

Introdução à linguagem Java

UA.DETI.POO

Paradigmas de programação

- ❖ As linguagens de programação baseiam-se em abstrações.
- ❖ Os paradigmas de programação definem os tipos de abstração e a forma como são combinadas num programa.

Imperativa

Que passos executar

Estruturada

Orientada a objetos

Declarativa

Que resultado obter

Funcional

Lógica

Programação Orientada a Objetos

- ❖ Paradigma mais comum em programação
 - Afeta análise, projeto (design) e programação
- ❖ A **análise** orientada por objetos
 - Determina **o que o sistema deve fazer**: Quais os **atores** envolvidos? Quais as **atividades** a serem realizadas?
 - **Decompõe** o sistema **em objetos**: Quais são? Que tarefas cada objeto terá que fazer?
- ❖ O **desenho** orientado por objetos
 - Define **como** o sistema será implementado
 - **Modela os relacionamentos** entre os objetos e atores (pode-se usar uma linguagem específica como UML)
 - Utiliza e reutiliza **abstrações** como classes, objetos, funções, frameworks, APIs, padrões de projeto

Programação Orientada a Objetos

❖ Objeto

- Cada entidade é representada por um objeto, que tem
estado: valor dos dados internos
comportamento: métodos (funções)

❖ Classe

- *blueprint* para criar objetos do mesmo tipo
- Define quais os dados e comportamentos que definem essa classe de objetos

❖ Desenho e programação orientada a objetos facilita:

- Modularidade
- Reutilização
- Substituição
- *Information-hiding*

Programação Orientada a Objetos

❖ Principais características

– Encapsulamento

Dados e funcionalidades sobre esses dados são implementadas e “escondidas” (*information hiding*) em estruturas (classes), fornecendo também modularidade e reusabilidade

– Herança

Classes (dados e comportamento) definidas com base em outras classes, permitindo reutilização e organização do código

– Polimorfismo

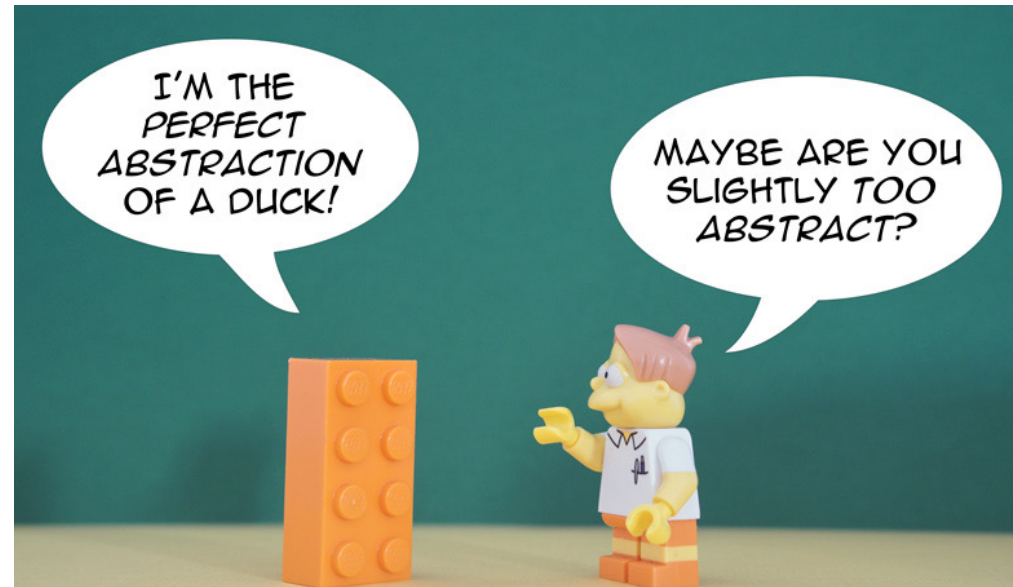
O comportamento de um objeto depende da natureza do objeto sobre o qual é invocado esse comportamento

