

# Curso Java



Autor: Lucas Abbade

# Ementa

1. Introdução
2. Algoritmos
3. Programação Orientada a Objeto
4. Interface Gráfica de Usuário
5. Conexão com Banco de Dados
6. Requests em API

# Introdução

# Introdução

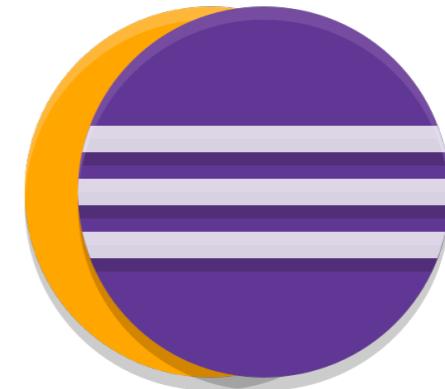
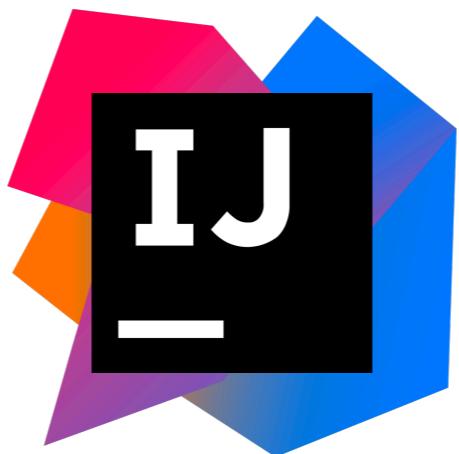
- Java nasceu na década de 90 com a proposta de ser uma “linguagem universal”
- Completamente orientada a objeto 
- Java Virtual Machine (JVM)

# JVM

A JVM permite que programas feitos em Java sejam executados em praticamente qualquer coisa.

# IDEs

- NetBeans
- Eclipse
- IntelliJ



# NetBeans

Criando um projeto no NetBeans:

1. Clique em “arquivo” -> “novo projeto”
2. “Java Application”
3. Dê um nome ao projeto
4. “finalizar”

# Hello World

```
public class HelloWorld
{
    public static void main(String[] args)
    {
        System.out.println("Hello World");
    }
}
```

# Hello World

A função “*main*”, assim como no c++, é chamada no início da execução.

# Algoritmos

# Algoritmos

- Variáveis e tipos primitivos
- *Standard Input/Output*
- Estruturas de decisão
- Estruturas de repetição
- Funções
- Exceções
- Estruturas de Dados

# Tipos Primitivos

- boolean
- int (short, long, long long)
- float, double
- char

# Variáveis

A declaração de variáveis no Java é feita da seguinte forma:

*tipoVar nomeVar;*

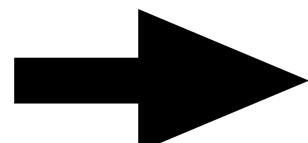
Opcionalmente é possível designar um valor inicial.

# Variáveis

```
int a;  
float b;  
char c;  
boolean d = true;  
double e = 2.73;
```

# Casting

Para converter um tipo primitivo em outro, é necessário realizar uma operação de *casting* em alguns casos\*, da seguinte forma:



