

## Java e Orientação a Objetos



## Contato



[samuel.souza@gec.inatel.br](mailto:samuel.souza@gec.inatel.br)



<https://github.com/Saam97>

Material e exemplos no [github](https://github.com/Saam97)!

### Ementa

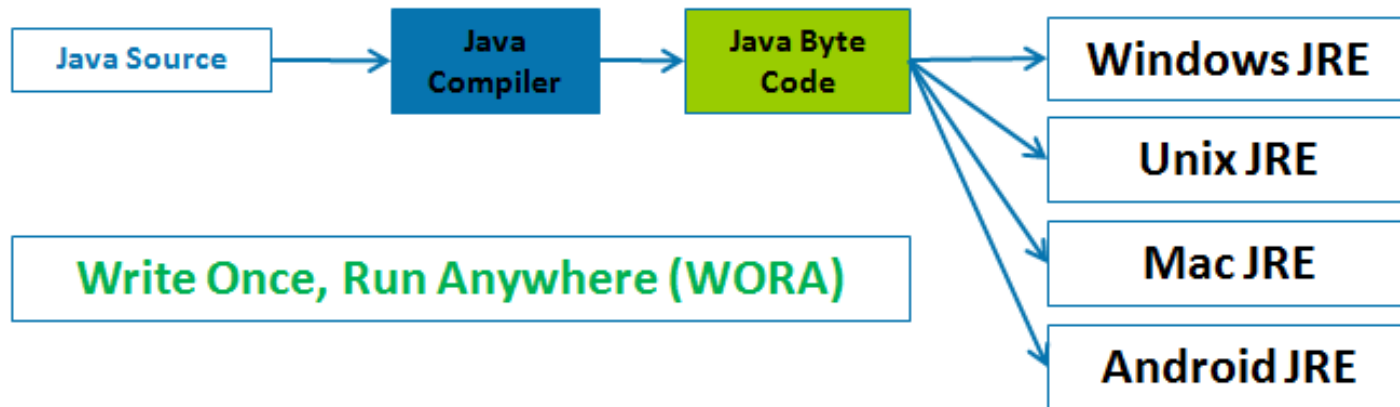
- Introdução
- Algoritmos e Estruturas de dados
- Programação Orientada a Objetos
- Threads
- Interface Gráfica com o Usuário

## Introdução

- O Java nasceu na década de 90 com a proposta de ser uma “linguagem universal”  
*“Write once, run anywhere” ou “Escreva uma vez, execute em qualquer lugar”*
- Completamente orientada a objeto
- Java Virtual Machine (JVM)

## Introdução

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



## Introdução

### IDEs



**NetBeans**



**IntelliJ**



**eclipse**

## Introdução

- Download JDK – Kit de Desenvolvimento Java

*(necessário criar conta na Oracle)*

<https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>

- Download NetBeans 8.2

<https://netbeans.org/downloads/8.2/>

## Introdução

### Hello World

1. Clique em “arquivo” -> “novo projeto”
2. “Aplicação Java”
3. Dê um nome ao seu projeto (sem espaços)
4. “Finalizar”



## Introdução

### Hello World

```
public class HelloWorld {  
  
    public static void main(String[] args) {  
        // TODO code application logic here  
        System.out.println("Hello World");  
    }  
  
}
```

## Introdução

### Hello World

O método ***public static void main(String[] args)*** é o método “main” do nosso projeto e assim como no c++ ele é chamado no início da execução.

## Variáveis

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

```
tipoVar nomeVar;
```

Também é possível adicionar um valor inicial, mas não é obrigatório

## Variáveis

- **Variáveis Primitivas:** armazenam valores

```
int valor;                double pi = 3.14159;
```

- **Variáveis de Referência:** armazenam um endereço de memória

```
String palavra = "Oi";    Carro fusca;
```

## Variáveis

### Tipos de variáveis primitivas

Inteiros	[	Byte	
	-	Short	
	-	Int	
	-	Long	
Ponto flutuante	[	Float	
	-	Double	
Lógico	[	Boolean	
Caractere	[	Char	

```
int a = 25;  
float b;  
char c;  
boolean d = true;  
double e;
```

## Variáveis

## Operadores em Java

- Multiplicação: \*
- Divisão: /
- Soma *ou* Concatenação: +
- Subtração: -
- Resto da Divisão: %

## Variáveis

### Operadores em Java (Comparação)

- Maior que: >
- Menor que: <
- Maior igual: >=
- Menor igual: <=
- Diferente: !=
- Igual: ==
- Lógica E: &&
- Lógica OU: ||

