



UNIVERSIDADE PAULISTA

Professor: Me. Pablo I. Gandulfo

Data: 01/08/2020

Versão: 1.1

# Aplicações de Linguagem de Programação Orientada a Objetos

---

MÓDULO INICIAL

## Ementa

---

- Material disponibilizado na Pasta da Turma no Teams
- Ementa:
  - Aplicações de Linguagem de Programação Orientada a Objetos - J16B.pdf
- Avaliação:
  - Trabalho: 5,0 pontos
  - Prova: 5,0 pontos
- Calendário: (aguardando a divulgação do Calendário 2020)
  - Prova: 15/11 (03/11 a 19/11)
  - Prova: 15/11 (03/11 a 19/11)
  - Prova Substitutiva: 29/11
  - Exame: 06/12

# Conteúdo Programático

---

## INTRODUÇÃO

### Iniciando

- Características Gerais da Linguagem
- Ambiente de Programação JAVA

### Sintaxe e Variáveis

- Tipos Básicos (Primitivos)
- Operadores
- Trabalhando com Variáveis
- Exercícios

### Controle de Fluxo e Iteração

- Controle de Fluxo (if/else - switch/case)
- Iteração - Loops (for - while - do/while)
- Exercícios

### O. O., Modelo, Classes, Instâncias e Referências

- O. O., Modelo, Classes, Instâncias e Referências
- Orientação a Objetos
- 4 Pilares da O. O.
- Exercícios

### Módulo 1 - AWT – Abstract Windowing Toolkit

### Módulo 2 - SWING – Parte 01 – Criação de Objetos – via código

### Módulo 3 - SWING – Parte 02 – Utilização de Objetos Visualmente

### Módulo 4 - SWING – Parte 03 – Tratamento de Eventos

### Módulo 5 - JDBC – java.sql - Conexão com Banco de Dados

### Módulo 6 - Manipulação de dados com linguagem SQL

### Módulo 7 - Design Patterns – DAO (Data Access Object)

### Módulo 8 – Hibernate

### Módulo 9 - Introdução a aplicação Web

### Módulo 10 - MVC – Model View Controller

### Módulo 11 - JSTL - Tratamento de erros

### Módulo 12 – Relatórios

The logo for UNIP (Universidade Paulista) features the letters 'UNIP' in a bold, italicized, yellow font with a black outline.The text 'UNIVERSIDADE PAULISTA' is written in a white, italicized, sans-serif font on a red rectangular background.

