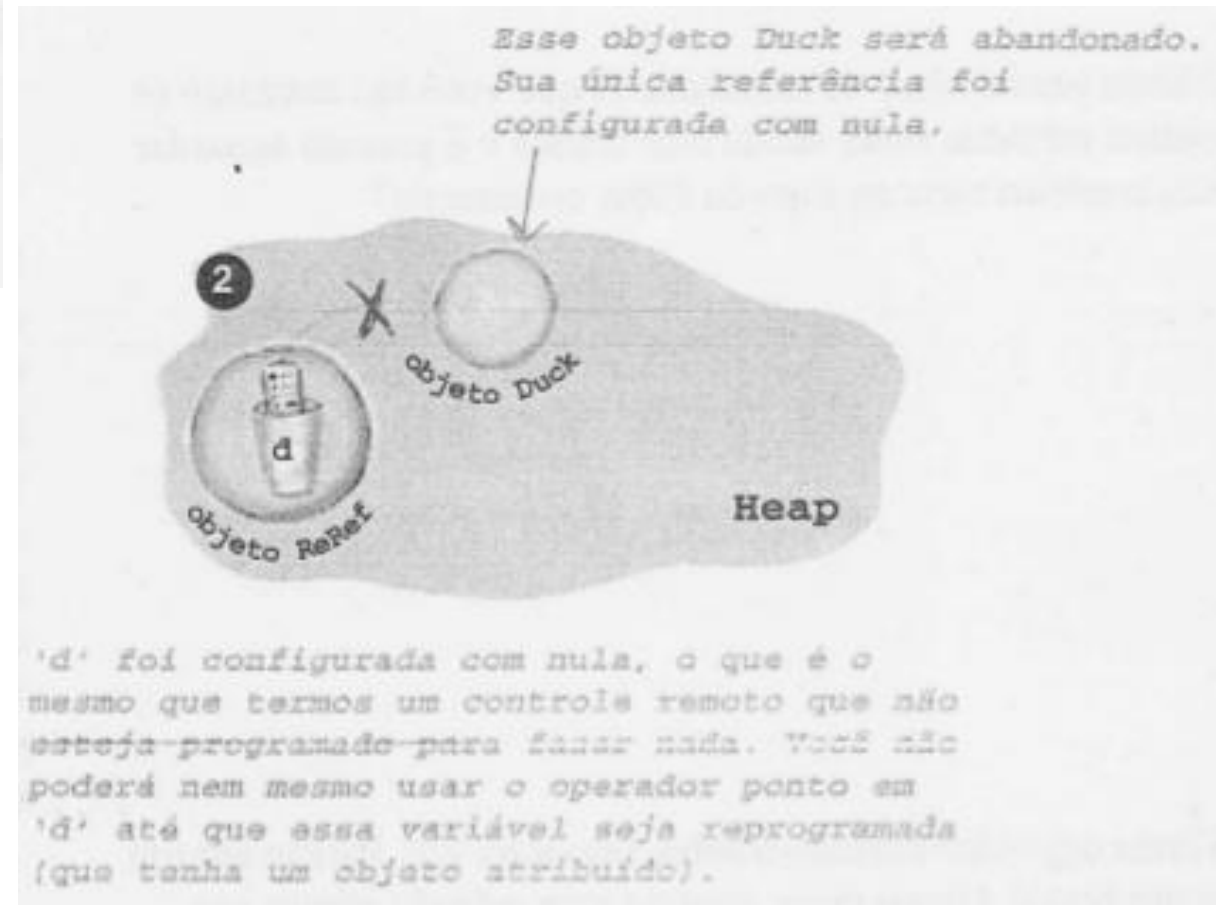


```
public class ReRef {  
    Duck d = new Duck();  
  
    public void go() {  
        d = new Duck();  
    }  
}
```

- 2º Caso:
  - A referencia é atribuída a outro objeto

## Onde o Coletor de Lixo atua?

```
public class ReRef {  
  
    Duck d = new Duck();  
  
    public void go() {  
        d = null;  
    }  
}
```



- Caso 3
  - Declarar explicitamente com uma referencia nula

# Estudo Complementar

- Garbage Collection: o GC do Java
  - <http://javafree.uol.com.br/artigo/1386/Garbage-Collection.html>
- How does garbage collection work?
  - <http://chaoticjava.com/posts/how-does-garbage-collection-work/>
- Coletor de Lixo Wikipedia
  - [http://pt.wikipedia.org/wiki/Ponteiro %28programa%C3%A7%C3%A3o%29](http://pt.wikipedia.org/wiki/Ponteiro_%28programa%C3%A7%C3%A3o%29)