## Casting

Quando for necessário converter um tipo primitivo em outro, realizamos uma operação chamada casting.

```
double pi = 3.14159;  
int p = (int) pi;
```

```
System.out.println(p);
```

 *Saída de dados*

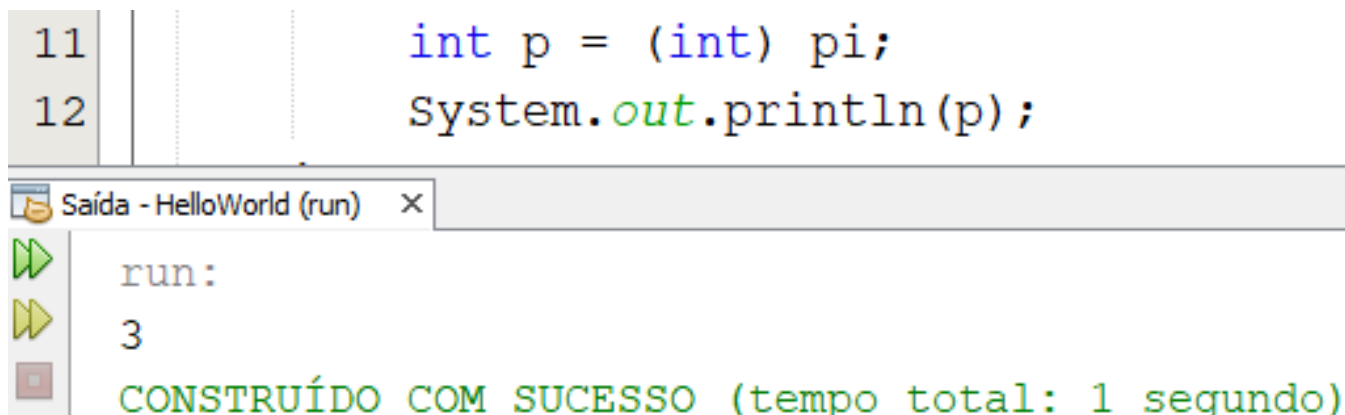
Nem todas as conversões precisam realizar o *casting*.



## Saída de dados

A saída padrão do Java é a função:

`System.out.println();`



The screenshot displays a Java IDE interface. The top section shows two lines of code: line 11 with `int p = (int) pi;` and line 12 with `System.out.println(p);`. Below the code editor, a window titled 'Saída - HelloWorld (run)' is open, showing the execution output. The output consists of three lines: 'run:' followed by the number '3' on the next line, and a final green-colored line stating 'CONSTRUÍDO COM SUCESSO (tempo total: 1 segundo)'.

```
11      int p = (int) pi;  
12      System.out.println(p);
```

Saída - HelloWorld (run) ×



run:  
3  
CONSTRUÍDO COM SUCESSO (tempo total: 1 segundo)

## Saída de dados

A função *print* reconhece todos os tipos primitivos de variáveis.

Mais adiante veremos como *printar* uma classe.

### Atalhos:

- `sout + TAB`  Escreve automaticamente a função
- `soutv + TAB`  Maneira “preguiçosa” de printar uma variável

## Entrada de dados

Existem diversas formas de realizar a entrada de dados no Java. Vamos utilizar a mais simples, que é criando um *objeto* da classe *Scanner*.

```
Scanner teclado = new Scanner(System.in);  
int num = teclado.nextInt();  
System.out.println("num = " + num);
```

## Entrada de dados

É necessário realizar um *import* da classe, pois a mesma não faz parte do pacote padrão. As IDEs normalmente são capazes de reconhecer a necessidade de importar a classe e fazem isso automaticamente.

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

## Entrada de dados

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

- `nextInt();`                      Leitura de um valor do tipo `int`
- `nextBoolean();`                Leitura de valores lógicos
- `nextFloat();`                    Leitura de valores do tipo `float`
- `nextLine();`                    Leitura de valores de texto (`String`)

## Exercício 1

Crie um novo projeto e imprima o valor inteiro de uma variável do tipo *double* a partir da entrada de dados do usuário.

## Estruturas de decisão

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

Exemplo:

```
Scanner teclado = new Scanner(System.in);  
int valor = teclado.nextInt();  
  
if (valor < 5) {  
    System.out.println("Valor menor que 5");  
} else if ( valor > 10 ) {  
    System.out.println("Valor maior que 10");  
} else {  
    System.out.println("Valor entre 5 e 10");  
}
```

## Estruturas de decisão – if/else

```
If (condição){
```

```
    ...
```

```
} else if ( outra condição ){
```

```
    ...
```

```
} else {
```

```
    ...
```

```
}
```



## Estruturas de decisão – if/else

- O uso do “else if” e “else” são opcionais.
- Podem ser usados vários “else if” conforme o necessário.
- O uso de “else if” é diferente do uso de vários “if”.