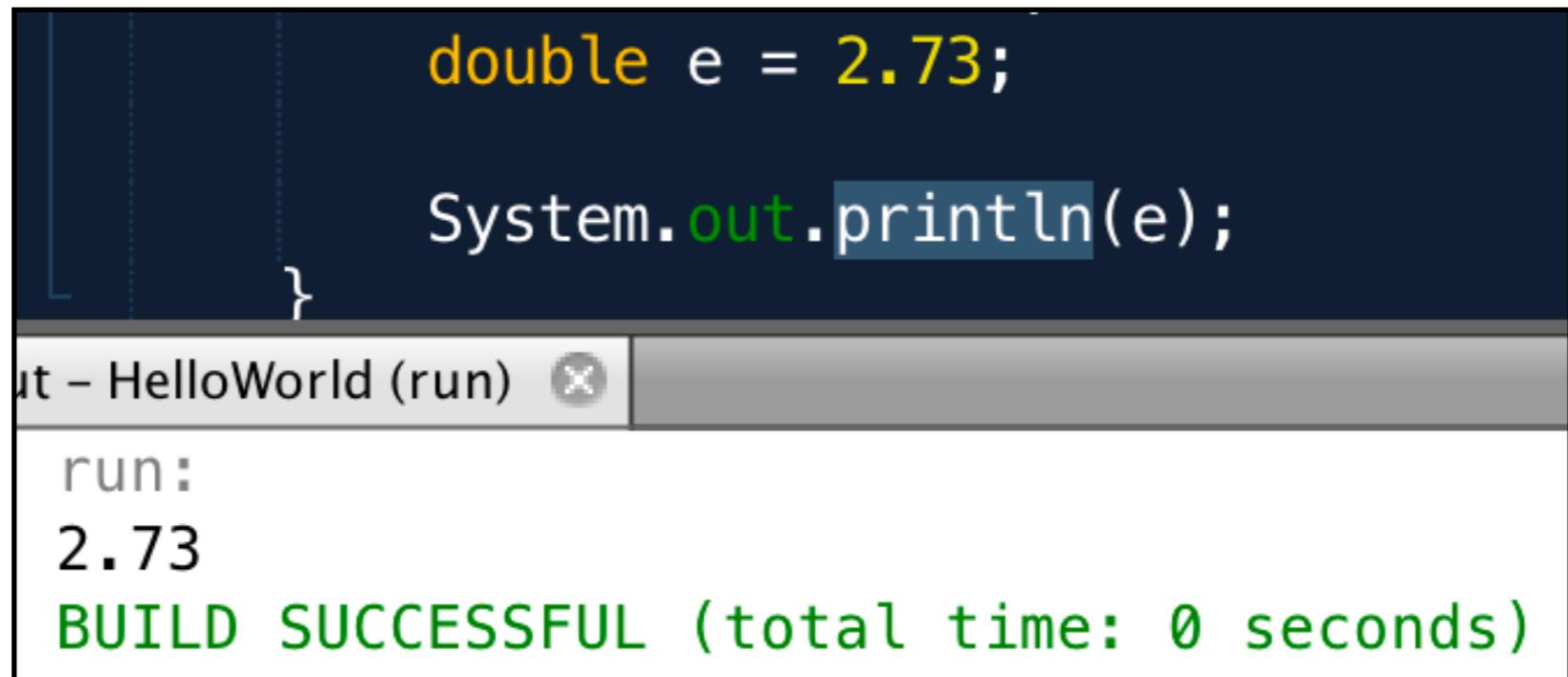
```
double e = 2.73;  
int e2 = (int)e;  
System.out.println(e2);
```

# *Standard Output*

A saída padrão é a função:

*System.out.println()*

# *Standard Output*



A screenshot of an IDE interface showing the standard output window. The code in the editor pane is:

```
    double e = 2.73;  
    System.out.println(e);  
}
```

The output window title is "out - HelloWorld (run)". The output text is:

```
run:  
2.73  
BUILD SUCCESSFUL (total time: 0 seconds)
```

# *Standard Output*

A função *print* é capaz de reconhecer automaticamente todos os tipos primitivos.

Falaremos sobre como ela lida com Classes futuramente.

# *Standard Input*

Para utilizar a entrada padrão existem algumas formas diferentes, a mais simples é criando um objeto da classe *Scanner*.

# *Standard Input*

```
Scanner sc = new Scanner(System.in);
int n = sc.nextInt();
System.out.println(n);
```

# *Standard Input*

A classe *Scanner* possui vários métodos (funções) para receber diferentes tipos de dados, exemplo:

- `nextInt()` <- Retorna a próxima int
- `nextBoolean()` <- Retorna a próxima boolean
- `nextDouble()` <- ... próxima double
- `nextLine()` <- ... próxima linha de texto

# *Standard Input*

Para utilizar a classe *Scanner* é necessário importá-la:

```
import java.util.Scanner;
```

As IDEs normalmente são capazes de reconhecer a necessidade de importar a classe e fazem isso automaticamente.

# Estruturas de Decisão

Estruturas de decisão são usadas para determinar o fluxo do código dependendo de uma condição.

