ChatGPT

Passagem de Parâmetros em Java: Valor vs Referência

Passagem por Valor vs Passagem por Referência

Em ciência da computação, **passagem por valor** significa que uma função recebe **uma cópia do valor** do argumento. Ou seja, a função chamada opera em uma variável local (cópia) independente da variável original; modificações nessa cópia **não afetam** o valor do chamador [1] [2]. Já **passagem por referência** significa fornecer à função chamada **um acesso direto ao local de memória** do argumento original, sem copiar seu valor [3]. Nesse modo, a função pode modificar diretamente o dado do chamador, pois **ambos compartilham o mesmo slot de memória** para aquele parâmetro [3].

É importante destacar que **Java adota somente passagem por valor** em todas as situações [4]. Não existe passagem por referência "verdadeira" em Java, no sentido usado em C++ ou outras linguagens. Em outras palavras, *sempre* que um método Java é chamado, o que se passa são cópias dos valores dos argumentos – sejam eles primitivos ou referências para objetos [2]. Como afirmado em uma explicação clássica: *"The Java programming language does not pass objects by reference; it passes object references by value"* [5]. Portanto, apesar da sintaxe simples ao chamar métodos, é fundamental entender o que está acontecendo "por baixo dos panos".

Tipos Primitivos vs. Objetos: Como o Java Trata Cada Caso

No caso de **tipos primitivos** (por exemplo, `int`, `double`, `boolean`), a passagem de parâmetro é direta: o valor literal da variável é copiado para o parâmetro do método. Assim, se alterarmos o valor dessa cópia dentro do método, isso não terá efeito algum fora dele, pois estamos manipulando apenas a cópia local. Por exemplo:

```
void alterarValor(int valor) {
    valor = 20;
}

int x = 10;
System.out.println("Antes: " + x);
alterarValor(x);
System.out.println("Depois: " + x);
```

Saída esperada:

```
Antes: 10
Depois: 10
```

Como mostrado acima, `x` permanece 10 após a chamada, já que o método `alterarValor` trabalhou sobre uma cópia de `x` [6].

Já para **objetos** (instâncias de classes, arrays, etc.), a situação merece cuidado. Em Java, variáveis de tipos não-primitivos **não armazenam o objeto em si**, mas sim uma **referência** (um ponteiro/endereço) para o objeto alocado na memória heap [7]. Quando passamos um objeto como argumento, na verdade o que é passado é **o valor dessa referência** – ou seja, uma cópia do endereço do objeto [8] [9]. **O objeto em si não é copiado** na chamada de método [8]. Em vez disso, tanto o chamador quanto o método chamado passam a ter referências (distintas) que apontam para o **mesmo** objeto na heap [10] [11].

Isso explica por que, em Java, métodos podem alterar o estado interno de objetos passados como parâmetro e essas alterações serem observadas pelo chamador. Ambos os nomes (a variável original e o parâmetro do método) referenciam o mesmo objeto em memória. Por exemplo:

```
void setNome(Pessoa p) {
    p.nome = "Alice";
}

Pessoa pessoa = new Pessoa("João");
setNome(pessoa);
System.out.println(pessoa.nome); // Imprime "Alice"
```

No exemplo acima, o método `setNome` recebe uma cópia da referência para o objeto `Pessoa` e modifica um de seus campos. Como o objeto é compartilhado entre chamador e chamado, o nome passa a ser "Alice" após a chamada – uma mudança visível fora do método [12]. Em outras palavras, *Java passa referências de objetos por valor*, o que muitas vezes leva à ideia equivocada de que objetos seriam passados por referência. Na verdade, a referência (endereço) é o valor que é copiado e passado ao método, representando o objeto na memória [2].

Stack vs. Heap: Onde Variáveis e Objetos Residem

Para entender melhor o modelo, lembremos da organização de memória em Java. A JVM utiliza principalmente duas áreas em tempo de execução: **stack (pilha)** e **heap**.