Professor: Me. Pablo I. Gandulfo

Data: 01/08/2020

Versão: 1.1

## INTRODUÇÃO - Iniciando

---

- Características Gerais da Linguagem
- Ambiente de Programação JAVA

## Características Gerais da Linguagem

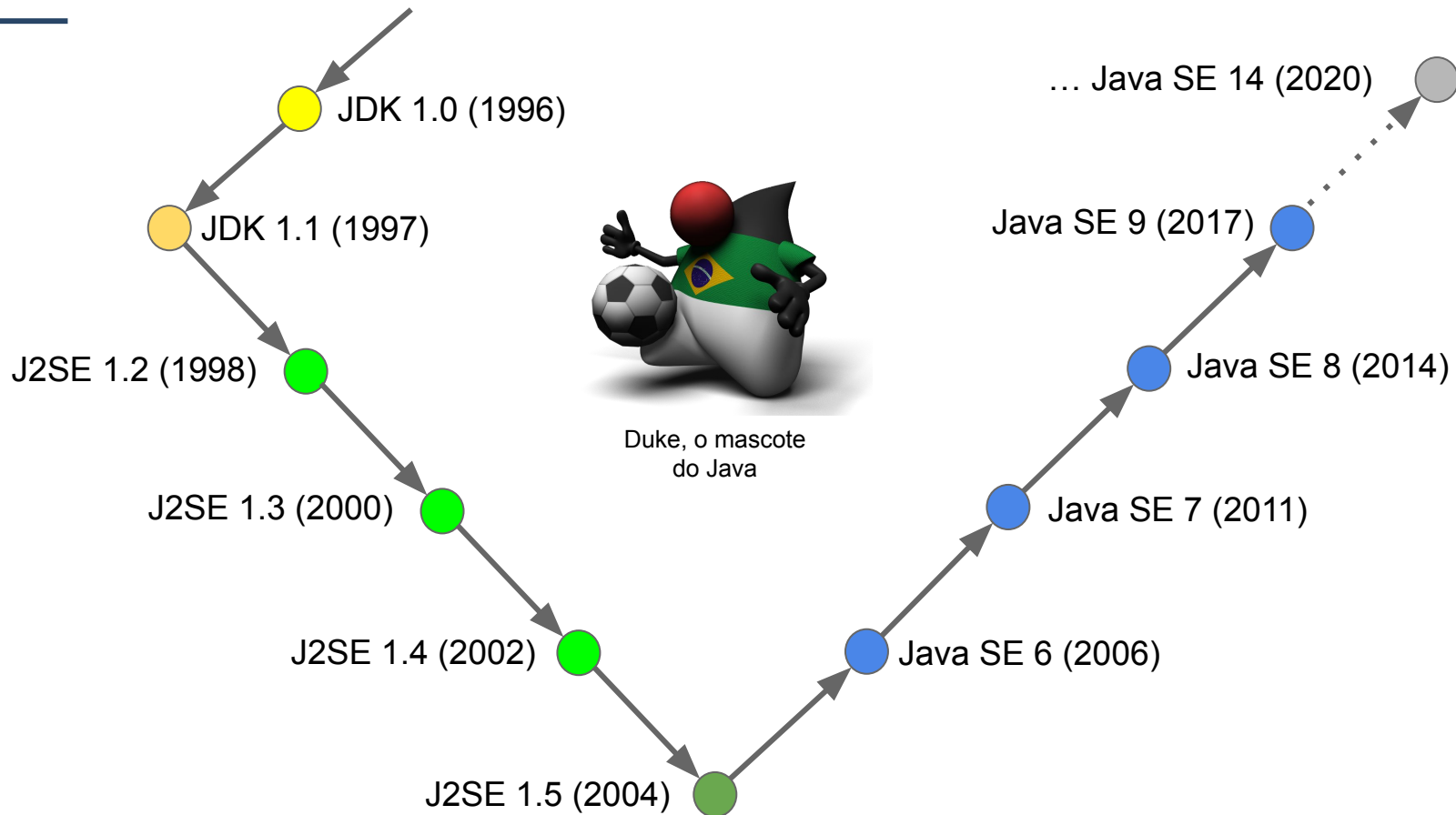
---

- Java é um **padrão aberto!**
  - Suas bibliotecas internas estão disponíveis para visualização
  - Produtos de terceiros são incentivados a utilizar a tecnologia
  - Comunidade para incentivar o seu progresso: Java Community Process (JCP) e Specification Requests (JSRs)
- Java é **gratuito!**
- Além disso, é:

● Simples	● Multiplataforma
● Orientada a Objetos	● Robusta e com Alta Performance
● Interpretada e Compilada	● Segura

## História

James Gosling / Mike Sheridan / Patrick Naughton, entre outros



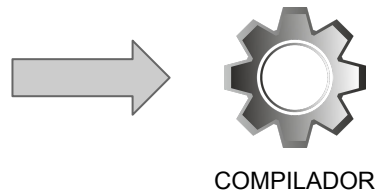
## Conceitos Básicos

- Computador  $\equiv$  [executa]  $\Rightarrow$  Programa  
 $\equiv$  [escrito em]  $\Rightarrow$  Linguagem de Máquina