# Estruturas de Decisão

Exemplo:

```
Scanner sc = new Scanner(System.in);
int n = sc.nextInt();

if (n < 5) {
    System.out.println("N menor do que 5");
} else if (n < 10) {
    System.out.println("N entre 5 e 10");
} else {
    System.out.println("N maior ou igual a 10");
}
```

# Estruturas de Decisão

Forma:

*if (condição) {*

*...*

*} else if (outra condição) {*

*...*

*} else {*

*...*

*}*

# Estruturas de Decisão

- O uso de “*else if*” e “*else*” são opcionais.
- Podem ser usados tantos “*else if*” quanto necessário.
- Note que o uso do “*else if*” é diferente do uso de vários “*if*”.

# Estruturas de Repetição

Estruturas de repetição servem para executar repetidamente um trecho de código, enquanto uma condição for verdadeira.

# *While*

```
while (n > 0) {  
    System.out.println(n);  
    n -= 1;  
}
```

# *While*

O código anterior será executado **enquanto** “n” for maior do que 0.

# *For*

O *for* é utilizado quando se sabe **exatamente** quantas vezes deseja-se repetir o trecho de código.

Estrutura do *for*:

*for (int i = valorInicial; i < valorFinal; incremento)*

# *For*

```
for (int i = 0; i < 10; i++) {  
    System.out.println(i);  
}
```

# *For*

- A variável de iteração do *for* (normalmente chamada de “i”) pode ter outros nomes. É comum em situações de repetições aninhadas que se use “i”, “j”, “k”...

# *For*

```
for (int i = 0; i < 10; i++) {  
    for (int j = 0; j < 100; j++) {  
        for (int k = 0; k < 5; k++) {  
            ...  
        }  
    }  
}
```

# *For vs While*

Normalmente o *while* é utilizado em situações que não sabemos quantas repetições acontecerão.

Em geral, o uso do *for* tende a ser mais comum.

# Funções

Funções na programação são muito similares à funções matemáticas (ou pelo menos deveriam ser), pois estas recebem **parâmetros** (ou **argumentos**) que são usados para determinar um **retorno**.

# Funções

Estrutura de uma função básica:

```
tipoRetorno nomeFunção (parâmetros) {  
    ...  
    return resultado;  
}
```

# Funções

```
int soma (int a, int b) {  
    return a + b;  
}
```

# Funções

Para chamar uma função:

*resultado = nomeFunção(parâmetros);*

# Funções

```
int r = soma(2, 3);
```

# Funções

É interessante tentar abstrair o código em várias funções simples e curtas, com nomes claros.

Isto torna o código mais fácil de ser entendido, corrigido e mantido, além de mais robusto.

# Escopo

O escopo de uma variável é onde aquela variável pode ser utilizada.

É uma boa prática declarar a variável no **menor escopo possível**, isso evita conflitos com variáveis de mesmo nome.

# Escopo

```
if (true) {  
    int a = 2;  
}  
System.out.println(a);|
```

# Escopo

Neste caso, o programa não consegue encontrar a variável “a”, pois ela está declarada dentro do escopo do *if* apenas.

# Exceções

Exceções são erros que acontecem no código, estes podem ser esperados (eg. divisão por 0) ou não (eg. erro de rede).

# Exceções

Há situações em que um programa não pode ser interrompido pela ocorrência de um erro, para isso, existe uma estrutura de controle de erros na maioria das linguagens, que começa com *try*.

# Exceções

```
try {  
    (trecho que pode dar errado)  
} catch (tipoDaExceção nomeProvisório) {  
    (tratamento do erro)  
} finally {  
    (bloco opcional que sempre é executado ao final, independente se houve erro ou não)  
}
```

# Exceções

Exemplo:

```
try {  
    int n = 2/0;  
} catch (Exception e) {  
    System.out.println(e);  
}
```

# Exceções

```
int[] n = new int[10];  
  
try {  
    n[132112] = 10;  
} catch (ArrayIndexOutOfBoundsException e) {  
    System.out.println("posição inválida");  
} finally {  
    System.out.println("eu sou opcional!");  
}
```

# Estruturas de Dados

Estruturas de dados são capazes de armazenar diversos valores, normalmente são dinâmicas e possuem diversas implementações (fila, pilha, lista ligada, tabelas Hash, árvores, grafos...).

