

CRIAÇÃO DE LISTAS EM JAVA

No Java, existem diferentes maneiras de criar listas imutáveis ou mutáveis utilizando métodos conhecidos como factory methods. Esses métodos fornecem formas convenientes de criar instâncias de coleções sem a necessidade de código repetitivo. Dentre os métodos da interface List, destacam-se:

- `Arrays.asList()`
- `List.of()`
- `List.copyOf()`

Neste artigo, vamos explorar cada um desses métodos, suas características, diferenças e como usá-los com exemplos práticos.

1. `Arrays.asList()`

O método `Arrays.asList()` foi introduzido nas primeiras versões de Java, como uma forma simples de converter arrays em listas. Ele recebe um array ou uma lista de argumentos e retorna uma lista fixa baseada no array.

Características:

- Retorna uma lista **mutável**, porém, **com tamanho fixo**. Ou seja, você pode **alterar os elementos da lista, mas não pode adicionar ou remover elementos**. A lista retornada é apenas uma visualização do array original.
- Modificações em um afetam o outro. Tanto o array original quanto a lista retornada compartilham a mesma referência de dados. Isso significa que se você modificar o array original, a lista refletirá essa mudança.

Aqui está o exemplo usando `Arrays.asList()` para criação de listas.

```
public class Principal {  
    public static void main(String[] args) {  
        // Criando um array de String  
        String[] array = {"A", "B", "C"};  
        // Convertendo o array em uma lista usando Arrays.asList()  
        List<String> lista = Arrays.asList(array);  
        // Modificar elementos da lista é permitido  
        lista.set(0, null);  
        System.out.println("Lista após modificar o primeiro elemento: " + lista);  
        System.out.println("Array após modificar a lista: " + Arrays.toString(array));  
        // Modificando o array original  
        array[1] = "Y";  
        System.out.println("Array após modificar o segundo elemento: " + Arrays.toString(array));  
    }  
}
```

```

// A lista reflete a mudança feita no array original
System.out.println("Lista após modificar o array: " + lista);

// Tentativa de adicionar ou remover elementos gera uma exceção
try {
    lista.add("D");
} catch (UnsupportedOperationException e) {
    System.out.println("Erro ao tentar adicionar um elemento: " + e.getMessage());
}
}
}

```

Saída esperada:

```

<terminated> ClassListOf [Java Application] C:\Program Files\ eclipse-java-2022-09-R-win32-x86_64\ eclipse\plugins\org.eclipse.justj.oper
Lista após modificar o primeiro elemento: [null, B, C]
Array após modificar a lista: [null, B, C]
Array após modificar o segundo elemento: [null, Y, C]
Lista após modificar o array: [null, Y, C]
Erro ao tentar adicionar um elemento: null

```

Explicação:

- Quando **você modifica o array original**, a lista gerada pelo `Arrays.asList()` também reflete essas mudanças, pois a lista é apenas uma **visualização** do array subjacente.
- No exemplo, modificamos o valor no índice 1 do array para "Y", e isso também alterou o valor na lista no mesmo índice. Ponto de atenção: Qualquer tentativa de adicionar ou remover elementos na lista gerará uma exceção `UnsupportedOperationException`, pois o tamanho da lista é fixo.
- Da mesma forma, quando você modifica a lista gerada pelo método `Arrays.asList()`, o array subjacente também será modificado, como é o caso da mudança do valor para *null* no elemento de índice 0 na lista.

2. List.of()

O método `List.of()` foi introduzido no Java 9 como parte do pacote de métodos `factory` para criar coleções imutáveis. Ele oferece uma maneira conveniente de criar listas de forma concisa.

Características:

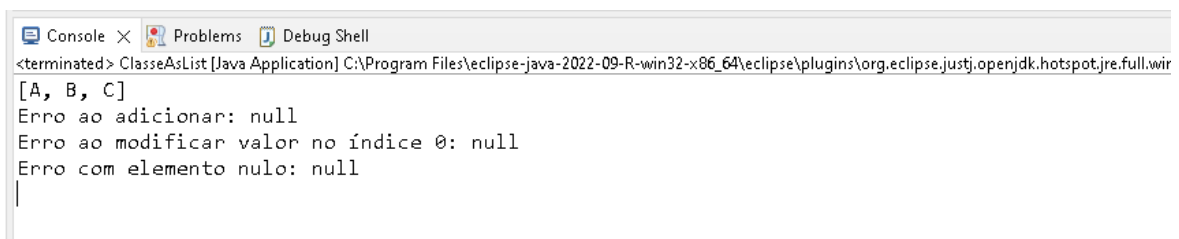
- Retorna uma lista imutável. Não é possível modificar a lista após sua criação (nem adicionar, remover ou alterar elementos).