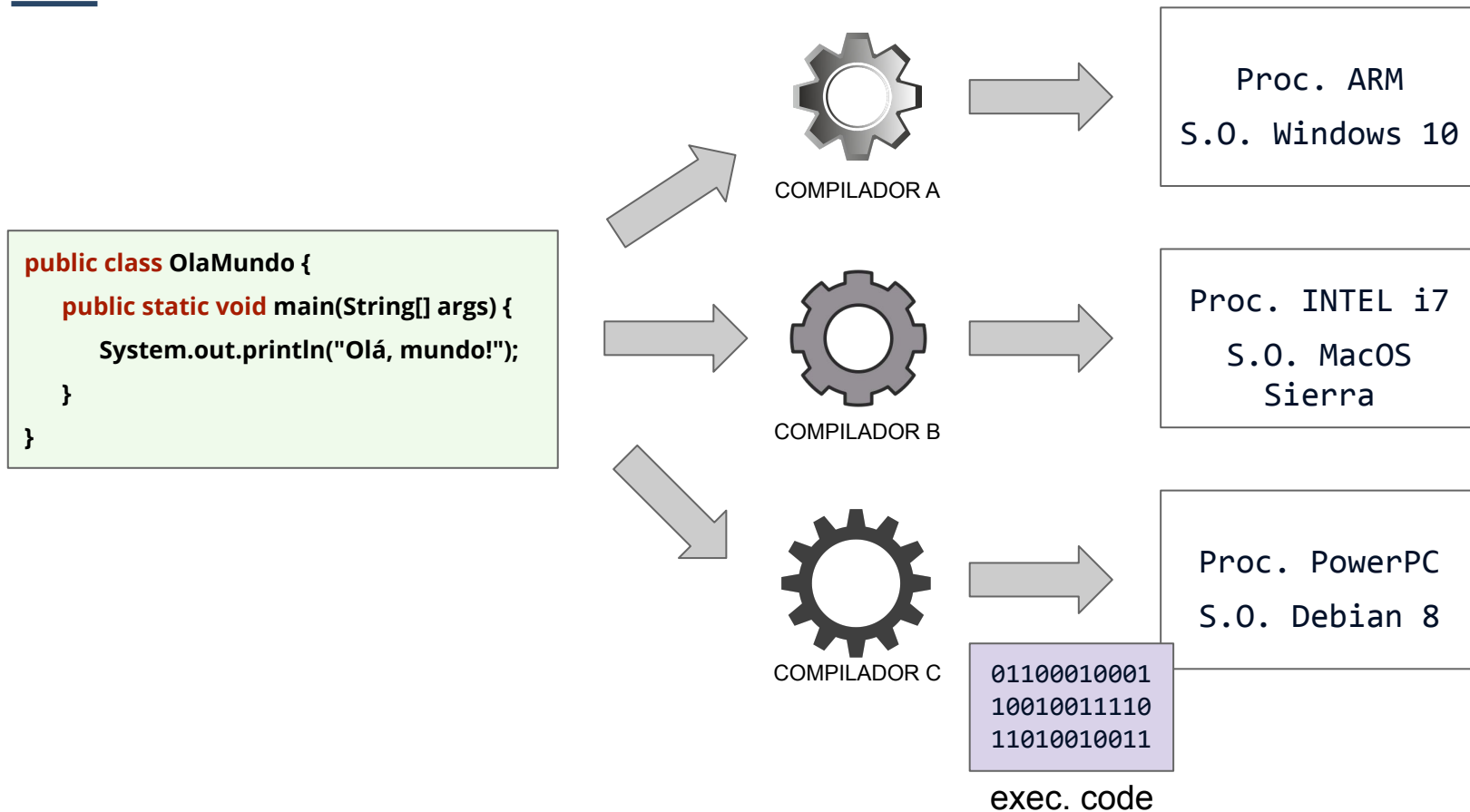
```
011000100010001101111011001010001101000100010001  
100100111101111011010011000010111011000100111010  
110100100111000100000000100100011100010001100001
```

- Desenvolvedor  $\equiv$  [escreve em]  $\Rightarrow$  Linguagem de Programação  
 $\equiv$  [traduz em]  $\Rightarrow$  Linguagem de Máquina

```
public class OlaMundo {  
    public static void main(String[] args) {  
        System.out.println("Olá, mundo!");  
    }  
}
```