- A **stack** é usada para armazenar variáveis locais, parâmetros e informações de chamadas de métodos (como o endereço de retorno) [13]. Sempre que um método é invocado, a JVM aloca um novo bloco na stack para suas variáveis e parâmetros. Esses valores existem apenas durante a execução do método e a alocação/desalocação na pilha segue a ordem LIFO (Last-In-First-Out, ou último a entrar, primeiro a sair) [14]. Tipos primitivos são armazenados diretamente na stack, bem como as referências para objetos (que ocupam alguns bytes representando um endereço) [15].

- O **heap** é a região onde os objetos são alocados de fato (via `new`) e de onde são liberados pelo garbage collector. Sempre que criamos um objeto, ele é colocado no heap, e variáveis de referência (na stack ou em campos de outros objetos) guardam o endereço desse objeto [16]. Qualquer objeto no heap pode ser acessado por qualquer parte do programa que possua uma referência a ele, enquanto cada thread possui sua própria stack – o conteúdo da pilha de uma thread não é acessível por outras threads [17].

Por exemplo, considere o código:

```
int a = 42;                 // (1) alocado na stack do método atual
Objeto obj = new
Objeto();   // (2) objeto alocado no heap, referência obj na stack
metodo(obj);                // (3) chamada de método com obj como argumento
```

1. Ao executar a linha (1), a variável primitiva `a` é criada no frame de stack do método atual e recebe o valor `42`.
2. Na linha (2), um novo `Objeto` é instanciado no heap. Na stack, é criada a variável de referência `obj` que guarda o endereço desse objeto recém-criado (por exemplo, `0xABCD1234`).
3. Ao chamar o método na linha (3), a JVM cria um novo frame de stack para `metodo`. O parâmetro deste método, digamos `param`, receberá **uma cópia do valor de referência** contido em `obj` – isto é, terá também o endereço `0xABCD1234` [18]. Agora, tanto `main` quanto `metodo` possuem referências (independentes) apontando para o mesmo objeto no heap.

"Passar uma cópia da referência": O que Isso Significa?

Diz-se que Java passa objetos "por referência" apenas no sentido de que passa **referências por valor**. A expressão "cópia da referência" significa exatamente isso: o conteúdo da variável de referência (um identificador/endereço do objeto na memória) é duplicado e entregue ao método chamado [9]. O método invocado recebe uma referência que **aponta para o mesmo objeto** que o argumento original, mas trata-se de uma variável de referência separada, armazenada em sua própria frame de stack [19].

Em outras palavras, o parâmetro do método e o argumento do chamador contêm valores (endereços) idênticos que referenciam o mesmo objeto no heap [19]. Qualquer operação que desreferencie essa referência e altere o objeto (por exemplo, modificar um atributo ou chamar um método do objeto) afetará o mesmo objeto compartilhado, tornando a mudança visível ao acessar o objeto via outra referência [11]. Porém, se dentro do método a variável de referência for alterada para apontar para outro objeto ou receber `null`, essa nova atribuição *não terá efeito* sobre a variável original do chamador [11] – afinal, estamos mudando apenas a cópia local da referência, e não a referência que existe no escopo do chamador.

Modificando Objetos vs. Reatribuindo Referências

Devido a esse comportamento, há uma diferença crucial entre **modificar o conteúdo de um objeto** passado como parâmetro e **reatribuir a referência** do parâmetro para um novo objeto.

- **Modificar o objeto (via referência):** Se um método altera o estado interno do objeto (por exemplo, modificando os campos de um objeto ou os elementos de um array), o chamador verá essas alterações, pois método e chamador compartilham o mesmo objeto na memória. No exemplo anterior, após chamar `setNome(pessoa)`, o campo `nome` do objeto `Pessoa` referenciado por `pessoa` foi alterado para "Alice" dentro do método, e essa modificação persiste fora dele [12]. Em termos práticos, operações como `parametro.atributo = valor` ou `parametro.metodoAlterador()` dentro da função afetam o objeto original.