# *Array*

O array é a mais simples das estruturas de dados, ele tem tamanho fixo e serve para armazenar um número pré-determinado de variáveis do mesmo tipo.

# *Array*

Para criar um *array*:

*tipo[ ] nome = new tipo[tamanho]*

# *Array*

Para acessar uma posição:

*nome[posição]*

# *Array*

```
int[] arr = new int[10];  
arr[0] = 2;  
arr[9] = 3;  
arr[10] = 4;
```

# *Array*

Rpare que a última linha causará um erro do tipo “*ArrayIndexOutOfBoundsException*”. Que pode ser traduzido como “índice do array fora dos limites”.

# *Array*

A razão disso é que a indexação (acesso às posições) começa em 0, portanto, um array de 10 posições termina na posição 9.

A posição 10 não existe, por isso resulta em um erro.

# Matrizes

Para criar uma matriz, basta colocar mais uma dimensão no *array*, da seguinte forma:

```
int[][] matrix = new int[3][5];
```

# Matrizes

Um *array* pode conter inúmeras dimensões, embora normalmente não haja necessidade para tal.

# *String*

*Strings* são basicamente *arrays* de caracteres, e  
são uma classe no Java.

# *String*

```
String s = "hello";
```

# Lista

Uma lista é basicamente um *array* de tamanho dinâmico, ou seja, é possível adicionar e remover elementos dela, e seu tamanho muda.

# Lista

Declaração:

*ArrayList<Tipo> nome = new ArrayList<>();*

# Lista

Métodos mais usados:

*.add(elemento)* <- adiciona elemento à lista

*.size()* <- retorna o tamanho da lista

*.clear()* <- remove todos os elementos da lista

*.get(posição)* <- retorna o elemento da posição

# Lista

Exemplo:

```
ArrayList<Integer> l = new ArrayList<>();
l.add(2);
l.add(5);
for (int i = 0; i < l.size(); i++) {
    System.out.println(l.get(i));
}
```

# HashMap

O *HashMap* é uma estrutura feita utilizando a técnica de *Hashing*.

Ele tem o objetivo de fazer uma associação entre 2 valores de qualquer tipo.

# HashMap

Sua principal vantagem é o desempenho, buscar um valor em um *HashMap* tem complexidade, teoricamente, sempre igual à 1, enquanto que a complexidade de se encontrar um valor em uma lista ou *array* aumenta de acordo com seu tamanho.

# HashMap

Declaração:

*HashMap<tipoChave, tipoValor> nome = new HashMap<>();*

# HashMap

Principais métodos:

<code>.put(chave, valor)</code>	<- insere um par de <chave, valor>
<code>.get(chave)</code>	<- retorna o valor associado à chave
<code>.clear()</code>	<- remove todos os pares
<code>.containsKey(chave)</code>	<- checa se a chave existe
<code>.isEmpty()</code>	<- checa se está vazio

# HashMap

Exemplo:

```
HashMap<String, Double> constantes = new HashMap<>();  
constantes.put("pi", 3.14);  
constantes.put("e", 2.71);  
System.out.println(constantes.get("pi"));
```

# Algoritmos

Para concluir a parte de algoritmos, vamos fazer um código que utiliza todos os conceitos vistos e que é amplamente utilizado.

Vamos fazer uma busca binária!

# Algoritmos

Para concluir a parte de algoritmos, vamos fazer um código que utiliza todos os conceitos vistos e que é amplamente utilizado.

Vamos fazer uma busca binária!

# Algoritmos

```
public static void main(String[] args)
{
    int[] numeros = {1, 2, 4, 6, 8, 11, 15, 19, 20};

    int posicao = binarySearch(numeros, 4);
    System.out.println(posicao);
}

public static int binarySearch(int[] arr, int num) {
    int i = 0;
    int j = arr.length;

    while (i < j) {
        int middle = (i + j)/2;

        if (arr[middle] == num) {
            return middle;
        }
        if (arr[middle] > num) {
            j = middle - 1;
        } else {
            i = middle + 1;
        }
    }

    return -1;      // nao achou
}
```

# Programação Orientada a Objeto

# Programação Orientada a Objeto

- O que é Orientação a Objeto
- Classes e objetos
- Atributos e métodos
- Modificadores de acesso e encapsulamento
- Construtores e overloading
- Herança e classes abstratas
- Overriding
- Polimorfismo

# Programação Orientada a Objeto

Orientação a Objeto é um paradigma de programação, uma forma de se programar, que ficou muito popular por facilitar a organização e reaproveitamento de projetos muito grandes.

