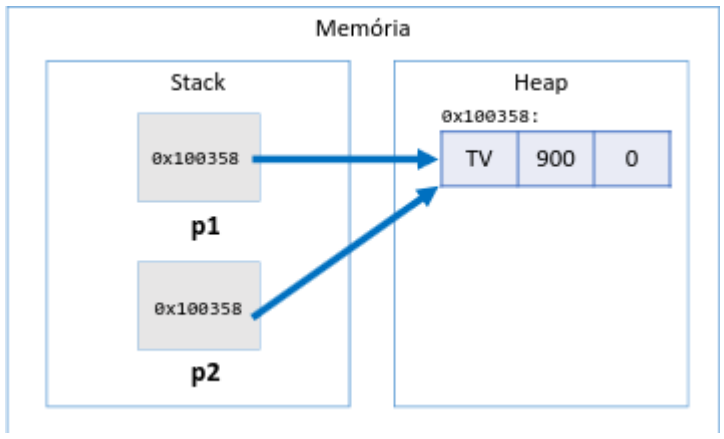
Seção 10: Arrays e listas

Tipo Referência VS Tipo Valor

- As classes são tipos referência, o que significa que variáveis cujo tipo são classes não armazenam diretamente os valores dos objetos, mas sim referências (ponteiros) para esses objetos na memória heap.
- Quando atribuímos uma variável a outra, como no exemplo `p2 = p1`, estamos fazendo com que ambas as variáveis apontem para o mesmo objeto na memória heap.

Exemplo:

```
Product p1, p2;
p1 = new Product("TV", 900.00, 0);
p2 = p1;
```



Em Java, a memória é dividida em duas principais áreas: **heap** e **stack**.

- **Heap**: onde são armazenados objetos e seus atributos, como no exemplo da `Product` com suas características (nome, preço, quantidade).
- **Stack**: onde ficam armazenadas as referências (endereços de memória) para esses objetos.

Referência "null"

Variáveis de tipo referência podem receber o valor `null`, indicando que não apontam para nenhum objeto.

Exemplo:

```
p2 = null;
```

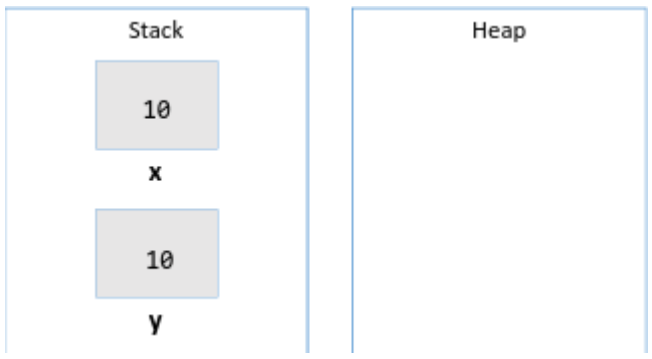
Tipos primitivos

Ao contrário dos tipos referência, os **tipos primitivos** são tipos valor. Eles armazenam diretamente o valor atribuído, como uma "caixa" que guarda o dado. Ao copiar uma variável primitiva para outra, ocorre a cópia do valor, não uma referência ao mesmo valor, como no exemplo:

Exemplo:

```
double x, y;
x = 10;
y = x;
```

- Neste caso, `y` recebe uma cópia do valor de `x`, e alterações em `x` não afetam `y`.
- Os dados de Tipo Primitivo ficam armazenados na Stack.



Comparação: Tipos referência vs. Tipos valor

Classe (Tipo Referência)	**Tipo Primitivo**
Vantagem: usufrui de todos recursos de Orientação a Objetos (OO)	Vantagem: é mais simples e mais performático
Variáveis são ponteiros	Variáveis são caixas
Objetos precisam ser instanciados usando `new` ou apontar para um objeto já existente.	Não precisa de instância. Uma vez declarados, estão prontos para uso.
Aceita valor `null`	Não aceita valor `null`
`Y = X;` "Y passa a apontar para onde X aponta"	`Y = X;` "Y recebe uma cópia de X"
Objetos instanciados no **heap**	"Objetos" instanciados no **stack**
Objetos não utilizados são desalocados pelo **garbage collector**	"Objetos" são desalocados imediatamente quando seu escopo de execução é finalizado.