## Linguagens de Máquina (Processador) e Bibliotecas (S.O.) são Específicas

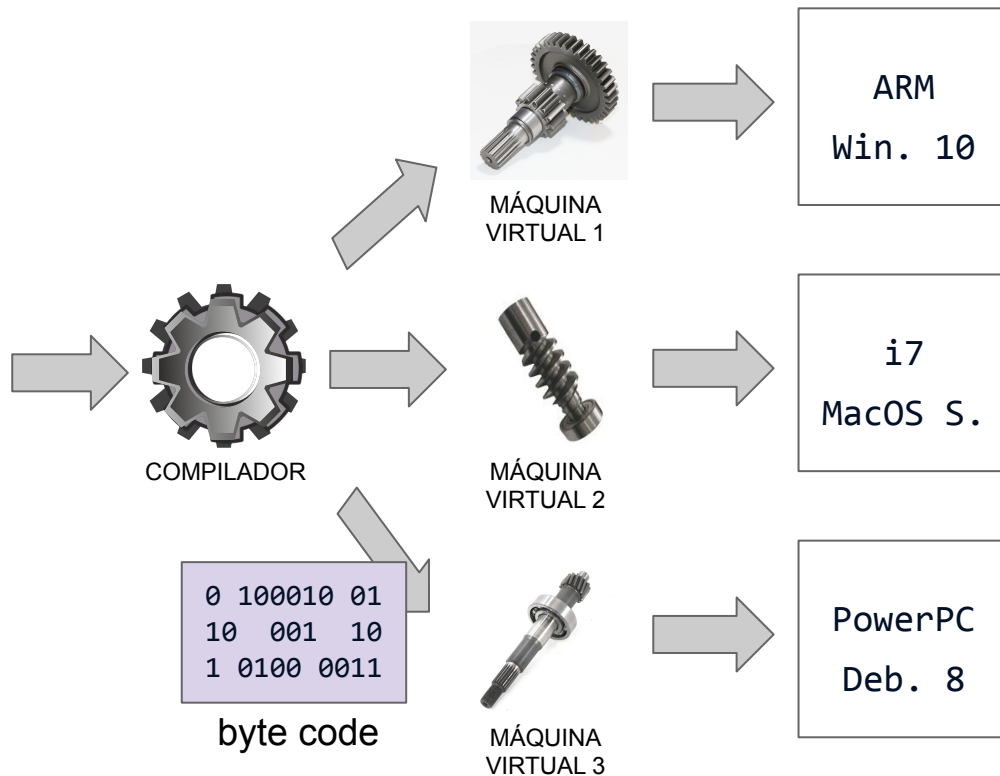




## Máquinas Virtuais (VM) como Possível Solução

```
public class OlaMundo {  
    public static void main(String[] args) {  
        System.out.println("Olá, mundo!");  
    }  
}
```

*Write once, run  
anywhere*



## Máquinas Virtuais: Vantagens vs Desvantagens

---

- Vantagem:
  - Não é necessário compilar o mesmo código várias vezes, para cada arquitetura de processador e sistema operacional
- Desvantagem:
  - Perda de performance, já que o código precisa ser processado pela VM, além do consumo de recursos envolvido
- Vantagem:
  - A VM pode efetuar otimizações em tempo de execução
  - A VM pode compilar partes do código para execução direta em tempo de execução (Just-in-time compilation - JIT)

## Primeiro Programa

---

- Arquivo OlaMundo.java:

```
public class OlaMundo {  
    public static void main(String[] args) {  
        System.out.println("Olá, mundo!");  
    }  
}
```

- Compilar e executar no *prompt* de comando:

```
> dir /b /w  
OlaMundo.java  
> javac OlaMundo.java  
> dir /b /w  
OlaMundo.class  
OlaMundo.java
```

```
> dir /b /w  
OlaMundo.class  
OlaMundo.java  
> java OlaMundo  
Olá, Mundo!
```