– Abstração

Abstração

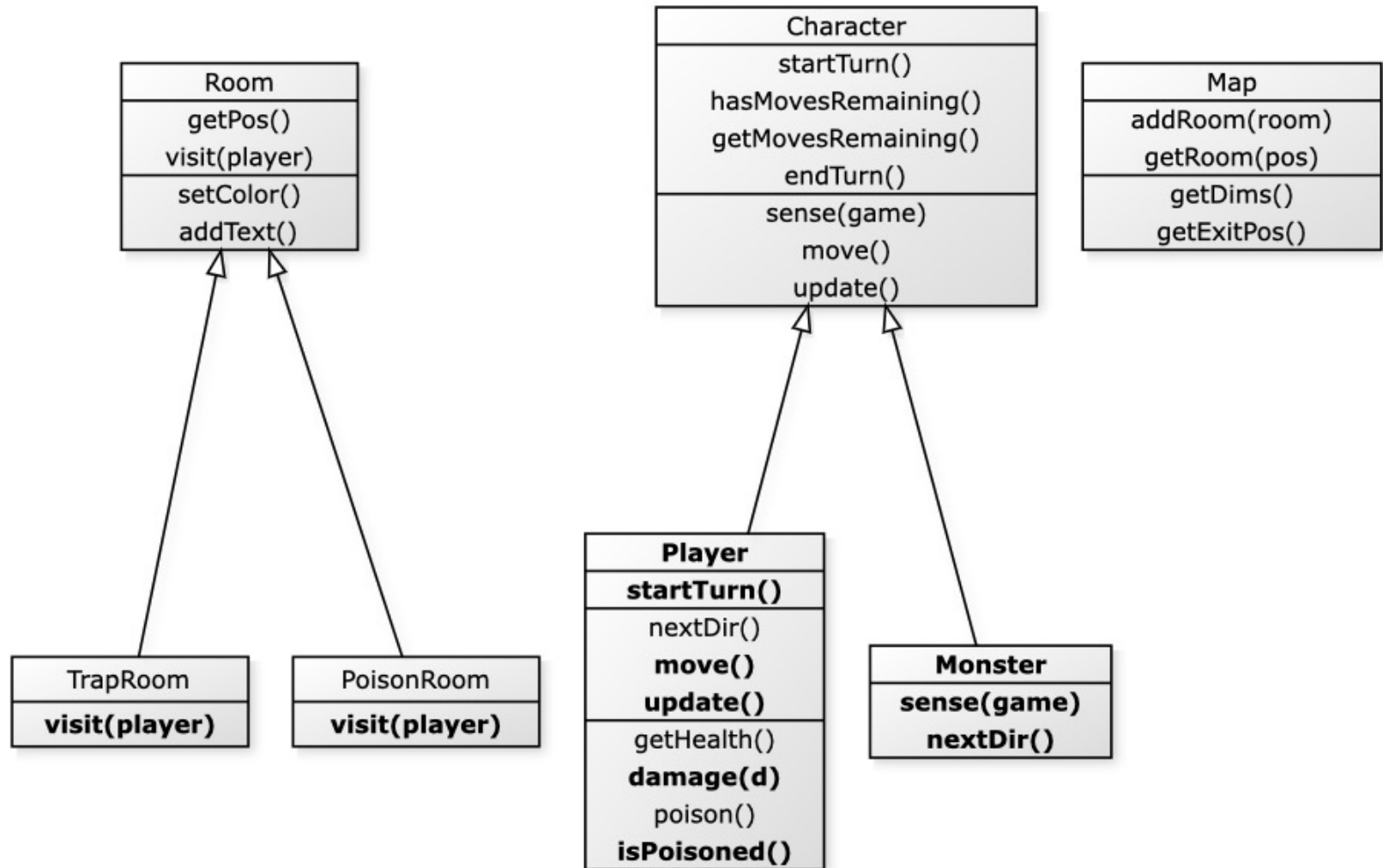
❖ *Abstraction is the purposeful suppression, or hiding, of some details of a process or artifact, in order to bring out more clearly other aspects, details, or structure. (T. Budd, An Introduction to Object-Oriented Programming)*