Inicialização

Variáveis primitivas devem ser inicializadas antes do uso, pois Java não atribui um valor padrão para elas automaticamente. Para objetos e arrays, Java atribui valores padrão, como `0` para números e `null` para objetos.

Exemplo:

```
Product p = new Product();
```

Desalocação de Memória - Garbage Collector e Escopo Local

Garbage Collector

- O **garbage collector** é responsável por gerenciar automaticamente a memória alocada dinamicamente no heap. Ele monitora os objetos que não estão mais em uso e realiza sua desalocação.
- Quando um objeto deixa de ter referências que o apontam, ele é marcado para ser desalocado pelo garbage collector em um momento futuro.

Exemplo:

```
Product p1, p2;
p1 = new Product("TV", 900.00, 0);
p2 = new Product("Mouse", 30.00, 0);
p1 = p2;
```

- Antes de `p1 = p2`: há dois objetos (`"TV"` e `"Mouse"`) no heap.
- Após `p1 = p2`: o objeto `"TV"` é potencialmente desalocado pelo garbage collector, pois não há mais referências para ele.

Desalocação por Escopo Local

- Variáveis locais, incluindo variáveis primitivas e referências a objetos, são desalocadas automaticamente quando o escopo de execução da função termina.

- No caso de variáveis locais que apontam para objetos, as referências no stack são removidas, mas o objeto no heap só será desalocado pelo garbage collector se não houver mais nenhuma referência a ele.

Exemplo com variáveis primitivas:

```
void method1() {
    int x = 10;
    if (x > 0) {
        int y = 20;
    }
    System.out.println(x);
}
```

- Aqui, a variável `y` é desalocada assim que o bloco `if` termina. A variável `x` é desalocada quando o método `method1` sai de execução.

Exemplo com objetos:

```
void method1() {
    Product p = method2();
    System.out.println(p.Name);
}

Product method2() {
    Product prod = new Product("TV", 900.0, 0);
    return prod;
}
```

- Neste caso, o objeto `"TV"` criado dentro de `method2()` permanece alocado no heap, mesmo que o escopo de `method2` tenha terminado, pois a referência `p` de `method1` ainda o acessa.

Vetores

Características:

- **Homogêneo**: Todos os elementos são do mesmo tipo.
- **Ordenado**: Os elementos são acessados por seus índices, começando de 0.
- **Alocação Contígua**: Memória alocada de uma vez só em um bloco contínuo.

Vantagens dos Vetores:

1. **Acesso rápido aos elementos**: Acesso imediato a qualquer posição do vetor através dos índices.

Desvantagens dos Vetores:

1. **Tamanho fixo**: O tamanho do vetor não pode ser alterado após sua criação.
2. **Dificuldade em inserções e deleções**: Adicionar ou remover elementos no meio do vetor pode ser ineficiente, pois exige o deslocamento de outros elementos.]

Estrutura:

```
int[] vetor = new int[5];
```

Boxing

- Processo de conversão de um valor primitivo em um objeto de classe correspondente.
- Um objeto é criado na memória contendo o valor do tipo primitivo, e a variável de referência aponta para esse objeto.