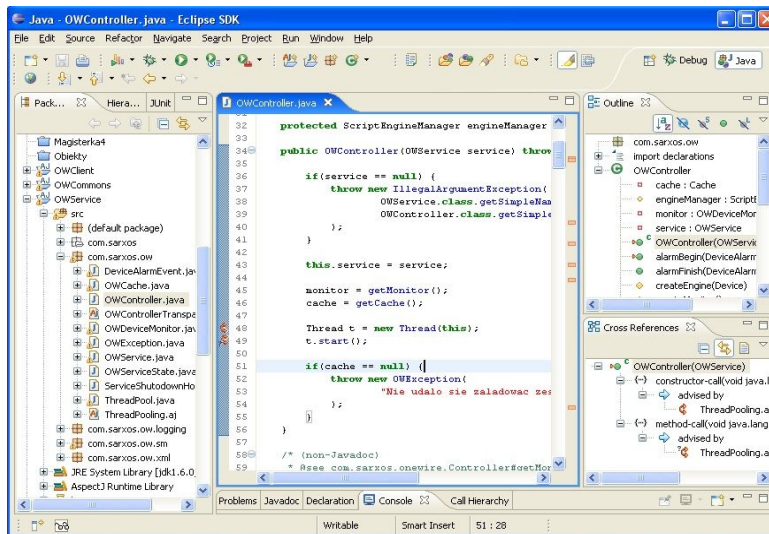
## Ambiente de Programação JAVA – Pacotes de Instalação

---

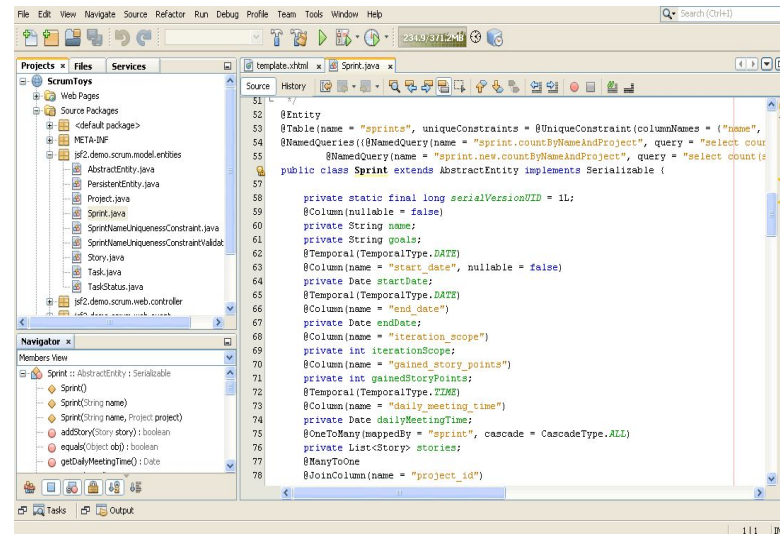
<div>Plataformas</div> <div>Pacotes</div>	<b>Java Santard Edition (SE)</b>	<b>Java Enterprise Edition (EE)</b>	<b>Java Micro Edition (ME)</b>
<b>Java Runtime Environment (JRE)</b>	Uso Desktop no Cliente	Uso Desktop no Servidor	Uso Disp. Móvel
<b>Java Development Kit (JDK)</b>	Desenv. Aplicações Básicas	Desenv. Aplicações Multicamadas, Distribuídas, e ...	Desenv. Aplicações para Disp. Móvel

## Ambiente de Programação JAVA – Ambientes de Desenvolvimento Integrado (IDE's)

Eclipse (IBM => Soft. Livre):



Netbeans (... => SUN => Soft. Livre):



IntelliJ, JDeveloper e outros ...