## Estruturas de decisão – switch-case

O switch case faz uma comparação com uma variável e dependendo do seu valor ele executa um trecho de código.

É uma ótima estrutura para criar um *menu de opções*.

## Estruturas de decisão – switch-case

```
Scanner teclado = new Scanner(System.in);  
// Pega o valor digitado e salva na variável  
String aux = teclado.nextLine();  
// Armazena somente a primeira letra da variável aux  
char conceito = aux.charAt(0);
```

```
switch (conceito) {  
    case 'A':  
        System.out.println("Média boa!");  
        break;  
    case 'B':  
        System.out.println("Média Razoável");  
        break;  
    case 'C':  
        System.out.println("Média ruim!");  
    default:  
        System.out.println("Média inválida");  
}
```

Caso nenhuma das opções seja a correta o bloco default é executado

## Exercício 2

Crie uma aplicação que o usuário irá inserir dois valores de notas (NP1 e NP2) e calcule a média final.

- Caso a média final for maior que 60, escreva “Aprovado”;
- Caso a média final for menor que 30, escreva “Reprovado”;
- Caso a média esteja entre 30 e 60, escreva “NP3” e peça para o usuário inserir um novo valor (nota da NP3) e calcule novamente a média. Caso seja maior que 50 escreva “Aprovado na NP3” e caso contrário escreva “Reprovado na NP3”.

## Estruturas de repetição

As estruturas de repetição servem para executar repetidamente um trecho de código, enquanto uma condição for verdadeira

Exemplos:

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

```
while (valor < 10) {  
    System.out.println(valor);  
    valor++;  
}
```

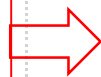
## Estruturas de repetição – while

Enquanto esta  
condição for verdadeira



```
while (valor < 10) {
```

Executa estes  
comandos



```
{ System.out.println(valor);  
  valor++;
```

```
}
```

Enquanto a variável “valor” for menor que 10, o código será executado.

## Estruturas de repetição – for

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

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

Variável de  
iteração

Valor inicial  
(geralmente  
zero)

Condição

Incremento  
da variável

## Exercício 3

Imprima todos os números primos de 1 até 50.

*Dica: faça as comparações com 2, 3, 5 e 7.*



## Estruturas de dados

As estruturas de dados são capazes de armazenar diversos valores.

Exemplos: pilha, lista ligada, fila, tabela Hash, árvores.

## Estruturas de dados

- Arrays

O *array* ou *vetor* é o mais simples tipo de estrutura de dados, ele possui um tamanho fixo e serve para armazenar um número pré-determinado de variáveis de mesmo tipo.

Para criar um array:

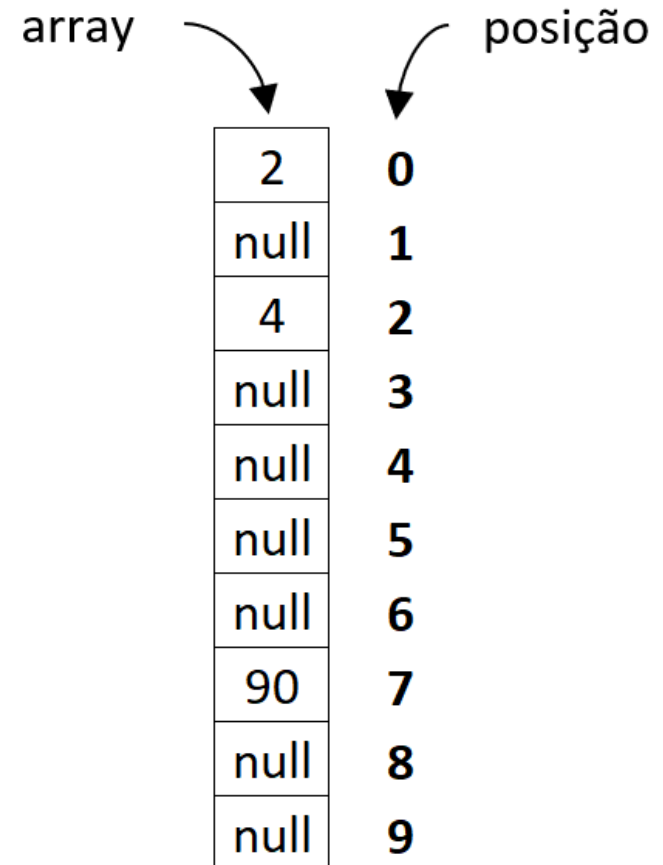
```
tipo[] nomeArray = new tipo[tamanho];
```