- **Reatribuir a referência:** Por outro lado, se o método executar algo como `parametro = new Objeto()`, isso fará com que o parâmetro local `parametro` passe a apontar para um novo objeto no heap, diferente do original. Essa mudança de referência **não se propaga** ao chamador

[20]. Após o retorno do método, a variável original do chamador ainda apontará para o objeto antigo, pois sua referência não foi alterada externamente. Em código:

```
void resetarContador(Contador c) {
    c.valor = 0;           // altera o objeto original referenciado
    c = new Contador();    // c agora referencia um novo objeto (apenas
dentro do método)
    c.valor = 99;          // altera o novo objeto
}

Contador meuCont = new Contador(42);
resetarContador(meuCont);
System.out.println(meuCont.valor);  // imprime 0, não 99
```

No exemplo acima, dentro de `resetarContador` definimos `c.valor = 0`, o que altera o objeto original (tornando `meuCont.valor` igual a 0 externamente). Em seguida, porém, `c = new Contador()` faz `c` referenciar um novo objeto – essa nova referência vale apenas no escopo do método. Quando o método termina, o chamador ainda possui uma referência para o objeto inicial (cujo valor foi zerado), sem saber que dentro do método chegou a existir um novo objeto com valor 99 [21].

Resumindo: **podemos alterar o objeto passado, mas não podemos trocar a referência que o chamador possui para aquele objeto**. Essa distinção frequentemente confunde iniciantes – muitos acham que "Java passa objetos por referência" porque veem efeitos colaterais em objetos mutáveis, mas o que ocorre é que a referência (valor) foi copiada. O objeto é o mesmo, então mudanças nele aparecem fora; porém, tentar redirecionar a referência dentro do método não muda a referência original do chamador [22].

Comparação com Linguagens que Suportam Passagem por Referência

Linguagens como C++ e C# oferecem mecanismos de passagem por referência genuína, o que difere do modelo do Java. Em C++, por exemplo, podemos declarar uma função `void f(int &x)` que recebe uma referência a um inteiro. Nesse caso, `x` dentro da função é apenas um *alias* do inteiro original passado; uma atribuição `x = 26` dentro de `f` modificaria diretamente a variável do chamador. De forma semelhante, C++ permite passar ponteiros para funções e, se manipulados apropriadamente (por exemplo, alterando o valor apontado), é possível obter efeito de passagem por referência.

Já o C# possui a palavra-chave `ref` (e `out`), permitindo semântica de referência em parâmetros. Por exemplo, um método em C# declarado como `void Incrementar(ref int n)` receberá `n` por referência, de modo que `n = n + 1` dentro do método alterará a variável original. Em Java, não há equivalente direto a `ref`. O código a seguir ilustra as diferenças:

- **Em C# (passagem por referência):**

```
void Swap(ref Objeto a, ref Objeto b) {
    Objeto temp = a;
    a = b;
```

```
    b = temp;
}

Objeto o1 = new Objeto("Azul");
Objeto o2 = new Objeto("Vermelho");
Swap(ref o1, ref o2);
// o1 agora referencia o objeto "Vermelho"; o2 referencia "Azul"
```

- **Em Java (tentativa equivalente, mas sem efeito no chamador):**

```
void swap(Objeto a, Objeto b) {
    Objeto temp = a;
    a = b;
    b = temp;
}

Objeto o1 = new Objeto("Azul");
Objeto o2 = new Objeto("Vermelho");
swap(o1, o2);
// o1 e o2 continuam referenciando "Azul" e "Vermelho", respectivamente
```
[23]

No código C#, como `a` e `b` são referências aos originais, a troca persiste fora do método. Em Java, `a` e `b` são cópias de referências; a troca ocorre apenas nessas cópias locais, sem afetar as variáveis `o1` e `o2` no escopo do chamador [23]. De fato, se tentarmos implementar uma função de swap em Java como acima, veremos que ela **não funciona** para alterar os objetos externos, o que comprova a natureza "*sempre por valor*" da passagem de parâmetros em Java. Em resumo, *Java não oferece passagem por referência real*, ao contrário de C++ (com referências/ponteiros) ou C# (`ref` / `out`) [24]. Essa decisão de projeto simplifica a linguagem e previne certas classes de erros, embora ocasionalmente exija abordagens alternativas para se alcançar resultados semelhantes.