- Não aceita elementos nulos; se um elemento nulo for passado, uma `NullPointerException` será lançada.

Exemplo:

```
public class ClasseAsList {  
    public static void main(String[] args) {  
        List<String> listaImutavel = List.of("A", "B", "C");  
        System.out.println(listaImutavel);  
  
        // Tentativa de modificar a lista gera uma exceção  
        try {  
            listaImutavel.add("D");  
        } catch (UnsupportedOperationException e) {  
            System.out.println("Erro ao adicionar: " + e.getMessage());  
        }  
  
        // Tentativa de modificar a lista gera uma exceção  
        try {  
            listaImutavel.set(0, "X");  
        } catch (UnsupportedOperationException e) {  
            System.out.println("Erro ao modificar valor no índice 0: " + e.getMessage());  
        }  
  
        // Tentativa de criar uma lista com elemento nulo  
        try {  
            List<String> listaComNulo = List.of("A", null, "C");  
        } catch (NullPointerException e) {  
            System.out.println("Erro com elemento nulo: " + e.getMessage());  
        }  
    }  
}
```

Saída esperada:



```
<terminated> ClasseAsList [Java Application] C:\Program Files\ eclipse-java-2022-09-R-win32-x86_64\ eclipse\ plugins\ org.eclipse.justj.openjdk.hotspot.jre.full.wir  
[A, B, C]  
Erro ao adicionar: null  
Erro ao modificar valor no índice 0: null  
Erro com elemento nulo: null  
|
```

Ponto de atenção: Ao contrário de `Arrays.asList()`, a lista criada por `List.of()` é verdadeiramente imutável. Qualquer tentativa de modificação resultará em uma exceção.

3. `List.copyOf()`

`List.copyOf()` O método `List.copyOf()` foi introduzido no Java 10 e serve para criar uma cópia imutável de uma coleção existente. Ele é útil quando você quer garantir que a lista original não possa ser modificada pela referência retornada.

Características:

- Cria uma cópia imutável de uma coleção passada como argumento.
- Se a coleção original for null, uma exceção `NullPointerException` será lançada.
- Se a coleção passada já for imutável (por exemplo, uma lista criada com `List.of()`), ela simplesmente a retorna sem criar uma nova cópia.

Exemplo:

```
public class ClasseCopyOf {

    public static void main(String[] args) {

        // Cenário 1: Criando uma lista mutável e uma cópia imutável
        List<String> listaMutavel = new ArrayList<>();
        listaMutavel.add("A");
        listaMutavel.add("B");
        listaMutavel.add("C");

        // Criando uma lista imutável a partir da lista mutável
        List<String> copiaImutavel = List.copyOf(listaMutavel);
        System.out.println("Cópia imutável: " + copiaImutavel);

        // Tentativa de modificar a lista imutável gera uma exceção
        try {
            copiaImutavel.set(0, "X");
        } catch (UnsupportedOperationException e) {
            System.out.println("Erro ao modificar cópia imutável: " + e.getMessage());
        }

        // Cenário 2: Criando uma cópia de uma lista já imutável
        List<String> listaImutavel = List.of("D", "E", "F");
        List<String> copiaDeListaImutavel = List.copyOf(listaImutavel);
    }
}
```

```

// Verificando se a lista imutável original e a cópia são a mesma instância
System.out.println("Lista imutável original: " + listaImutavel);
System.out.println("Cópia de lista imutável: " + copiaDeListaImutavel
System.out.println("São a mesma instância? " + (listaImutavel ==
copiaDeListaImutavel));

```

```

// Cenário 3: Tentativa de criar uma cópia de uma lista que contém null
List<String> listaComNull = new ArrayList<>();
listaComNull.add("G");
listaComNull.add(null);
listaComNull.add("H");

try {
    List<String> copiaComNull = List.copyOf(listaComNull);
} catch (NullPointerException e) {
    System.out.println("Erro ao tentar copiar lista com null: " +
e.getMessage());
}
}
}

```

Saída esperada:

```

<terminated> ClasseCopyOf [Java Application] C:\Program Files\ eclipse-java-2022-09-R-win32-x86_64\ eclipse\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_1
Cópia imutável: [A, B, C]
Erro ao modificar cópia imutável: null
Lista imutável original: [D, E, F]
Cópia de lista imutável: [D, E, F]
São a mesma instância? true
Erro ao tentar copiar lista com null: null

```

Ponto de atenção:

`List.copyOf()` cria uma cópia imutável da lista passada como argumento, tornando-a segura contra modificações.