Para acessar uma posição:

```
nomeArray[posição]
```

## Estruturas de dados

```
int[] array = new int[10];  
array[0] = 2;  
array[2] = 4;  
array[7] = 90;  
  
array[10] = 6;
```



## Estruturas de dados

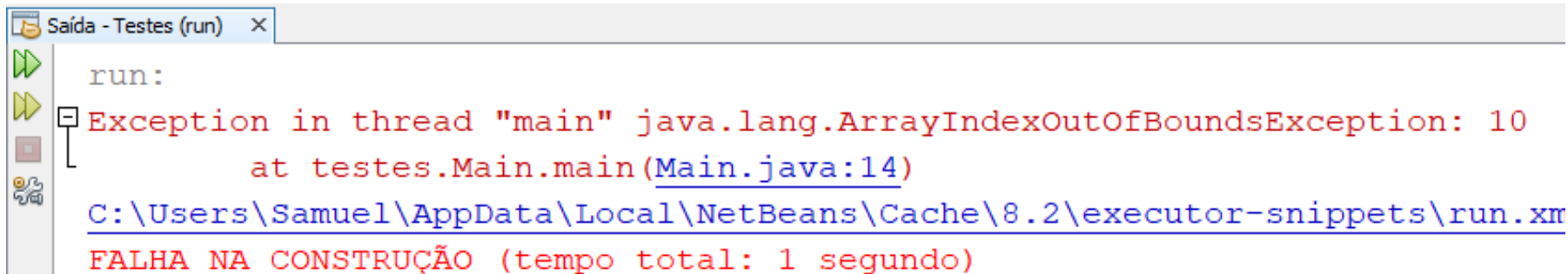
- Arrays

Todo array começa na posição zero, ou seja, seu índice vai de 0 até o seu tamanho - 1. (Neste caso de 0 a 9)

Portanto se tentarmos acessar a posição 10 irá ocorrer um erro, ou como são chamados em Java, uma *Exceção*.

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

⇒ `array[10] = 6;`



```
Saída - Testes (run) x
run:
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
    at testes.Main.main(Main.java:14)
C:\Users\Samuel\AppData\Local\NetBeans\Cache\8.2\executor-snippets\run.xml
FALHA NA CONSTRUÇÃO (tempo total: 1 segundo)
```

## Estruturas de dados

- Matrizes

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

```
int[] [] matriz = new int[10][5];
```

## Estruturas de dados

- String

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

```
String minhaString = "Hello World";
```

## Estruturas de dados

- Lista

Uma lista é basicamente um *array* de tamanho dinâmico, ou seja, ela cresce ou diminui conforme elementos são adicionados e removidos.

Exemplo:

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



*\*Não pode ser primitivo\**

## Estruturas de dados

- Lista

Métodos mais usados para manipular uma lista:

.add(elemento);      -> adiciona um elemento na lista

.size();              -> retorna o tamanho da lista

.remove(elemento); -> remove um elemento da lista

.get(posição);      -> retorna o elemento da posição



## Estruturas de dados

- Lista

```
ArrayList<Integer> lista = new ArrayList<>();  
lista.add(2);  
lista.add(25);  
for (int i = 0; i < lista.size(); i++) {  
    System.out.println( lista.get(i) );  
}
```

## Exercício 4

Simule um Campo minado por meio de uma Matriz 2x2 e coloque 1 bomba em uma posição randômica dessa Matriz. Depois, o usuário deverá caminhar pelo Campo Minado e aquele que conseguir caminhar por todas as posições sem bombas zera o jogo.

### *Dicas:*

- *Gerar um numero aleatório: `Random randomGenerator = new Random();`*
- *`numAleatorio = randomGenerator.nextInt(2);`*
- *Faça um Loop que questione o jogador sobre a posição da Matriz que ele deseja caminhar;*
- *`Scanner teclado = new Scanner(System.in);`*
- *`int valor = teclado.nextInt();`*

## Tratamento de Erros

Exceções são erros que acontecem no código, estes podem ser esperados, como por exemplo uma divisão por zero, ou não, como por exemplo um erro na rede.

## Tratamento de Erros

Há situações em que um programa não pode ser interrompido pela ocorrência de um erro, e para isto existe uma estrutura de controle de erros.