- Esconder/remover detalhes não necessários, mantendo o essencial
- Simplificação
- Generalização
- Ideia vs realidade



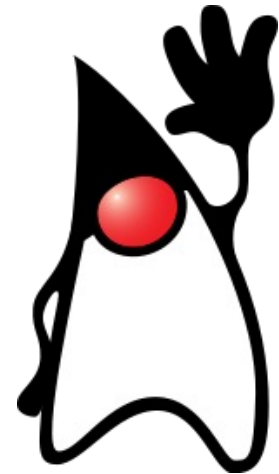
<https://thevaluable.dev/abstraction-type-software-example/>

Exemplo: Escape the cave!



A linguagem Java

- ❖ Java é uma das linguagens orientadas a objetos
 - Suporta também outros paradigmas (imperativa, estruturada, genérica, concorrente, reflexiva)
- ❖ Desenvolvida na década de 90, pela *Sun Microsystems*.
 - Sintaxe similar a C/C++
- ❖ Em 2008, foi adquirida pela *Oracle*.
- ❖ Página oficial:
 - <https://www.java.com>



A linguagem Java

❖ Design goals ([The Java Language Environment](#))

- Simple, Object Oriented, and Familiar
 - “fundamental concepts ... are grasped quickly”*
 - “designed to be **object oriented** from the ground up”*
 - “look and feel of C++” minus “the unnecessary complexities”*
- Robust and Secure
 - “extensive compile-time checking, followed by ...run-time checking”*
 - “memory management model... **new** operator... Garbage collector”*
- Architecture Neutral and Portable
 - “Java Compiler generates (architecture neutral) **bytecodes**”*
- High Performance
- Interpreted, Threaded, and Dynamic
 - Java Virtual Machine (JVM) *“**interpreter** can execute Java bytecodes”*
 - Dynamic binding

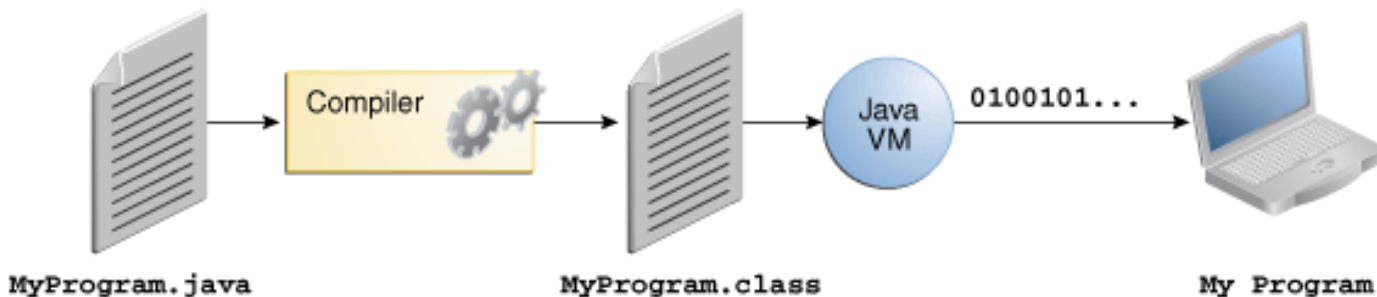
Atenção: Java não é uma linguagem puramente interpretada, como Python
Código Java é compilado para bytecode, que por sua vez é interpretado e executado pela JVM

Características gerais

- ❖ Software de código aberto, disponível sob os termos da GNU General Public License
- ❖ Facilidade de internacionalização (suporta nativamente caracteres UNICODE)
- ❖ Vasto conjunto de bibliotecas
- ❖ Facilidades para criação de programas distribuídos e multitarefa
- ❖ Libertação automática de memória por processo de coletor de lixo (*garbage collector*)
- ❖ Carregamento dinâmico de código
- ❖ Portabilidade