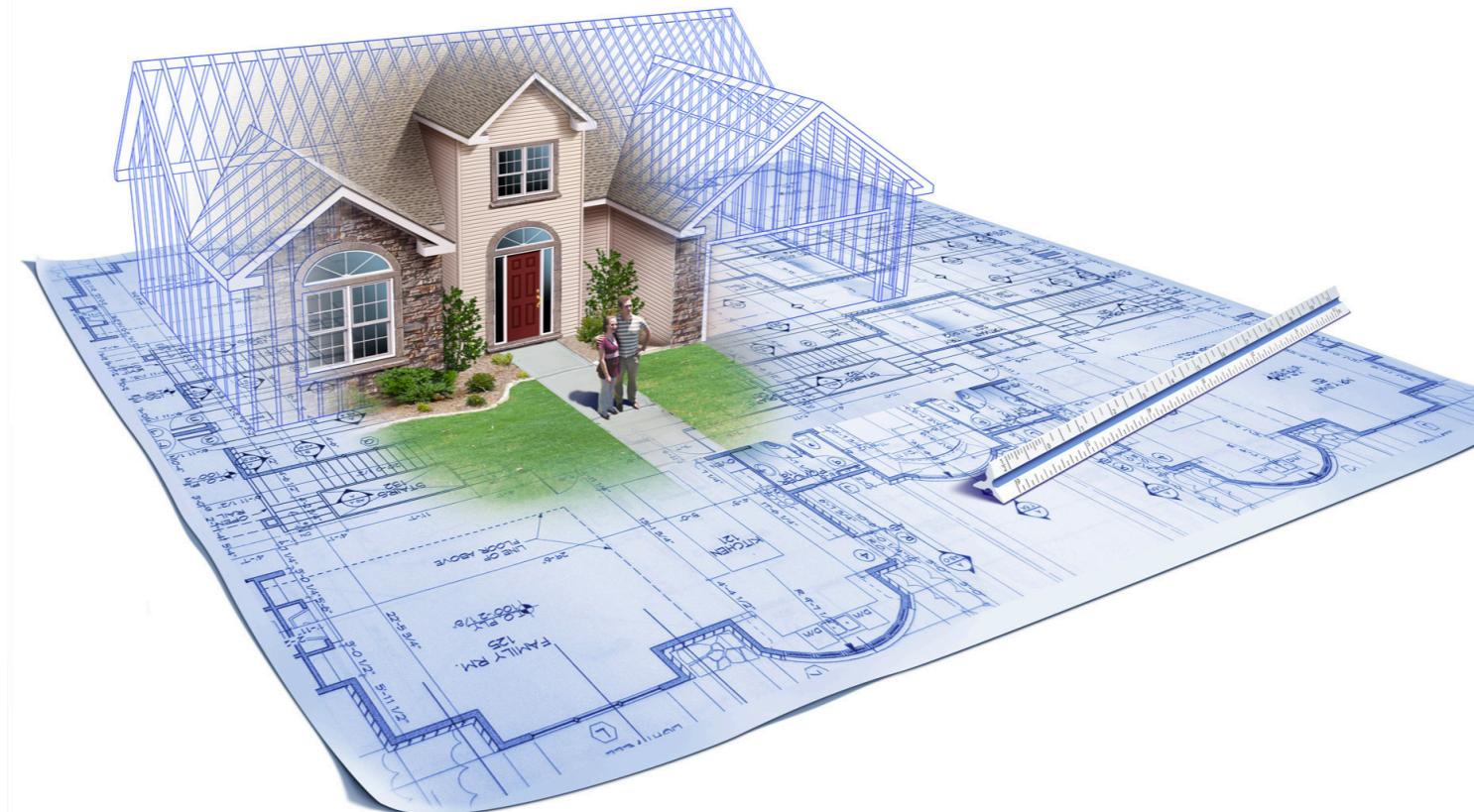
Outros paradigmas bem conhecidos são: procedural (C, Python) e funcional (Kotlin, Python).

