Manual Básico de List e Arrays

> Referência: **Capítulo 7** do livro **Introdução à Programação Orientada a Objetiso Usando Java**.

Em Java, **lista (List)** e **array** são estruturas de dados diferentes com características distintas:

Array

- **Tamanho fixo**: Uma vez criado, não pode ser alterado
- **Tipo primitivo ou objeto**: Pode armazenar tipos primitivos (`int`, `double`, etc.) ou objetos
- **Sintaxe**: `Tipo[] nome = new Tipo[tamanho];`
- **Acesso direto**: Acesso muito rápido via índice
- **Métodos limitados**: Poucos métodos built-in

```
// Array de inteiros
int[] numeros = new int[5];
numeros[0] = 10;
numeros[1] = 20;

// Array de Strings
String[] nomes = {"Ana", "João", "Maria"};
```

List (Interface)

- **Tamanho dinâmico**: Cresce automaticamente conforme necessário
- **Apenas objetos**: Só armazena objetos (não tipos primitivos)
- **Implementações**: `ArrayList`, `LinkedList`, `Vector`
- **Métodos ricos**: `add()`, `remove()`, `contains()`, `size()`, etc.
- **Flexibilidade**: Mais operações disponíveis

```
// ArrayList (implementação mais comum)
List<String> lista = new ArrayList<>();

lista.add("Ana");
lista.add("João");
lista.add("Maria");
lista.remove("João"); // Remove elemento

// LinkedList
List<Integer> numeros = new LinkedList<>();
numeros.add(1);
numeros.add(2);
```

Principais diferenças:

Característica	Array	List
Tamanho	Fixo	Dinâmico
Tipos primitivos	Sim	Não (usa wrappers)
Desempenho	Mais rápido	Um pouco mais lento
Flexibilidade	Baixa	Alta
Métodos	Poucos	Muitos

Quando usar cada um:

- **Array**: Quando o tamanho é conhecido e fixo, e performance é crítica
- **List**: Quando precisa de flexibilidade no tamanho e funcionalidades avançadas

Conversão entre eles:

```
// Array para List
String[] array = {"a", "b", "c"};
List<String> lista = Arrays.asList(array);

// List para Array
List<String> lista = new ArrayList<>();
lista.add("a");
lista.add("b");

String[] array = lista.toArray(new String[0]);
```

Observação: A escolha depende das necessidades específicas do seu projeto!

Wrappers e Exemplos Práticos de Arrays e Listas em Java

O que são Wrappers?

Wrappers (ou classes invólucro) são classes em Java que encapsulam tipos primitivos, permitindo que sejam tratados como objetos. Isso é necessário porque estruturas como Listas só trabalham com objetos, não com tipos primitivos.

As principais classes wrapper são:

- `Integer` para `int`
- `Double` para `double`
- `Boolean` para `boolean`
- `Character` para `char`
- E outras (`Long`, `Float`, `Short`, `Byte`)

Exemplos Práticos

1. Arrays (Tamanho Fixo)

```java
// Array de tipos primitivos (int)
int[] notas = new int[5];
notas[0] = 8;
notas[1] = 7;
notas[2] = 9;

// Array de objetos (Strings)
String[] frutas = {"Maçã", "Banana", "Laranja"};

// Percorrendo um array
System.out.println("Notas dos alunos:");
for( int i = 0; i < notas.length; i++){
    System.out.println("Aluno "+(i+1) +": "+notas[i]);
}

// Usando for-each
System.out.println("\nFrutas disponíveis:");
for( String fruta : frutas){
    System.out.println(fruta);
}
```

2. Listas (Tamanho Dinâmico)

```java
import java.util.ArrayList;
import java.util.List;

// Lista de objetos (usando wrappers para números)
List<Integer> notas = new ArrayList<>();

notas.add(8);  // Autoboxing: int vira Integer automaticamente
notas.add(7);
notas.add(9);
notas.add(6); // Pode adicionar dinamicamente

// Lista de Strings
List<String> frutas = new ArrayList<>();

frutas.add("Maçã");
frutas.add("Banana");
frutas.add("Laranja");
frutas.add("Uva"); // Cresce automaticamente

// Removendo um elemento
frutas.remove("Banana");

// Percorrendo uma lista
System.out.println("Notas dos alunos:");
for( int i = 0; i < notas.size(); i++){
    System.out.println("Aluno "+(i+1) +": "+notas.get(i));
```

```
}

// Usando for-each
System.out.println("\nFrutas disponíveis:");
for( String fruta : frutas){
    System.out.println(fruta);
}

// Verificando se contém um elemento
if(frutas.contains("Maçã")){
    System.out.println("\nTemos maçãs!");
}
```

3. Exemplo com Wrappers

```
// Usando wrappers explicitamente
List<Integer> numeros = new ArrayList<>();
numeros.add(Integer.valueOf(10)); // Criando Integer explicitamente
numeros.add(20); // Autoboxing automático

// Convertendo entre primitivo e wrapper
int valorPrimitivo = numeros.get(0).intValue(); // Integer para int
Integer valorWrapper = Integer.valueOf(30); // int para Integer

// Trabalhando com outros wrappers
List<Double> precos = new ArrayList<>();
precos.add(10.99);
precos.add(25.50);

List<Boolean> status = new ArrayList<>();
status.add(true);
status.add(false);
```