Escrever e executar programas

- ❖ Todo o código fonte é escrito em ficheiros de texto simples que terminam com a extensão **.java**.
 - São compilados com o compilador **javac** para ficheiros **.class**.
- ❖ Um ficheiro **.class** contém código bytecode que é executado por uma máquina virtual.
 - Não contém código nativo do processador
 - É executado sobre uma instância da Java Virtual Machine



Java Virtual Machine

❖ **Vantagens => grande portabilidade**

- A JVM é um programa que carrega e executa os aplicativos Java, convertendo o bytecode em código nativo.
- Assim, estes programas são independentes da plataforma onde funcionam.
- O mesmo ficheiro .class pode ser executado em máquinas diferentes (que corram Windows, Linux, Mac OS, etc.).

❖ **Desvantagem => menor desempenho**

- O código é mais lento se comparado com a execução de código nativo (e.g. escrito em C ou C++).

Estrutura básica de um programa Java

- ❖ O que noutras linguagens se designa por **programa principal** é em Java uma classe declarada como **public class** na qual definimos uma função chamada **main()**
 - Declarada como public static void
 - Com um parâmetro args, do tipo String[]
- ❖ Este é o formato padrão, absolutamente fixo

```
// inclusão de pacotes/classes externas
// o pacote java.lang é incluído automaticamente

public class Exemplo {
    // declaração de dados que compõem a classe
    // declaração e implementação de métodos
    public static void main(String[] args) {
        /* início do programa */
    }
}
```

Exemplo simples

```
package aula01;  
  
public class MyFirstClass {  
  
    public static void main(String[] args) {  
        System.out.println("Hello class!");  
    }  
}
```

Hello class!

Espaço de Nomes - Package

```
package aula01;
```

- ❖ Em Java a gestão do espaço de nomes (*namespace*) é efetuado através do conceito de package.
 - Evita conflitos de nomes de classes
- ❖ O espaço de nomes é baseado numa estrutura de sub-directórios
 - O package 'aula01' do exemplo anterior vai corresponder a uma entrada de directório
 - O "*Fully Qualified Name*" da classe será aula01.MyFirstClass
 - Este FQN corresponde ao caminho aula01/MyFirstClass.class
- ❖ Voltaremos a isto mais tarde

Variáveis e tipos primitivos

❖ Java é *statically typed*: todas as variáveis têm de ser declaradas e o seu tipo tem de ser definido

– <https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html>

Type	Size	Range	Default
boolean	1 bit	true or false	false
byte	8 bits	[-128, 127]	0
short	16 bits	[-32,768, 32,767]	0
char	16 bits	['\u0000', '\uffff'] or [0, 65535]	'\u0000'
int	32 bits	[-2,147,483,648 to 2,147,483,647]	0
long	64 bits	$[-2^{63}, 2^{63}-1]$	0
float	32 bits	32-bit IEEE 754 floating-point	0.0
double	64 bits	64-bit IEEE 754 floating-point	0.0

Variáveis e tipos primitivos

```
package aula01;

public class Testes {

    public static void main(String[] args) {
        boolean varBoolean = true;
        char varChar = 'A';
        byte varByte = 100;
        double varDouble = 34.56;

        System.out.println(varBoolean);
        System.out.println(varChar);
        System.out.println(varByte);
        System.out.println(varDouble);
    }
}
```

```
true
A
100
34.56
```

Declaração e inicialização de variáveis

- ❖ As variáveis locais têm de ser inicializadas
- ❖ Podemos fazê-lo de várias formas:
 - na altura da definição:

```
double peso = 50.3;  
int dia = 18;
```
 - usando uma instrução de atribuição (símbolo '='):

```
double peso;  
peso = 50.3;
```
 - lendo um valor do teclado ou de outro dispositivo:

```
double km;  
km = sc.nextDouble();
```

```
(...)  
double uninitialized;  
System.out.println(uninitialized);  
(...)
```