## Tratamento de Erros

No Java funciona da seguinte forma:

```
try{  
    (Trecho que pode dar errado)  
} catch (tipoDaException nomeProvisorio) {  
    (Tratamento do erro)  
} finally {  
    (Bloco opcional que sempre é executado ao final,  
    independente se houve erro ou não)  
}
```

## Tratamento de Erros

### Exemplo

```
int[] n = new int[10];  
try {  
    n[4515] = 10;  
} catch (ArrayIndexOutOfBoundsException e) {  
    System.out.println("Posição Inválida !");  
} finally {  
    System.out.println("Eu sou opcional!");  
}
```

# Programação Orientada a Objetos

## Programação Orientada a Objetos

- O que é Orientação a Objetos
- Classes e Objetos
- Atributos e métodos
- Modificadores de acesso e encapsulamento
- Herança e Classes Abstratas
- Polimorfismo



## Programação Orientada a Objetos

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 Objetos

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

## Programação Orientada a Objetos

Podemos pensar em **classes** como uma receita de bolo, e um **objeto** como sendo o bolo.

A classe pode ser entendida como uma especificação do objeto.

A partir dela, é possível criar vários objetos (bolos) diferentes.

## Programação Orientada a Objetos

### Exemplo de Classe:

```
public class Pessoa {  
    // Características (atributos)  
    String nome;  
    int idade;  
    String cpf;  
  
    // Ações (métodos)  
    public void comer() {  
        // comer  
    }  
  
    public void respirar() {  
        // respirar  
    }  
}
```

## Programação Orientada a Objetos

### Exemplo de Objeto

```
public static void main(String[] args) {  
  
    Pessoa pessoa = new Pessoa();  
    pessoa.nome = "Samuel";  
    pessoa.idade = 21;  
    pessoa.cpf = "123456789-10";  
  
    pessoa.respirar();  
}
```

# Programação Orientada a 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.

*O que toda Pessoa tem?*

## Atributos e métodos

Métodos são ações ou comportamentos de uma classe.  
No exemplo anterior, os métodos são *andar*, *comer* e *respirar*.

*O que toda Pessoa faz?*



## Exercício 5

Implemente uma Calculadora que realize as seguintes operações:

- Soma
- Subtração
- Divisão
- Multiplicação
- Potenciação
- Raiz Quadrada

*Dica: utilize alguns métodos da classe Math*

## Pilares da Orientação a Objetos

A programação Orientada a Objetos se baseia em quatro pilares

- Encapsulamento
- Herança
- Polimorfismo
- Composição (Abstração)



# Encapsulamento

## Modificadores de Atributos

No Java existem algumas *Keywords* que identificam o tipo de atributo de uma variável.

- Final – Indica que aquele atributo é uma CONSTANTE e seu valor não pode ser alterado;
- Static – Faz com que o atributo seja compartilhado por todos os objetos da classe.

## Modificadores de Atributos

```
public class Matricula {  
  
    final int TOTAL_CREDITOS = 27;  
    static String faculdade = "Inatel";  
    String curso;  
    int numMatricula;  
    String email;  
}
```

O número de créditos sempre é o mesmo (Constante)

Não importa qual seu curso, você está matriculado no Inatel (mesmo valor para todos os objetos matricula)

Os demais valores variam de acordo com cada objeto

## Modificadores de Acesso

São palavras-chave que definem quem pode “ver” os atributos e métodos de uma classe.

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

## Modificadores de Acesso

```
public class Conta {  
    private float saldo;  
    protected float limite;  
    public int numero;  
    public int agencia;  
    String cpf;  
  
    public void saque(float valor) {  
        saldo = saldo - valor;  
    }  
  
    public void deposito(float valor) {  
        saldo = saldo + valor;  
    }  
}
```

## Modificadores de Acesso

Podem ser de quatro tipos:

- Private
- Protected
- Public
- Default (sem nenhuma palavra)



## Modificadores de Acesso

- Private

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

```
private float saldo;
```

## Modificadores de Acesso

- Protected

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

```
protected float limite;
```

## Modificadores de Acesso

- Public

Este modificador de acesso deixa o atributo ou método completamente aberto, ou seja, qualquer objeto consegue interagir com atributos ou métodos deste tipo.

```
public int numero;  
public int agencia;
```

## Modificadores de Acesso

- Default (Padrão)

É o modificador de acesso padrão quando não definimos os atributos ou métodos com os modificadores citados. Estes podem ser acessados por outros objetos, porém não **podem ser acessados pelos seus filhos**.

```
String cpf;
```

## Modificadores de Acesso

Já que uma imagem vale mais que mil palavras ...