Exemplo Prático com Diagrama de Memória

Diagrama ilustrando a stack e a heap durante a passagem de parâmetros em Java (adaptado de JournalDev/DigitalOcean). No exemplo acima, o método `main` cria uma variável primitiva e um objeto, então invoca o método `foo` passando esse objeto como argumento. O diagrama mostra que:

1. Ao entrar em `main`, é alocado um frame na stack contendo a variável primitiva `i` (com valor 1) e a referência `obj` para um objeto no heap [25].
2. O objeto criado por `new Object()` reside na heap, enquanto a variável `obj` na stack armazena o endereço desse objeto (por exemplo, `0x001`) [26].
3. Quando `main` chama `foo(obj)`, a JVM cria um novo frame de stack para o método `foo`. O parâmetro `param` em `foo` recebe **uma cópia da referência** – digamos, `0x001` – apontando para o mesmo objeto no heap que `obj` apontava [18].
4. Dentro de `foo`, ao usar `param` para acessar/alterar o objeto (por exemplo, `param.toString()` ou modificando um atributo), estamos manipulando o **mesmo objeto compartilhado** entre `foo` e `main`. Qualquer mudança feita no objeto através de `param` será visível quando acessarmos o objeto através de `obj` em `main` [27].

5. Se `foo` atribuir um novo objeto a `param` (por exemplo, `param = new Objeto()` dentro de `foo`), essa nova referência ficará **restrita ao frame de** `foo`. Ao terminar `foo`, seu frame é liberado e essa nova referência se perde, enquanto `obj` em `main` continua apontando para o objeto original no heap – portanto, o objeto original permanece intacto exceto por modificações internas que `foo` possa ter feito antes da reatribuição [21].

Esse exemplo reforça visualmente o modelo de execução do Java: variáveis primitivas e referências vivem na stack, objetos vivem na heap, e o que é passado aos métodos são valores (copiados) – seja um valor numérico ou um endereço de objeto. Alterar o objeto através de uma referência refletirá em todas as referências que apontam a ele, mas alterar para onde a referência local aponta não afeta as demais referências fora do método.

Contornando a Ausência de Passagem por Referência em Java

Na prática, às vezes desejamos que um método altere múltiplos valores ou objetos "externos" (como faria com parâmetros por referência) ou retorne mais de um resultado. Já que não podemos passar variáveis por referência em Java, existem alguns **padrões e técnicas** para contornar essa limitação:

- **Usar objetos contêiner (wrapper) mutáveis:** Encapsule os valores que precisam ser modificados em objetos. Por exemplo, ao invés de passar um `int` que deve ser alterado pelo método, podemos criar uma classe `Contador` com um campo `valor`. Passamos uma instância de `Contador` e o método pode fazer `contador.valor++`. Como o objeto é compartilhado, o chamador verá o novo valor. Classes utilitárias como `AtomicInteger` ou `AtomicReference<T>` também podem servir como wrappers mutáveis. Vale lembrar que os *wrapper* primitivos padrão de Java (`Integer`, `Double` etc.) são **imutáveis** – atribuir um novo `Integer` dentro do método não atualiza o original [28]. Portanto, use wrappers próprios ou tipos projetados para mutação. Essa abordagem torna explícito que o método está alterando estado, o que pode melhorar a clareza em comparação a truques com referências.

- **Usar arrays ou coleções mutáveis:** Arrays em Java são objetos mutáveis e podem ser usados para simular múltiplos retornos ou saída por referência. Por exemplo, podemos passar um array de tamanho 1 contendo um valor primitivo; dentro do método, alterar `arr[0]` refletirá fora do método, já que o array é compartilhado (a referência do array é passada por valor) [29]. Da mesma forma, podemos usar um `List` mutável para acumular resultados dentro do método. Contudo, essa técnica deve ser usada com critério – passar um array apenas para obter um "retorno" extra pode tornar o código menos legível (é uma solução considerada pouco elegante) [30] [31].