Regras e convenções

- ❖ Java é *case-sensitive*
- ❖ Nomes das variáveis devem começar por uma letra (não usar \$ ou _ apesar de permitidos)
- ❖ Nomes devem ser em minúsculas ou *camel case*
`dia, temperatura, velocidadeMaxima`
(e sem acentos, cedilhas, ...)
- ❖ Evitar abreviações difíceis de perceber (ex: `vm`)
- ❖ Para constantes, usar maiúsculas e _
`VELOCIDADE_MAXIMA`

Estilo de escrita de código

❖ Classes

String, Player, Student, AveiroStudent

❖ Métodos e variáveis

area, fillArea()

❖ Constantes

PI, SPEED_LIMIT

❖ Estrutura do código

```
class AllTheColorsOfTheRainbow {  
    int anIntegerRepresentingColors;  
    void changeTheHueOfTheColor(int newHue) {  
        // ...  
    }  
    // ...  
}
```

Operadores

- ❖ Os operadores utilizam um, dois ou três argumentos e produzem um valor novo.
- ❖ Java inclui os seguintes operadores:
 - atribuição: `=`
 - aritméticos: `*`, `/`, `+`, `-`, `%`, `++`, `--`
 - relacionais: `<`, `<=`, `>`, `>=`, `==`, `!=`
 - lógicos: `!`, `||`, `&&`
 - manipulação de bits: `&`, `~`, `|`, `^`, `>>`, `<<`
 - operador de decisão ternário `?`

Expressões com operadores

❖ Atribuição

```
int a = 1; // a toma o valor 1
int b = a; // b toma o valor da variável a
a = 2; // a fica com o valor 2, b tem valor 1
```

❖ Aritméticos

```
double x = 2.5 * 3.75 / 4 + 100; // prioridade?
double y = (2.5 * 3.75) / (4 + x);
int num = 57 % 2; // resto da divisão por 2
```

❖ Relacionais

```
boolean res = (x >= y);
boolean e = (x == y); // e <- "x igual a y"?
```

❖ Lógicos

```
char code = 'F';
boolean capitalLetter = (code >= 'A') && (code <= 'Z');
```

```
int age=11;
boolean isTeen = (age >= 13) && (age <= 19); // short circuit evaluation
```

Precedência de operadores

- ❖ A ordem de execução de operadores segue regras de precedência.

```
int a = 5;  
int b = -15;  
double c = ++a-b/30;
```

- ❖ Para alterar a ordem e/ou clarificar as expressões complexas sugere-se que usem parênteses.

```
c = (++a)-(b/30);
```

Operator Precedence

Operators	Precedence
postfix	<i>expr</i> ++ <i>expr</i> --
unary	++ <i>expr</i> -- <i>expr</i> + <i>expr</i> - <i>expr</i> ~ !
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	
logical AND	&&
logical OR	
ternary	? :
assignment	= += -= *= /= %= &= ^= = <<= >>= >>>=

Operadores aritméticos unários

- ❖ Os operadores unários de incremento (++) e decremento (--) podem ser utilizados com variáveis numéricas.
- ❖ Quando colocados antes do operando são pré-incremento (++x) ou pré-decremento (--x).
 - a variável é primeiro alterada antes de ser usada.
- ❖ Quando colocados depois do operando são pós-incremento (x++) e pós-decremento (x--)
 - a variável é primeiro usada na expressão e depois alterada.

```
int a = 1;  
int b = ++a;  
int c = b++;
```


Operadores aritméticos unários

- ❖ Os operadores unários de incremento (++) e decremento (--) podem ser utilizados com variáveis numéricas.
- ❖ Quando colocados antes do operando são pré-incremento (++x) ou pré-decremento (--x).
 - a variável é primeiro alterada antes de ser usada.
- ❖ Quando colocados depois do operando são pós-incremento (x++) e pós-decremento (x--)
 - a variável é primeiro usada na expressão e depois alterada.

```
int a = 1;  
int b = ++a; // a = 2, b = 2  
int c = b++; // b = 3, c = 2
```

Constantes / Literais

- ❖ Literais são valores invariáveis no programa

23432, 21.76, false, 'a', "Texto", ...