MODIFICADOR	CLASSE	MESMO PACOTE	PACOTE DIFERENTE (SUBCLASSE)	PACOTE DIFERENTE(GLOBAL)
Public				
Protected				
Default				
Private				

### Exemplo

É interessante  
que outros  
possam ver o  
número e  
agência

Não é interessante  
deixar o saldo como  
público

O mesmo para o limite

```
public class Conta {  
    private float saldo;  
    protected float limite;  
    {  
        public int numero;  
        public int agencia;  
        String cpf;  
    }  
  
    public void saque(float valor) {  
        saldo = saldo - valor;  
    }  
  
    public void deposito(float valor) {  
        saldo = saldo + valor;  
    }  
}
```

E que possam  
realizar  
tais métodos

## Modificadores de Acesso

Mas e se precisarmos alterar o valor de uma variável do tipo *private*?

E se precisarmos obter o valor desta variável em determinado momento?

## Encapsulamento

Trata-se de isolar uma variável, geralmente 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.



## Encapsulamento

```
// Getter: retorna o valor da variável saldo
public float getSaldo() {
    return saldo;
}

// Setter: muda o valor da variável saldo
public void setSaldo(float saldo) {
    if ( saldo >= 0 ) {
        this.saldo = saldo;
    }else {
        System.out.println("Novo saldo inválido !");
    }
}
```

## Encapsulamento

Não é necessário encapsular todos os atributos da classe. Isto varia de acordo com a sua lógica ou aplicação.

Usando o exemplo anterior:

Faz sentido alterar o saldo de uma conta a não ser por

```
public void setSaldo(float saldo) {  
    if ( saldo >= 0 ) {  
        this.saldo = saldo;  
    }else {  
        System.out.println("Novo saldo inválido !");  
    }  
}
```

## Construtor

O *Construtor* nada mais é que o método que é chamado **sempre** quando instanciamos um objeto. É possível passar alguns parâmetros no construtor para instanciar um objeto.

```
Computador pc = new Computador();
```



Construtor

Por padrão o construtor é *vazio* e possui o mesmo nome da Classe.

## Construtor

Por exemplo, todo computador obrigatoriamente tem um processador. Então, podemos passar como parâmetro um processador assim que criarmos o objeto.

```
public class Computador {  
    public double tamanhoTela;  
    public String processador;  
    public String placaVideo;  
    public int memoriaRam;  
    public boolean temSsd;  
  
    public Computador(String processador) {  
        this.processador = processador;  
    }  
}
```

## Construtor

constructor Computador in class Computador cannot be applied to given types;  
required: String  
found: no arguments  
reason: actual and formal argument lists differ in length  
----  
(Alt-Enter mostra dicas)

gs) {

Computador pc = new Computador();

## Construtor

```
public class Main {  
    public static void main(String[] args) {  
        Computador pc = new Computador("Core i7");  
  
        pc.memoriaRam = 8;  
        pc.placaVideo = "Não Possui";  
        pc.tamanhoTela = 15.6;  
        pc.temSsd = true;  
    }  
}
```

## A palavra reservada *this*

Notou algo diferente? A palavra reservada *this* é usada quando se deseja fazer referência ao próprio objeto.

Parâmetro da função  
↓  
`public Computador(String processador) {`  
Atributo do objeto → `this.processador = processador;`  
`}`

De modo geral, a variável “processador” é o parâmetro recebido pela função, enquanto a variável “this.processador” é o atributo *processador* do próprio objeto.

## Exercício 6

Na era dos piratas, o Governo Mundial está preocupado por quantos piratas estão a solta e quais a suas recompensas e pediu para que você desenvolva um *software* para auxiliá-los e manter a Terra Sagrada a salvo.

Crie uma aplicação que realize o cadastro de novos piratas e exiba todos os piratas já cadastrados por meio de um *Array*, onde cada pirata possui um nome, recompensa e tripulação e pode conquistar uma ilha. Não esqueça de contar quantos piratas já existem !!

*Dica: Utilize uma variável estática para contar quantos piratas existem. Não é necessário entrada de dados. Crie um array com um tamanho razoável.*



## Exercício 6

Desafio: Imprima o valor total das recompensas e as informações de todos os piratas.

# Herança

## Herança

Você já deve ter ouvido a expressão “ele herdou os olhos da mãe” ou “ela herdou a cor do cabelo do pai”.

Em POO também existe este conceito, porém ele se aplica nas classes. É possível criar classes **filhas** de outras classes, de tal forma que a classe **filha herde** todos os *atributos e métodos* 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 um aluno no portal do aluno. Um monitor, que também é aluno, é capaz de fazer tudo o que um aluno faz, e também é capaz de acessar o módulo de monitor (para registrar frequência, notas e etc) de sua turma.