Professor: Me. Pablo I. Gandulfo

Data: 01/08/2020

Versão: 1.1

## INTRODUÇÃO – Sintaxe e Variáveis

---

- Comentários
- Identificadores
- Tipos Básicos (Primitivos)
- Operadores
- Trabalhando com Variáveis
- Convenções
- Exercícios

## Comentários

---

- De uma linha => `“//”`

```
public static void main(String[] args) {  
    // Imprimindo um texto na tela  
    System.out.println("Olá, mundo!");  
}
```

- De uma ou mais linhas => `“/* ... */”`

```
/**  
 * Método main  
 * Ponto de entrada do programa  
 */  
public static void main(String[] args) {  
    System.out.println("Olá, mundo!");  
}
```

## Identificadores

---

- Identificador é o nome que utilizamos para representar variáveis, classes, objetos, etc. (~ Matemática  $\Rightarrow$  incógnitas  $x, y, \dots$ )
  - não pode ser uma palavra-reservada (palavra-chave)
  - não pode ser **true** e **false**, literais que representam os tipos lógicos (booleanos), e **null**, literal que representa o tipo nulo
  - não pode conter espaços em brancos ou outros caracteres de formatação
- O identificador deve ser único no seu espaço de utilização
- Exemplos:

idade

validade

endereço

quantidadeItens

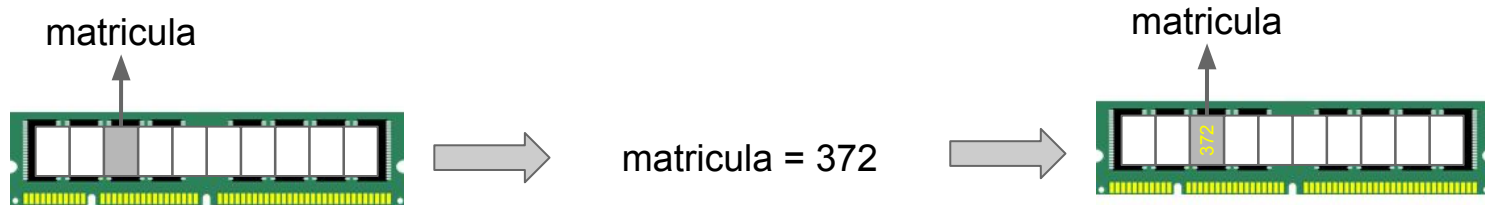
situacaoPedido

preço



## Variáveis

- Um programa basicamente manipula dados, normalmente contidos em variáveis (armazenadas na memória RAM)
- Uma variável pode guardar um dado (ou uma coleção) de diversos tipos: números, textos, lógicos (verdadeiro ou falso) e referências a objetos
- O nome da variável permite referenciá-la para leitura e atribuição de valor



## Declaração e Inicialização de Variáveis

---

- Para utilizar variáveis no código, elas devem ser previamente declaradas (em qualquer linha de um bloco):

```
// Uma variável do tipo inteira (int) de nome codigoMatricula  
int codigoMatricula;  
  
// Uma variável do tipo decimal (double) de nome salario  
double salario;
```

- Toda variável precisa ser inicializada antes de ser utilizada, através do operador de atribuição '='

```
// Inicialização  
codigoMatricula = 10;  
  
// Uso correto  
System.out.println(codigoMatricula);  
  
// Erro de compilação  
System.out.println(salario);
```

## Tipos Básicos (primitivos)

Tipo	Descrição	Tamanho
<b>byte</b>	Valor inteiro entre -128 e 127 (inclusive)	1 byte
<b>short</b>	Valor inteiro entre -32.768 e 32.767 (inclusive)	2 bytes
<b>int</b>	Valor inteiro entre -2.147.483.648 e 2.147.483.647 (inclusive)	4 bytes
<b>long</b>	Valor inteiro entre -9.223.372.036.854.775.808 e 9.223.372.036.854.775.807 (inclusive)	8 bytes
<b>float</b>	Valor com ponto flutuante entre $1,40129846432481707 \times 10^{-45}$ e $3,40282346638528860 \times 10^{38}$ (positivo ou negativo)	4 bytes
<b>double</b>	Valor com ponto flutuante entre $4,94065645841246544 \times 10^{-324}$ e $1,79769313486231570 \times 10^{308}$ (positivo ou negativo)	8 bytes
<b>boolean</b>	true ou false	1 bit
<b>char</b>	Um único caractere Unicode de 16 bits. Valor inteiro e positivo entre 0 (ou '\u0000') e 65.535 (ou '\uffff').	2 bytes

## Tipos Básicos (primitivos) - cont.

---

- Obs.: não existe tipo básico que armazene textos. O tipo **char** apenas armazena um caracter. Para armazenar um texto é necessário utilizar o tipo **String**, que é um tipo complexo.