- ❖ Normalmente o compilador sabe determinar o seu tipo e interpretá-lo.

```
int x = 1234;  
char ch = 'Z';
```

- ❖ Em situações ambíguas podemos adicionar caracteres especiais:

- **l/L** = long, **f/F** = float, **d/D** = double
- **0x/0X**valor = valor hexadecimal
- **0**valor = valor octal.

```
long a = 23L;  
double d = 0.12d;  
float f = 0.12f; // obrigatório
```

Conversão de tipo de variável

- ❖ Podemos guardar um valor com menor capacidade de armazenamento numa variável com maior capacidade de armazenamento
- ❖ A conversão respetiva será feita automaticamente:
 - byte -> short (ou char) -> int -> long -> float -> double
- ❖ A conversão inversa gera um erro de compilação.
 - Entretanto podemos sempre realizar uma conversão explícita através de um operador de conversão:

```
int a = 3;  
double b = 4.3;  
double c = a; // conversão automática de int para double  
a = (int) b; // b é convertida/truncada forçosamente para int
```

Imprimir variáveis e literais

❖ `System.out.println(...);`

- escreve o que estiver entre (..) e muda de linha

❖ `System.out.print(...);`

- escreve o que estiver entre (..) e não muda de linha

❖ Exemplos

```
String nome = "Adriana";  
int x = 75;  
double r = 19.5;  
System.out.println(2423);  
System.out.print("Bom dia " + nome + "!");  
System.out.println();  
System.out.println("Inteiro de valor: " + x);  
System.out.println("Nota final: " + r);
```

```
2423  
Bom dia Adriana!  
Inteiro de valor: 75  
Nota final: 19.5
```

Ler dados

- ❖ Podemos usar a classe Scanner para ler dados a partir do teclado.

```
import java.util.Scanner;  
...  
Scanner sc = new Scanner(System.in);
```

- ❖ Métodos úteis da classe Scanner:
 - `nextLine()` – lê uma linha inteira (String)
 - `next()` – lê uma palavra (String)
 - `nextInt()` – lê um inteiro (int)
 - `nextDouble()` – lê um número real (double)

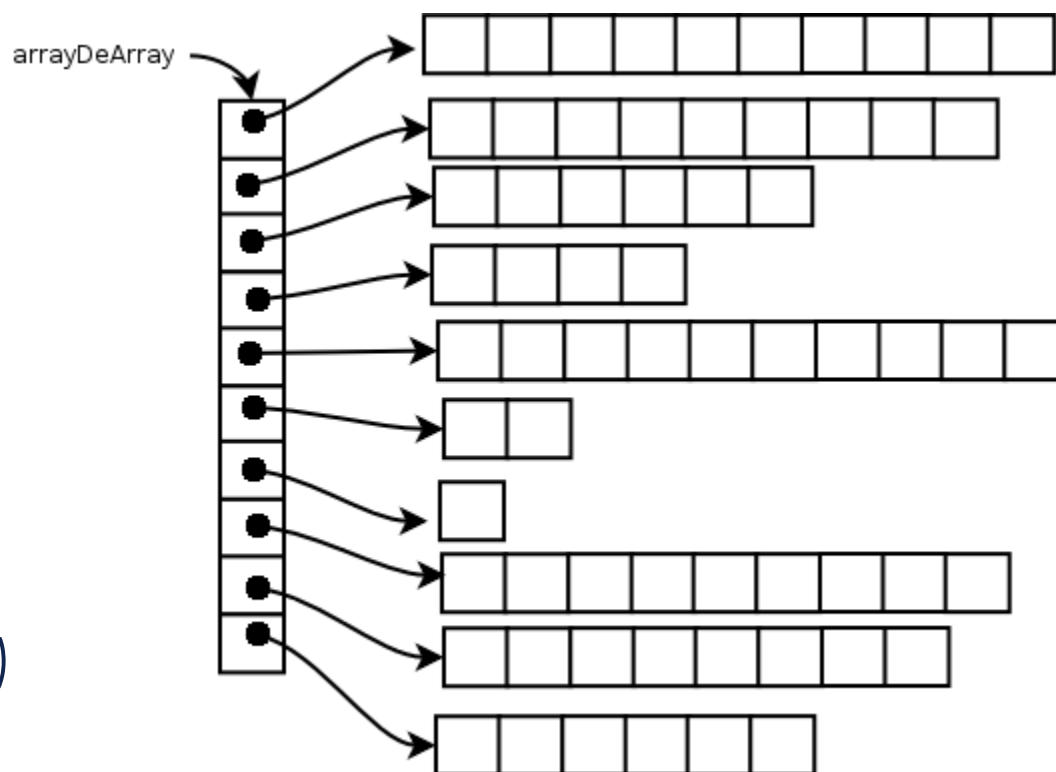
Exemplo

```
import java.util.Scanner;
public class Testes {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Qual é o teu nome? ");
        String nome = sc.nextLine();
        System.out.print("Que idade tens? ");
        int idade = sc.nextInt();
        System.out.print("Quanto pesas? ");
        double peso = sc.nextDouble();
        System.out.println("Nome: " + nome);
        System.out.println("Idade: " + idade + " anos");
        System.out.println("Peso: " + peso + "Kgs.");
        sc.close();
    }
}
```

```
Qual é o teu nome? Ana Lima
Que idade tens? 28
Quanto pesas? 55
Nome: Ana Lima
Idade: 28 anos
Peso: 55.0Kgs.
```