Comparação entre os métodos

Característica	<i>Array.asList()</i>	<i>List.of()</i>	<i>List.copyOf()</i>
Introdução	Java 1.2	Java 9	Java 10
Mutabilidade	Mutável (apenas elementos)	Imutável	Imutável
Base	Baseado no array original	Independente	Cópia independente
Tamanho	Tamanho fixo	Tamanho fixo	Tamanho fixo
Elementos nulos	Permitidos	Não permitidos	Não permitidos
Desempenho	-	Mais eficiente para listas pequenas	-
Caso de uso	Ver um array como uma Lista	Criar pequenas listas imutáveis	Criar cópia imutável de uma coleção
Lança exceção em	operações de adicionar/remover	qualquer modificação	qualquer modificação

Quando usar cada método

1. `Array.asList()`:
 - Use quando precisar de uma visão de lista de um array existente.
 - Útil quando você quer modificar os elementos do array através da interface `List`.
 - Ideal para situações em que você precisa alternar entre array e `List`.
2. `List.of()`:
 - Use para criar pequenas listas imutáveis de forma rápida e eficiente.
 - Ótimo para constantes ou quando você tem certeza de que a lista não será modificada.
 - Perfeito para passar argumentos imutáveis para métodos.
3. `List.copyOf()`:
 - Use quando precisar de uma cópia imutável de uma coleção existente.
 - Útil para criar snapshots imutáveis de dados que podem mudar em outro lugar.
 - Ideal para garantir a imutabilidade em APIs públicas.

Armadilhas comuns a evitar

1. Com `Array.asList()`:
 - Tentar adicionar ou remover elementos (lança `UnsupportedOperationException`).
 - Esquecer que modificações na lista afetam o array original.

- Usar com tipos primitivos (cria uma lista com um único elemento do tipo array). [1]
- 2. Com `List.of()`:
 - Tentar modificar a lista resultante (lança `UnsupportedOperationException`).
 - Passar elementos nulos (lança `NullPointerException`).
 - Assumir que é mais eficiente para listas grandes (pode não ser o caso).
- 3. Com `List.copyOf()`:
 - Assumir que mudanças na coleção original afetarão a lista copiada.
 - Tentar modificar a lista resultante (lança `UnsupportedOperationException`).
 - Passar uma coleção com elementos nulos (lança `NullPointerException`).

Dicas de performance

1. `Array.asList()`:
 - Eficiente em termos de memória para listas grandes, pois não cria uma nova estrutura de dados.
 - Acesso rápido por índice, pois é respaldado por um array.
2. `List.of()`:
 - Mais eficiente em termos de memória para listas pequenas (até 10 elementos).
 - Otimizado para criação rápida de listas imutáveis.
3. `List.copyOf()`:
 - Pode ser menos eficiente para coleções muito grandes, pois cria uma cópia.
 - Eficiente para garantir imutabilidade sem o overhead de wrappers defensivos. [2]

Dica geral: Para melhor performance, escolha o método que melhor se alinha com o tamanho da sua lista e os requisitos de mutabilidade.

[1]

Arrays.asList() e Tipos Primitivos em Java

Quando você usa `Arrays.asList()` com um array de tipos primitivos (como `int[]`, `double[]`, etc.), o resultado pode ser surpreendente e não intuitivo.

O Comportamento Esperado vs. Real

Comportamento Esperado (Incorreto): Muitos desenvolvedores esperam que `Arrays.asList(intArray)` crie uma `List<Integer>` contendo os elementos do array de inteiros.

Comportamento Real: `Arrays.asList(intArray)` cria uma `List<int[]>` contendo um único elemento, que é o próprio array de inteiros.

Exemplo Demonstrativo

```
int[] intArray = {1, 2, 3, 4, 5};
List<int[]> list = Arrays.asList(intArray);

System.out.println(list.size()); // Imprime: 1
System.out.println(list.get(0)); // Imprime: [I@<endereço de memória>
```

Explicação

1. `Arrays.asList()` espera argumentos de tipo objeto (`T...`).
2. Um array de primitivos (`int[]`) é tratado como um único objeto em Java.
3. Portanto, o método cria uma lista contendo esse único objeto array.