## Operadores Aritméticos

---

- Soma +
- Subtração -
- Multiplicação \*
- Divisão /
- Módulo (resto da divisão inteira) %

**Obs.:** a precedência de operadores no JAVA segue as mesmas regras da matemática

```
int doisMaisTres = 2 + 3;           // doisMaisTres = 5
int cincoMenosDois = 5 - 2;         // cincoMenosDois = 3
int quatroVezesOito = 4 * 8;        // quatroVezesOito = 32
int dozeDivididoPorQuatro = 12 / 4; // dozeDivididoPorQuatro = 3
int oitoModuloTres = 8 % 3;          // oitoModuloTres = 2
```

## Operadores de Atribuição

---

- Simples =
- Incremental +=
- Decremental -=
- Multiplicativa \*=
- Divisória /=
- Modular %=

```
int valor = 3;      // valor = 3
valor += 5;         // valor = 8
valor -= 6;         // valor = 2
valor *= 5;         // valor = 10
valor /= 2;         // valor = 5
valor %= 3;         // valor = 2
```

## Operadores Unários

---

- Sinal positivo +
- Sinal negativo -
- Pré e pós-incremento ++
- Pré e pós-decremento --
- Negação !

**Obs.:** quando os operadores ++ e -- estão à esquerda, eles aplicam o incremento antes de executar a linha de código. Caso contrário, aplicam depois de executar a linha de código.

```
int valorPositivo = +15;    // valorPositivo = 15
int valorNegativo = -317;   // valorNegativo = -317
int valor = 0;
valor++;                    // valor = 1
int resultado;
resultado = ++valor;        // resultado = valor = 2
valor--;                    // valor = 1
boolean b = !(valor == 1);  // b = false
```

## Operadores Relacionais

---

- Igualdade ==
- Diferença !=
- Menor <
- Menor ou igual <=
- Maior >
- Maior ou igual >=

```
int valor = 3;  
boolean b = false;  
  
b = (valor == 3);    // b = true  
b = (valor != 3);    // b = false  
b = (valor < 3);     // b = false  
b = (valor <= 3);    // b = true  
b = (valor > 2);     // b = true  
b = (valor >= 4);    // b = false
```



## Operadores Lógicos

---

- "E" lógico &&
- "OU" lógico ||

```
int valor = 3;  
boolean b = false;  
b = (valor < 12) && (valor > 2);      // b = true  
b = (valor < 15) && (valor != 3);    // b = false  
b = (valor <= 3) || (valor > 20);    // b = true  
b = (valor < 2) || (valor > 8);      // b = false
```

## Trabalhando com Variáveis

---

```
public class ExemploVariaveis {  
    public static void main(String[] args) {  
        int idade = 30 + 2;  
        System.out.println("Idade: " + idade);  
  
        char letraL = 'l';  
        System.out.println("Letra L: " + letraL);  
  
        boolean eCrianca = (idade < 12);  
        System.out.println("É criança: " + eCrianca);  
  
        idade += 1;  
        System.out.println("Agora a idade é: " + idade + ".");  
    }  
}
```

## Casting e Promoção

---

- Um valor de maior grandeza (ex.: double) não pode ser diretamente atribuído à uma variável de menor grandeza (ex.: int). Para tanto, é necessário informar explicitamente que se deseja moldar (~ arredondar) o valor, processo de nome **casting**.
- Um valor de menor grandeza é implicitamente promovido para uma grandeza maior, quando assim for esperado. Esse processo tem o nome de **promoção**.

```
double pi = 3.14;  
int i = (int) pi;           // casting do valor de pi para atribuir em i, ficando i = 3  
  
float f = i;                // promoção do valor inteiro de i para float, ficando f = 3
```