Exemplo:

```
int x = 10;
Object obj = x;
```

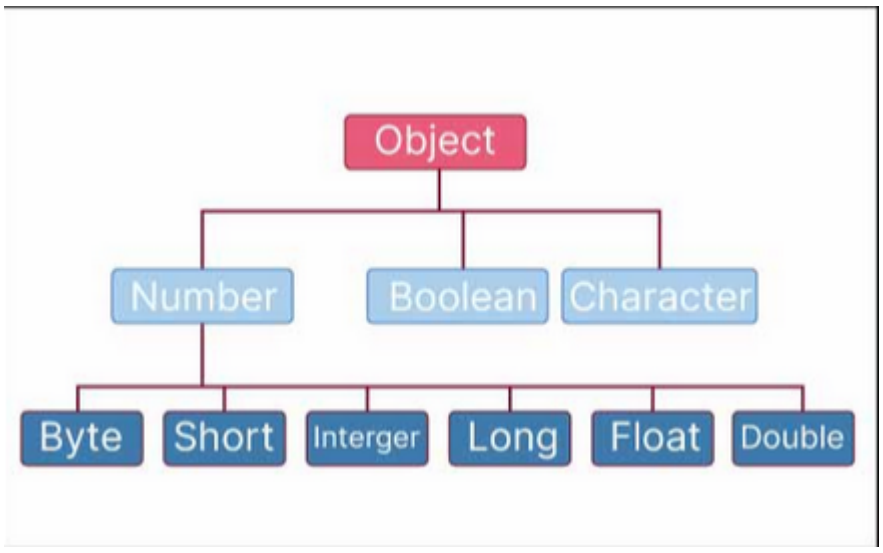
Unboxing

- Processo inverso, onde um objeto de referência é convertido de volta em um valor primitivo.
- Um valor primitivo é extraído do objeto de referência e armazenado em uma variável do tipo primitivo.

```
int y = (int) obj;
```

Wrapper Classes

- Classes que correspondem aos tipos primitivos, como `Integer` para `int`, `Boolean` para `boolean`, entre outros.
- Permitem que tipos primitivos sejam tratados como objetos, facilitando o *boxing* e *unboxing* automáticos.
- As *wrapper classes* aceitam o valor `null`, enquanto os tipos primitivos não.



- **Utilidade prática:**
  - Uma classe `Product` pode usar `Double` em vez de `double` para o campo `price`, permitindo que o valor seja `null` em casos como preços desconhecidos ou não aplicáveis.
- **Importância em Banco de Dad**
  - Em bancos de dados, certos campos podem conter valores `null`, e as *wrapper classes* garantem que essa equivalência seja mantida entre a tabela e a classe na memória.

Laço *For-Each*

- Uma sintaxe opcional e simplificada para percorrer coleções em Java, substituindo o laço `for` tradicional.
- O *for-each* percorre todos os elementos de uma coleção, chamando cada elemento pelo apelido definido.
- **Sintaxe**:

```
String[] vet = {"Maria", "Bob", "Alex"};
for (String obj : vet) {
	System.out.println(obj);
}
```

- O laço percorre o vetor usando um índice para acessar cada posição.
- **Vantagens do For-Each**
  - **Simplicidade**: Não é necessário gerenciar índices ou limites da coleção manualmente.
  - **Código mais limpo**: Torna a leitura do código mais direta, especialmente quando não há necessidade de acessar elementos diretamente por posição.

Lista

- Diferente do vetor, a lista é iniciada vazia e o seus elementos são alocados sob demanda.
- Cada elemento ocupa um "nó" na lista.
- Cada nodo dessa lista aponta para o próximo nodo, até o último nodo que é nulo
- **Tipo (interface):** List
- Por ser do tipo interface, ela não pode ser instanciada, por isso, é necessário de outra classe para instancia-la.
- Classes que a implementam: ArrayList, LinkedList, etc.

Estrutura:

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

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

- **Vantagens:**
  - Tamanho variável
  - Facilidade para se realizar inserções e deleções
- **Desvantagens:**
  - Acesso sequencial aos elementos (é necessário percorrer toda a lista para chegar no elemento desejado).
- Obs: Algumas classes que implementam a interface List otimizam o acesso a esses elemento. Como por exemplo a classe ArrayList.

Operações comuns:

- Tamanho da lista: **size()**
- Obter o elemento de uma posição: **get(position)**
- Inserir elemento na lista: **add(obj)**, **add(int, obj)**
- Remover elementos da lista: **remove(obj)**, **remove(int)**
- Remover elementos da lista por um predicado: removeIf(Predicate)
- Encontrar posição de elemento: indexOf(obj), lastIndexOf(obj)
- Filtrar lista com base em predicado: List result = list.stream().filter(x → x > 4).collect(Collectors.toList());
- Encontrar primeira ocorrência com base em predicado: Integer result = list.stream().filter(x → x > 4).findFirst().orElse(null);

Exemplos:

```
int primeiroElemento = numeros.get(0);
numeros.set(2, 10);
numeros.add(6);
numeros.remove(Integer.valueOf(3));
List<String> listaFiltrada = nomes.stream().filter(x → x.charAt(0) == 'M').collect(Collectors.toList());
Integer result = list.stream().filter(x → x > 4).findFirst().orElse(null);
```