[2]

Eficiência de `List.copyOf()` para Garantir Imutabilidade

A afirmação "Eficiente para garantir imutabilidade sem o overhead de wrappers defensivos" refere-se a uma vantagem específica do método `List.copyOf()` introduzido no Java 10. Vamos detalhar isso:

Contexto: Wrappers Defensivos

Antes de `List.copyOf()`, uma prática comum para garantir imutabilidade era usar "wrappers defensivos". Por exemplo:

```
public List<String> getItems() {
    return Collections.unmodifiableList(new ArrayList<>(this.items));
}
```

Este método cria uma nova `ArrayList` e a envolve com `Collections.unmodifiableList()`. Isso resulta em dois objetos extras:

1. A nova `ArrayList`
2. O wrapper `unmodifiable`

Como `List.copyOf()` Melhora Isso

`List.copyOf()` é mais eficiente porque:

1. **Cópia Única:** Cria apenas uma cópia imutável da lista original.

2. **Sem Wrapper Adicional:** A lista retornada já é imutável, sem necessidade de um wrapper.
3. **Otimização Interna:** Se a lista de entrada já for imutável, `List.copyOf()` pode simplesmente retornar a mesma instância.

Exemplo de Uso

```
public List<String> getItems() {  
    return List.copyOf(this.items);  
}
```

Benefícios de Desempenho

1. **Menor Consumo de Memória:** Menos objetos criados significam menos overhead de memória.
2. **Melhor Desempenho:** Menos indireção na estrutura de objetos pode levar a um acesso mais rápido aos dados.
3. **Coleta de Lixo Eficiente:** Menos objetos para o garbage collector processar.

Considerações

- O custo de copiar os elementos ainda existe, mas é geralmente menor que o overhead de wrappers múltiplos.
- Para listas muito grandes, o custo da cópia pode ser significativo. Nestes casos, retornar uma visão não modificável da lista original (usando `Collections.unmodifiableList(this.items)`) pode ser mais eficiente, mas com o risco de que alterações na lista original afetem a visão retornada.

Lista Imutável vs. Visão Não Modificável em Java

Embora ambos os conceitos impeçam modificações diretas na lista retornada, há diferenças cruciais entre uma lista imutável e uma visão não modificável.

Visão Não Modificável (*Unmodifiable View*)

Criada usando `Collections.unmodifiableList()`.

Características:

1. **Barreira de Modificação:** Impede modificações diretas na lista retornada.
2. **Referência à Lista Original:** Mantém uma referência à lista original.
3. **Reflete Mudanças:** Mudanças na lista original são refletidas na visão.

Exemplo:

```
List<String> original = new ArrayList<>(Arrays.asList("a", "b", "c"));
List<String> unmodifiableView = Collections.unmodifiableList(original);
original.add("d");
System.out.println(unmodifiableView); // Imprime: [a, b, c, d]
```

Lista Imutável

Criada usando List.copyOf() ou List.of().

Características:

1. **Verdadeiramente Imutável:** Não pode ser modificada de forma alguma.
2. **Cópia Independente:** Cria uma nova lista, independente da original (para List.copyOf()).
3. **Não Reflete Mudanças:** Mudanças na lista original não afetam a lista imutável.

Exemplo:

```
List<String> original = new ArrayList<>(Arrays.asList("a", "b", "c"));
List<String> immutableCopy = List.copyOf(original);
original.add("d");
System.out.println(immutableCopy); // Imprime: [a, b, c]
```

Principais Diferenças

1. **Isolamento de Mudanças:**
 - Visão Não Modificável: Não isola de mudanças na lista original.
 - Lista Imutável: Completamente isolada da lista original.
2. **Uso de Memória:**
 - Visão Não Modificável: Usa menos memória (apenas uma referência extra).
 - Lista Imutável: Usa mais memória (cópia completa dos elementos).
3. **Garantias de Imutabilidade:**
 - Visão Não Modificável: Imutabilidade "superficial".
 - Lista Imutável: Imutabilidade "profunda".
4. **Performance em Cenários de Múltiplas Threads:**

- Visão Não Modificável: Pode requerer sincronização adicional.
- Lista Imutável: Intrinsecamente thread-safe.

Quando Usar Cada Uma

- **Visão Não Modificável:**
 - Quando você quer evitar modificações acidentais, mas ainda permitir que a lista reflita mudanças na fonte original.
 - Em cenários de economia de memória, especialmente com listas grandes.
- **Lista Imutável:**
 - Quando você precisa de uma garantia forte de que a lista não mudará.
 - Em programação concorrente, onde a imutabilidade fornece garantias de thread-safety.
 - Quando você quer uma "fotografia" do estado da lista em um determinado momento.