# Programação Orientada a Objeto

Em POO, o programa é organizado por meio de **classes**, cada uma determina as características e funcionamento de um tipo de **objeto**.

# Classes e objetos

Podemos pensar em **classes** como a planta de uma casa, enquanto que o **objeto** seria a casa finalizada.



# Classes e objetos

Classe:

```
public class Pessoa
{
    // características (atributos)
    String nome;
    int idade;
    String cpf;

    // ações (métodos)
    public void andar(){
        // anda
    }

    public void respira() {
        // respira
    }

    public void come() {
        // come
    }
}
```

# Classes e objetos

Objeto:

```
public static void main(String[] args)
{
    Pessoa pessoa = new Pessoa();
    pessoa.nome = "Joao";
    pessoa.idade = 25;
    pessoa.cpf = "12312312312";

    pessoa.respirar();
}
```

# Classes e objetos

O objeto é como uma variável do tipo de sua classe, porém com atributos próprios.

# Atributos e métodos

Atributos são características específicas de um objeto. No exemplo anterior, seriam o *nome*, *idade* e *CPF*.

Basicamente, são como variáveis internas do objeto, que definem **propriedades** que ele possui.

# Atributos e métodos

Métodos são **funções** que especificam **ações** que o objeto pode realizar.

No exemplo anterior, os métodos são *andar, respirar e comer*.

# Atributos e métodos

Outro exemplo:

```
public class Carro
{
    String placa;
    String marca;
    String modelo;
    String cor;
    int kilometragem;
    int numCilindros;

    public void Acelerar(){
        bombearGasolina();
        abreValvulasDeInjecao();
        // ...
    }

    private void bombearGasolina() {
        // joga a gasolina pro motor
    }

    private void abreValvulasDeInjecao() {
        // permite q a gasolina entre na camara de combustao
    }
}
```

# Modificadores de Acesso

Modificadores de acesso são *keywords* que estabelecem quem pode interagir com os atributos e métodos de uma **classe**.

Estes devem ser especificados antes do tipo na criação de um atributo ou método.

# Modificadores de Acesso

Estes são:

- *private*
- *protected*
- *public*

# Modificadores de Acesso

## *Private:*

Apenas o próprio objeto tem acesso à atributos e métodos *private*. Outros objetos **não** conseguem vê-los.

# Modificadores de Acesso

*Private:*

Além de ser importante do ponto de vista de segurança definir quais atributos devem ser privados, isto também ajuda a evitar que métodos sejam utilizados de forma inapropriada.

# Modificadores de Acesso

## *Protected:*

Atributos e métodos do tipo *protected* só podem ser acessados pela própria classe **e por seus filhos**. Este conceito será explicado em detalhes mais a frente, em **herança**.

# Modificadores de Acesso

*Public:*

Atributos e métodos públicos são completamente abertos, portanto, qualquer objeto pode interagir com os mesmos.

# Encapsulamento

**Encapsulamento** é um conceito comum em orientação a objeto. Trata-se de isolar uma variável, tornando-a *private*, e disponibilizando métodos *public* que sejam capazes de interagir com esta variável, porém garantindo que seu acesso seja feito corretamente.

Estes métodos são chamados de *Getters* e *Setters*.

# Getter e Setters

Exemplo:

```
public class Pessoa
{
    // atributos
    public String nome;
    public int idade;
    private String cpf;

    public String getCpf(Pessoa requisitante) {
        if (hasAuthorization(requisitante)) {
            return cpf;
        } else {
            return "pessoa não autorizada";
        }
    }

    private boolean hasAuthorization(Pessoa p) {
        // checks if p has authorization
        return true;
    }
}
```

# Getter e Setters

Exemplo:

```
public class Pessoa
{
    // atributos
    public String nome;
    public int idade;
    private String cpf;
    private int salario;

    public void setSalario(int salario) {
        if (salario > 0) {
            this.salario = salario;
        }
    }

    public int getSalario() {
        return salario;
    }
}
```

# **Getter e Setters**

Boas práticas:

- Definir os nomes dos métodos *get/set* sempre como: *getAtributo* e *setAtributo*.
- **NÃO** é necessário encapsular **TODOS** os atributos da classe. Encapsule somente aqueles que requerem alguma ação no *getter* ou *setter*.