Usando deste conceito de Herança, podemos representar o aluno e monitor da seguinte forma:

## Herança

```
public class Aluno {  
    public String nome;  
    public int matricula;  
    private int senhaPortal;  
  
    public Aluno(String nome, int matricula, int senhaPortal) {  
        this.nome = nome;  
        this.matricula = matricula;  
        this.senhaPortal = senhaPortal;  
    }  
  
    public void verHorario() {  
        // Mostra o horário do aluno  
    }  
  
    public void verNotas() {  
        // Mostra as notas do aluno  
    }  
}
```

### Herança

Dizemos que a classe Monitor é filha de Aluno através da palavra reservada *extends*

```
public class Monitor extends Aluno {  
  
    public Monitor(String nome, int matricula, int senhaPortal) {  
        super(nome, matricula, senhaPortal);  
    }  
  
    public void registrarFrequencia() {  
        // Abre o módulo de frequência  
    }  
  
}
```

## Herança

```
public static void main(String[] args) {  
  
    Monitor monitor = new Monitor("Samuel", 1, 1234);  
    monitor.verHorario();  
}
```

Note que, apesar do método “verHorario” não existir dentro da classe Monitor, este pode ser chamado pois foi **herdado** da classe **Aluno**.

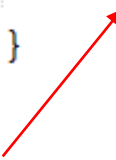


## Herança

Dentro do construtor do monitor vemos o uso da palavra *super*. Super é usado para acessar os atributos e métodos da classe mãe.

No nosso exemplo, foi usado para chamar o construtor da classe mãe.

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



## Exercício 7 – Parte I

Implemente um sistema para o gerenciamento de uma concessionária de carros em geral. Nesta concessionária são vendidos carros do tipo **Camaro, Uno, Gallardo e Fusca**.

Cada carro obrigatoriamente tem uma marca, cor, velocidade máxima, número de marchas, seu preço de venda, preço de compra e ano do modelo. O Camaro pode possuir *ar condicionado e teto solar*, o Uno *pode ou não ter escada*, um Gallardo *pode ser conversível* e o Fusca tem um *estado de conservação e pode ter um rádio*.

## Exercício 7 – Parte II

Para cada carro, crie um método para calcular seu preço de venda com base no ano de seu modelo, seguindo a tabela a baixo:

Ano	Valor de venda (Porcentagem sobre o valor de compra)
< 1980	10%
1980 a 1990	15%
1990 a 2000	17,50%
2000 a 2010	19%
2010 a 2020	25%

## Problemas

E se eu quiser armazenar todos estes carros dentro de um Array, eu terei que criar um Array para cada modelo diferente?

O código ficaria mais confuso, extenso e difícil de dar manutenção, e é aí que vem o **Polimorfismo**.

# Polimorfismo

## Polimorfismo

Polimorfismo significa mais de uma forma. Permite que um mesmo nome represente vários comportamentos ou objetos diferentes.



## Polimorfismo

### **Sobreposição**

Permite referenciar um objeto de uma subclasse como um objeto da superclasse.

### **Sobrecarga**

Permite ter vários métodos com o mesmo nome, diferenciando pela sua assinatura, quantidade e tipos de parâmetros.

## Polimorfismo

Pegando nosso exemplo anterior, um Aluno pode ser somente um Aluno ou também pode ser um monitor. Então, isso quer dizer que todo monitor é aluno, certo?



## Polimorfismo

Portanto, temos um exemplo do primeiro tipo de *Polimorfismo: Sobreposição.*

```
Aluno a1 = new Aluno();
```

```
Aluno a2 = new Monitor();
```

```
a1.verHorario();
```

```
a2.verHorario();
```

## Polimorfismo

Outro tipo de Polimorfismo é o de **Sobrecarga**, onde o mesmo método pode ser implementado de maneiras diferentes.

```
public void verHorario() {  
    // Mostra o horário do aluno  
}  
  
public void verHorario(Date dia) {  
    // Mostra o horário do aluno em um dia específico  
}  
  
public void verHorario(String periodo) {  
    // Mostra o horário de algum período (Manha/Tarde,  
}
```

Mesmo  
nome

Parâmetros  
diferentes

# Composição (Abstração)

## Classes Abstratas

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

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

## Classes Abstratas

### Exemplo

Imagine um sistema bancário onde só existam dois tipos de contas: PessoaFísica e PessoaJurídica. Podemos criar uma classe genérica chamada *conta* e a partir dela criar as outras.