4. Cenários de Uso Prático

Array (use quando):

- Sabemos o número exato de elementos antecipadamente
- Precisamos de máximo desempenho em operações de acesso
- Trabalhamos com tipos primitivos e queremos evitar overhead

List (use quando):

- O número de elementos é variável ou desconhecido
- Precisamos adicionar/remover elementos frequentemente
- Precisamos de métodos utilitários (busca, ordenação, etc.)
- Estamos trabalhando exclusivamente com objetos

Conversões entre Array e List

```java
// De Array para List
String[] array = {"A", "B", "C"};
List<String> lista = Arrays.asList(array);

// De List para Array
List<String> frutasList = new ArrayList<>();
frutasList.add("Maçã");
frutasList.add("Banana");

String[] frutasArray = frutasList.toArray(new String[0]);

// Com tipos primitivos (requer conversão)
int[] numerosArray = {1, 2, 3};
List<Integer> numerosList = new ArrayList<>();
for( int num : numerosArray){
    numerosList.add(num); // Autoboxing
}
```

O que é Overhead?

Overhead é um termo em computação que se refere ao "custo extra" ou "sobrecarga" adicional que ocorre quando usamos certas abstrações ou funcionalidades em relação a uma abordagem mais direta e simples.

Overhead no Contexto de Wrappers e Listas

No caso das **wrappers** e **listas** em Java, o overhead se manifesta de várias formas:

1. **Overhead de Memória**

```java
// Array de primitivos (menos overhead)
int[] numerosArray = new int[1000]; // ≈ 4KB de memória

// Lista de wrappers (mais overhead)
List<Integer> numerosList = new ArrayList<>(); // Muito mais memória
```

Cada objeto `Integer` consome aproximadamente **16-24 bytes** adicionalmente ao valor inteiro em si (4 bytes), devido aos:

- Cabeçalho do objeto
- Metadados da JVM
- Alinhamento de memória

2. **Overhead de Performance**

```java
// Operação direta com array (rápido)
int soma = 0;
for (int i = 0; i < numerosArray.length; i++) {
```

```
    soma += numerosArray[i]; // Acesso direto à memória
}

// Operação com lista (mais lento)
int soma = 0;
for (int i = 0; i < numerosList.size(); i++) {
    soma += numerosList.get(i); // Envolve chamada de método + unboxing
}
```

3. **Overhead de Processamento (Autoboxing/Unboxing)**

```
List<Integer> lista = new ArrayList<>();

// Autoboxing (overhead de conversão)
lista.add(10); // Equivale a: lista.add(Integer.valueOf(10));

// Unboxing (overhead de conversão)
int valor = lista.get(0); // Equivale a: int valor = lista.get(0).intValue();
```

Tipos Comuns de Overhead

Tipo de Overhead	Descrição	Exemplo
Memória	Uso adicional de memória	Wrappers consome mais memória que primitivos
CPU	Processamento adicional	Autoboxing/unboxing, chamadas de método
Tempo	Tempo de execução adicional	ArrayList realocando array interno
Comunicação	Dados adicionais em transmissões	Cabeçalhos em pacotes de rede

Quando o Overhead é Aceitável

O overhead das listas e wrappers é geralmente aceitável porque:

1. **Produtividade**: As listas oferecem métodos úteis que aceleram o desenvolvimento
2. **Flexibilidade**: Tamanho dinâmico é essencial em muitos cenários
3. **Legibilidade**: Código mais claro e expressivo
4. **Manutenibilidade**: Mais fácil de dar manutenção

Exemplo Prático de Overhead

```
public class ExemploOverhead {
    public static void main(String[] args) {
        // Teste com array (baixo overhead)
        long inicio = System.nanoTime();
```

```
        int[] array = new int[1000000];
        for (int i = 0; i < array.length; i++) {
            array[i] = i;
        }
        long fimArray = System.nanoTime();

        // Teste com lista (alto overhead)
        List<Integer> lista = new ArrayList<>();
        for (int i = 0; i < 1000000; i++) {
            lista.add(i); // Autoboxing + possível redimensionamento
        }
        long fimLista = System.nanoTime();

        System.out.println("Tempo array: " + (fimArray - inicio) + " ns");
        System.out.println("Tempo lista: " + (fimLista - fimArray) + " ns");
    }
}
```

Conclusão

O **overhead** é o preço que pagamos por abstrações mais convenientes. Em Java:

- **Arrays** têm menos overhead mas são menos flexíveis
- **Listas** têm mais overhead mas oferecem mais funcionalidades
- **Wrappers** permitem que primitivos se comportem como objetos, com custo adicional

Observação: A escolha entre baixo overhead (arrays) e conveniência (listas) depende do contexto específico da aplicação.