# Construtores e Overloading

Construtores são métodos utilizados para inicializar um objeto.

Por meio de um construtor, podemos criar um objeto utilizando uma função, e passando todos os parâmetros daquele objeto.

# Construtores e Overloading

Construtores são definidos da seguinte maneira:

```
public NomeDaClasse(parâmetros) {  
    (...)  
}
```

# Construtores e Overloading

Exemplo:

```
public class Pessoa
{
    // atributos
    public String nome;
    public int idade;
    private String cpf;
    private int salario;

    public Pessoa(String nome, int idade, String cpf, int salario) {
        this.nome = nome;
        this.idade = idade;
        this.cpf = cpf;
        this.salario = salario;
    }
}
```

# Construtores e Overloading

A *keyword* **this** faz referência ao próprio objeto, em outras palavras, a variável “nome” dentro do construtor, é o parâmetro recebido pela função. Já a variável “`this.nome`”, é o atributo nome do próprio objeto.

# Construtores e Overloading

*Overloading* de métodos significa criar o mesmo método porém recebendo parâmetros diferentes, e executando de forma diferente.

# Construtores e Overloading

Exemplo:

```
public class Pessoa
{
    // atributos
    public String nome;
    public int idade;
    private String cpf;
    private int salario;

    public Pessoa(String nome, int idade, String cpf, int salario) {
        this.nome = nome;
        this.idade = idade;
        this.cpf = cpf;
        this.salario = salario;
    }

    public Pessoa() {
        this("", 0, "", 0);
    }
}
```

# Construtores e Overloading

No exemplo anterior, vemos que foi feito um *overloading* do construtor da classe Pessoa.

Por meio do construtor Pessoa(), podemos criar um objeto da classe Pessoa sem passar nenhum atributo.

# Construtores e Overloading

É muito comum criar *overloading* de construtores vazios para testar uma aplicação.

Podemos ver mais exemplos de *overloading* a seguir:

# Construtores e Overloading

```
public class Calculadora
{
    public int soma(int a, int b) {
        return a + b;
    }
    public double soma(double a, double b) {
        return a + b;
    }
    public int soma(int a, int b, int c) {
        return a + b + c;
    }
}
```

# Construtores e Overloading

Neste caso temos 3 definições diferentes do método “soma”, com parâmetros e execuções diferentes. Importante notar entretanto que todos são somas, portanto, não há motivo para nomeá-los diferentemente.

# Herança

Herança é um conceito de POO que nos permite criar classes **filhas** de outras classes, de forma que a classe **filha herde** todos os métodos e atributos da classe **mãe**.

# Herança

Há certos casos em que diferentes classes compartilham atributos e métodos. Podemos então reduzir a repetição de código por meio de herança.

# Herança

Exemplo: Um aluno do Inatel é capaz de acessar os módulos de aluno do portal do aluno.

Um monitor, que também é aluno, é capaz de fazer tudo que o aluno faz, e também é capaz de acessar o módulo do monitor, registrar frequências e notas de suas turmas.

Podemos representar o aluno e o monitor da seguinte forma:

# Herança

```
public class Aluno
{
    public String nome;
    public int matricula;
    private int senha;

    public Aluno(String nome, int matricula, int senha) {
        this.nome = nome;
        this.matricula = matricula;
        this.senha = senha;
    }

    public void verHorario() {
        // exibe o horario do aluno
    }

    public void verNotas() {
        // exibe as notas
    }
}
```

# Herança

```
public class Monitor extends Aluno
{
    public Monitor(String nome, int matricula, int senha) {
        super(nome, matricula, senha);
    }

    public void registrarFrequencia() {
        // abre o modulo de registrar frequencia
    }
}
```

# Herança

```
public static void main(String[] args)
{
    Monitor monitor = new Monitor("marquin", 1, 2312);
    monitor.verHorario();
}
```

# Herança

Note que, apesar de o método “verHorario” não existir dentro de monitor, este pode ser chamado, pois foi **herdado** da classe **aluno**.