- **Retornar múltiplos valores (tuplas ou objetos agregadores):** Em vez de abusar de efeitos colaterais, você pode projetar o método para retornar um objeto que contenha todos os resultados desejados. Por exemplo, se uma função precisa devolver dois números, crie uma classe simples `Par` com dois campos (ou use `AbstractMap.SimpleEntry`, ou uma biblioteca utilitária de tuplas) e retorne uma instância dessa classe. Embora isso implique criar uma nova classe ou objeto, muitas vezes essa é a abordagem mais limpa e orientada a objetos – os dados de saída ficam bem definidos e tipados, e o chamador lida com um único objeto resultado. Essa estratégia evita a necessidade de simuladores de passagem por referência, ao custo de um pouco mais de código (que pode ser minimizado com recordes ou classes aninhadas, por exemplo).

- **Reestruturar o código:** Em alguns cenários, a demanda por “múltiplos retornos” ou alteração de vários argumentos indica que o design pode ser melhorado. Pode-se avaliar dividir a função em funções menores (cada uma retornando algo específico), ou transformar variáveis relacionadas em atributos de um mesmo objeto/contexto que é passado ao método (e assim alterado internamente). Como ressaltado por um desenvolvedor experiente, se for necessário criar novos tipos apenas para encapsular retornos, **que assim seja, pois essa pode ser a solução correta** – provavelmente o modelo de objetos não estava representando bem o problema, daí a dificuldade que levou a querer usar passagem por referência [32] [33]. Em outras palavras, repensar a estrutura dos dados e responsabilidades dos métodos muitas vezes elimina a necessidade de truques de passagem por referência.

Em resumo, embora o Java não suporte passagem por referência na assinatura de métodos, ele permite alcançar resultados semelhantes através de objetos mutáveis e padrões de projeto adequados. A escolha da abordagem depende do contexto: para situações simples, um wrapper mutável ou um pequeno array pode resolver [34]; já em casos mais complexos, vale considerar retornos compostos ou refatoração do código. O mais importante é ter em mente que, sob o capô, **Java sempre passa uma cópia do valor (seja um literal primitivo ou uma referência a objeto)** [4] – e cabe a nós, desenvolvedores, trabalharmos com essa característica a favor, construindo soluções claras e seguras dentro desse modelo.

---

[1] [4] [7] [8] [9] [10] [11] [19] [24] java - Qual a diferença entre "passagem por valor" e "passagem por referência"? - Stack Overflow em Português
https://pt.stackoverflow.com/questions/59437/qual-a-diferen%C3%A7a-entre-passagem-por-valor-e-passagem-por-refer%C3%AAncia

[2] [23] Java is Pass by Value, Not Pass by Reference | DigitalOcean
https://www.digitalocean.com/community/tutorials/java-is-pass-by-value-and-not-pass-by-reference

[3] [5] Java passing by reference, where a good detailed explaination? - Oracle Forums
https://forums.oracle.com/ords/r/apexds/community/q?question=java-passing-by-reference-where-a-good-detailed-explainatio-7288

[6] Passagem por referencia vs Passagem por valor | by Thiago Emidio | Medium
https://medium.com/@thiagofarbo/passagem-por-referencia-vs-passagem-por-valor-d8923ea67ab9

[12] [22] methods - Is Java "pass-by-reference" or "pass-by-value"? - Stack Overflow
https://stackoverflow.com/questions/40480/is-java-pass-by-reference-or-pass-by-value

[13] [15] [16] Pilha e Heap - Java - GUJ
https://www.guj.com.br/t/pilha-e-heap/212040

[14] [17] [18] [25] [26] 关于java的堆（stack）和栈（heap）-Java Heap Space vs Stack – Memory Allocation in Java_some java objects are put in the heap, while some -CSDN博客
https://blog.csdn.net/hotdust/article/details/51810991

[20] [21] [27] Java: Pass By Value Or Pass By Reference | by Ananya Sen | Nov, 2020 | Medium | Medium
https://ananya281294.medium.com/java-passing-by-value-or-passing-by-reference-c75e312069ed

[28] [29] [30] [34] Como fazer para alterar num método um tipo primitivo? - Java - GUJ
https://www.guj.com.br/t/como-fazer-para-alterar-num-metodo-um-tipo-primitivo/25645

[31] [32] [33] When you need pass by reference in Java to assign values to multiple parameters, how do you do it? - Stack Overflow
https://stackoverflow.com/questions/2762171/when-you-need-pass-by-reference-in-java-to-assign-values-to-multiple-parameters