Tipos referenciados

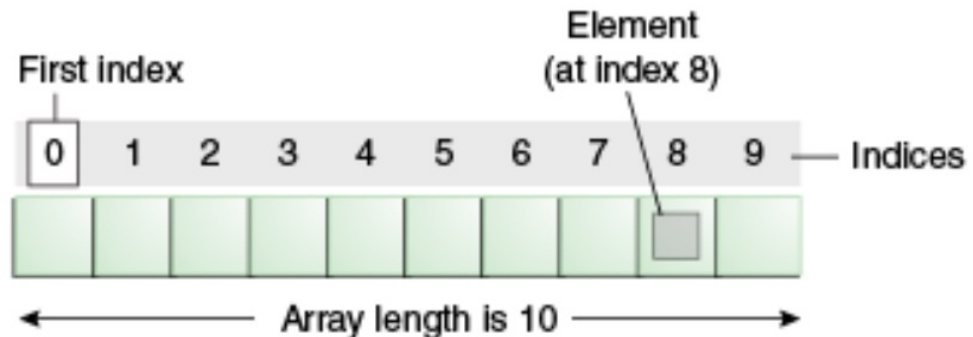
- ❖ Variáveis destes tipos não contêm os valores mas os endereços para acesso aos valores efetivos



- ❖ Incluem:
 - Vetores (arrays)
 - Objetos

Vetores em Java

- ❖ Os vetores em Java têm **dimensão fixa**, não podendo aumentar de dimensão em tempo de execução
- ❖ A instrução **new** cria um vetor com a dimensão indicada e inicializa todas as posições
 - Para os tipos **primitivos** com o valor por omissão
 - Para **referências**, com o valor null



An array of 10 elements.

Vetores em Java

- ❖ Podemos declarar **vetores** (arrays) de variáveis de um mesmo tipo

```
int[] vet1;  
int vet2[]; // sintaxe alternativa e equivalente, mas não encorajada
```

- ❖ Para além da declaração, precisamos ainda de definir a sua **dimensão**.