# Herança

Dentro do construtor do monitor, vemos o uso da *keyword super*. Super é usado para acessar métodos da classe mãe. Nesse caso, foi utilizado para chamar o construtor da classe mãe.

# Classes Abstratas

Classes abstratas são classes que **não podem ser instanciadas**. Isto significa que não é possível criar objetos destas classes.

O objetivo de classes abstratas, é gerar um modelo mais abstrato que diferentes classes possam herdar, reduzindo o código.

# Classes Abstratas

Por exemplo: desejamos criar duas classes para um jogo, uma delas representará cachorros, a outra gatos. Existem semelhanças muito grandes entre os dois, portanto, podemos abstrair vários atributos e métodos para uma classe abstrata, que podemos chamar de “Animal”.

# Classes Abstratas

```
public abstract class Animal
{
    public String raca;
    public int idade;

    public abstract void comer();
    public abstract void falar();
}
```

# Classes Abstratas

```
public class Cachorro extends Animal
{
    public Cachorro(String raca, int idade) {
        this.raca = raca;
        this.idade = idade;
    }

    @Override
    public void comer() {
        // cachorro come
    }

    @Override
    public void falar() {
        System.out.println("AU AU");
    }
}
```

# Classes Abstratas

```
public class Gato extends Animal
{
    public Gato(String raca, int idade) {
        this.raca = raca;
        this.idade = idade;
    }

    @Override
    public void comer() {
        // gato come
    }

    @Override
    public void falar() {
        System.out.println("miau");
    }
}
```

# Classes Abstratas

Para tornar uma classe abstrata é necessário usar a *keyword abstract* em sua declaração, como pode ser visto na classe “Animal”.

# *Overriding*

Provavelmente você notou as linhas “`@override`” no exemplo anterior.

A anotação `override` é usada para criar um método igual de uma classe mãe, porém sobre-escrevendo sua funcionalidade.

# *Overriding*

Dessa forma, quando o método for chamado, o código executado será da classe filha.

Neste caso, utilizamos o *override* para gerar os métodos definidos pela classe mãe abstrata.

# *Overriding*

Também é possível usar *overriding* para sobre-escrever uma função já definida de uma classe mãe, que pode não ser abstrata.

No exemplo do aluno e monitor, seria possível mudar o comportamento de certos métodos da classe **aluno**, dentro da classe **monitor**, utilizando *overriding*.

# Polimorfismo

Polimorfismo é uma técnica em que podemos executar o mesmo método, em classes diferentes, da mesma maneira, com resultados diferentes.

# Polimorfismo

Podemos ver isto no último exemplo dos animais. Tanto gatos quanto cachorros, implementam o método “comer” e “falar”, porém *comem* e *falam* de maneiras diferentes.

# Polimorfismo

Entretanto, como as funções são herdadas da classe abstrata “Animal”, podemos tratar tanto gatos quanto cachorros da mesma maneira, como no exemplo a seguir:

# Polimorfismo

```
public static void main(String[] args)
{
    Animal[] animais = new Animal[4];
    animais[0] = new Cachorro("Pinscher", 12);
    animais[1] = new Gato("Persa", 7);
    animais[2] = new Gato("Garfield", 14);
    animais[3] = new Cachorro("Husky", 4);

    for (int i = 0; i < animais.length; i++){
        System.out.println(animais[i].raca);
        animais[i].falar();
    }
}
```

# Polimorfismo

---

run:

Pinscher

AU AU

Persa

miau

Garfield

miau

Husky

AU AU

BUILD SUCCESSFUL

# Polimorfismo

Como tanto gatos quanto cachorros, são classes filhas da classe Animal, conseguimos salvar todos em um array do tipo Animal. Não seria possível por exemplo salvar um Cachorro num array de gatos, e vice versa.

# Polimorfismo

Além disso, como “falar” está declarado como um método abstrato da classe “Animal”, quando chamamos o método, ele executa a versão sobre-escrita das classes filhas, gerando resultados diferentes para o gato e o cachorro.

Isto é polimorfismo.