## Classes Abstratas

### Exemplo

```
public class Conta {  
  
    protected float saldo;  
    protected float limite;  
    public int numero;  
    public int agencia;  
  
    public void saque(float valor) {  
        saldo = saldo - valor;  
    }  
  
    public void deposito(float valor) {  
        saldo = saldo + valor;  
    }  
}
```

## Classes Abstratas

### Exemplo

```
public class ContaPessoaFisica extends Conta {  
  
    private String nomePessoa;  
    private String cpf;  
  
    public void mostraInfo() {  
        // Mostra as Informações da Conta  
    }  
}  
  
public class ContaPessoaJuridica extends Conta {  
  
    private String razaoSocial;  
    private float limiteFinanceiro;  
  
    public void mostraInfo() {  
        // Mostra as informações da conta  
    }  
}
```

## Classes Abstratas

### Exemplo

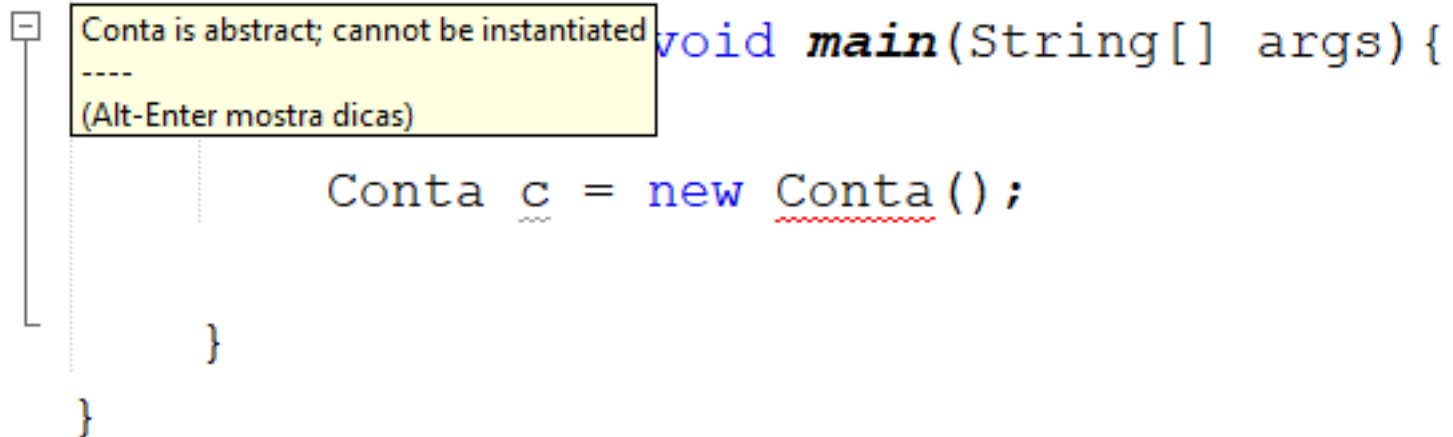
Não faz sentido existir uma *Conta* por si só. Ou ela deve ser de uma Pessoa física ou jurídica.

Para evitarmos este tipo de problema, basta transformar a Classe *Conta* em uma classe **Abstrata**.



## Classes Abstratas

### Exemplo



```
void main(String[] args) {  
    Conta c = new Conta();  
}  
}
```

Conta is abstract; cannot be instantiated  
----  
(Alt-Enter mostra dicas)

## Overriding

Usamos a notação *@Override* para indicar que um método está sendo *sobrescrito*.

Esta notação é usada para criar um método igual o método da classe mãe, porém **sobrescrevendo** a sua funcionalidade.

## Overriding

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

```
public abstract class Conta {  
  
    protected float saldo;  
    protected float limite;  
    public int numero;  
    public int agencia;  
  
    public void mostraInfo() {  
        System.out.println("Saldo: R$" + this.saldo);  
        System.out.println("Limite: R$" + this.limite);  
    }  
}
```

## Overriding

```
public class ContaPessoaFisica extends Conta {  
  
    private String nomePessoa;  
    private String cpf;  
  
    @Override  
    public void mostraInfo() {  
        super.mostraInfo();  
        System.out.println("Nome do Titular: " + this.nomePessoa);  
        System.out.println("CPF: " + this.cpf);  
    }  
}
```

## Overriding

```
public class ContaPessoaJuridica extends Conta {  
  
    private String razaoSocial;  
    private float limiteFinanceiro;  
  
    @Override  
    public void mostraInfo() {  
        super.mostraInfo();  
        System.out.println("Razão Social: " + this.razaoSocial);  
        System.out.println("Limite Financeiro: R$" + this.limiteFinanceiro);  
    }  
}
```

## Exercício 8

Utilizando o código do exercício anterior, faça com que todos os carros sejam armazenados em um Array.

Sobrescreva o método *toString()* para mostrar as informações do carro na classe mãe e sobrescreva em cada classe filha.

Torne a classe carro abstrata.

# Fim da primeira parte!

Link dos exercícios no [GitHub](#)!