- inicialização com valores por omissão:

```
int[] v1 = new int[3]; // vetor com 3 elementos: 0, 0 ,0
```

- declaração e inicialização com valores específicos

```
int[] v2 = { 1, 2, 3 }; // vetor com 3 elementos: 1, 2, 3  
// ou
```

```
int[] v3 = new int[] { 1, 2, 3};
```

Acesso a elementos do vetor

- ❖ Os elementos são acedidos através de índices.
 - O índice do primeiro elemento é 0 (zero).

```
int[] tabela = new int[3]; // índices entre 0 e 2
tabela[0] = 10;
tabela[1] = 20;
tabela[2] = 30;
tabela[3] = 11; // erro!!
```

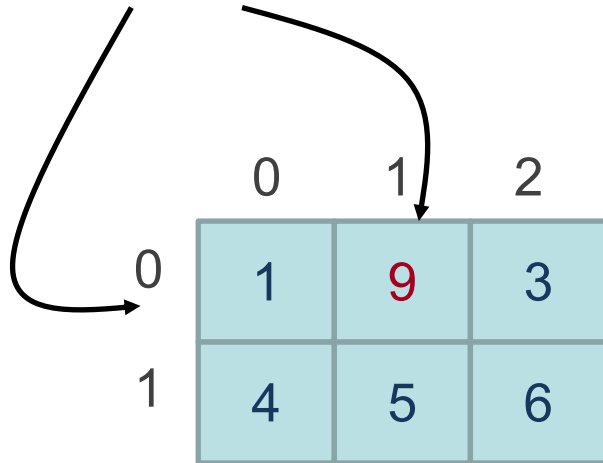
- ❖ O tamanho de um vetor **v** é dado por **v.length**.

```
System.out.println(tabela.length); // 3
```

Vetores multidimensionais

- ❖ É possível criar vetores multidimensionais, i.e. vetores de vetores:

```
int[][] a = { { 1, 2, 3, }, { 4, 5, 6, } } ;  
System.out.println(a.length); // 2  
System.out.println(a[0].length); // 3  
a[0][1] = 9;
```



Vetores multidimensionais

- ❖ São vetores de vetores (arrays de arrays)
 - São implementados usando aninhamento/cascata

```
int tabela[][]= new int[30][20];
```

- Define tabela como sendo do tipo int[][]
- Reserva, dinamicamente, um vetor de 30 elementos, cada um deles do tipo int[20]
- Reserva 30 vetores de 20 inteiros e guarda a referência (endereço) para cada um destes no vetor de 30 posições

Sumário

- ❖ Programação orientada a objetos
- ❖ Estrutura de um programa em java
 - Classe principal, função main
- ❖ Dados
 - Tipos primitivos, variáveis
- ❖ Operadores e precedências
- ❖ Expressões com operadores
- ❖ Vetores

- ❖ Próxima aula
 - Instruções condicionais: if, if .. else, switch
 - Ciclos: while, do ... while, for
 - Modularidade

Saber mais

The Java™ Tutorials

Java Tutorials Learning Paths

<https://docs.oracle.com/javase/tutorial/tutorialLearningPaths.html>

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases and might use technology that is no longer current.

Are you a student trying to learn the Java language or a professional seeking to expand your skill set? If you are feeling a bit overwhelmed by the breadth of the Java platform, here are a few learning paths to help you get started with your Java learning experience.



New To Java



The following trails are most useful for beginners:

- [Getting Started](#) – An introduction to Java technology and lessons on installing Java development software and using it to create a simple program.
- [Learning the Java Language](#) – Lessons describing essential concepts such as classes, objects, inheritance, datatypes, generics, and packages.
- [Essential Java Classes](#) – Lessons on exceptions, basic input/output, concurrency, regular expressions, and the platform environment.

Building On The Foundation



Ready to dive deeper into the technology? See the following:

- [Collections](#) – Lessons on using and extending the Java Collections Framework.
- [Lambda Expressions](#): Learn how and why to use Lambda Expressions.
- [Aggregate Operations](#): Explore how Aggregate Operations provide powerful filtering capabilities.
- [Packaging Programs In JAR Files](#) – Lesson on creating and using JAR files.
- [Internationalization](#) – An introduction to designing software for various languages and regions.
- [Reflection](#) – An API that represents ("reflects") the class structure of the Java Virtual Machine.