## Conversões de Tipos Básicos

PARA:\nDE:	byte	short	char	int	long	float	double
byte	----	implícito	(char)	implícito	implícito	implícito	implícito
short	(byte)	----	(char)	implícito	implícito	implícito	implícito
char	(byte)	(short)	----	implícito	implícito	implícito	implícito
int	(byte)	(short)	(char)	----	implícito	implícito	implícito
long	(byte)	(short)	(char)	(int)	----	implícito	implícito
float	(byte)	(short)	(char)	(int)	(long)	----	implícito
double	(byte)	(short)	(char)	(int)	(long)	(float)	----

## Convenções

---

- Na convenção de nomes no Java, os nomes de variáveis devem seguir o padrão denominado ***lower camel case***, com palavras compostas unidas sem espaço e cada palavra iniciando com maiúscula, com exceção da primeira palavra, que deverá ser iniciar com minúscula
- Exemplos:
  - codigoMatricula
  - nomeDoFuncionario
  - listaDeAprovados

## Exercício 1 – Cálculo do Balanço Trimestral

---

- Numa empresa há tabelas com o quanto foi gasto em cada mês. Para fechar o balanço do segundo trimestre, precisamos somar o gasto total. Sabendo que, em Abril, foram gastos 5.000 reais, em Maio, 28.000 reais e em Junho, 11.000 reais, faça um programa que calcule e imprima o gasto total no trimestre. Siga esses passos:
  - Crie uma classe chamada BalancoTrimestral com um bloco main, como nos exemplos anteriores
  - Dentro do main declare uma variável inteira chamada gastosAbril e inicialize-a com 5.000
  - Crie também as variáveis gastosMaio e gastosJunho, inicializando-as com 28.000 e 11.000, respectivamente. Utilize uma linha para cada declaração.
  - Crie uma variável chamada gastosTrimestre e inicialize-a com a soma das outras 3 variáveis:  

```
int gastosTrimestre = gastosAbril + gastosMaio + gastosJunho;
```
  - Imprima a variável gastosTrimestre

## Exercício 2 - Utilização de Operadores

---

- Adicione código (sem alterar as linhas que já existem) no programa a seguir para imprimir o resultado:

Resultado: 12, 12.3, true, w

```
public class ExercicioSimples {  
    public static void main(String[] args) {  
        int i = 10;  
        double d = 2;  
        boolean b = false;  
        char c = 'p';  
  
        // imprime concatenando diversas variáveis  
        System.out.println("Resultado: " + i + ", " + d + ", " + b + ", " + c);  
    }  
}
```

## Resposta do Exercício 1

---

```
public class BalancoTrimestral {  
    public static void main(String[] args) {  
        short gastosAbril = 5000;  
        short gastosMaio = 28000;  
        short gastosJunho = 11000;  
        int gastosTrimestre = gastosAbril + gastosMaio + gastosJunho;  
  
        System.out.println("Os gastos totais do 1º trimestre foram R$ " + gastosTrimestre);  
    }  
}
```



## Resposta do Exercício 2

---

```
public class ExercicioSimples {  
    public static void main(String[] args) {  
        int i = 10;  
        double d = 2;  
        boolean b = false;  
        char c = 'p';  
  
        i += (int) d;  
        d = i + 0.3;  
        b = !b;  
        c += 7;  
  
        // imprime concatenando diversas variáveis  
        System.out.println("Resultado: " + i + ", " + d + ", " + b + ", " + c);  
    }  
}
```



UNIVERSIDADE PAULISTA

Professor: Me. Pablo I. Gandulfo

Data: 01/08/2020

Versão: 1.1

## INTRODUÇÃO - Controle de Fluxo e Iteração

---

- Controle de Fluxo (*if/else* - *switch/case*)
- Iteração - *Loops* (*for* - *while* - *do/while*)
- Exercícios

## Controle de Fluxo – *if/else*

---

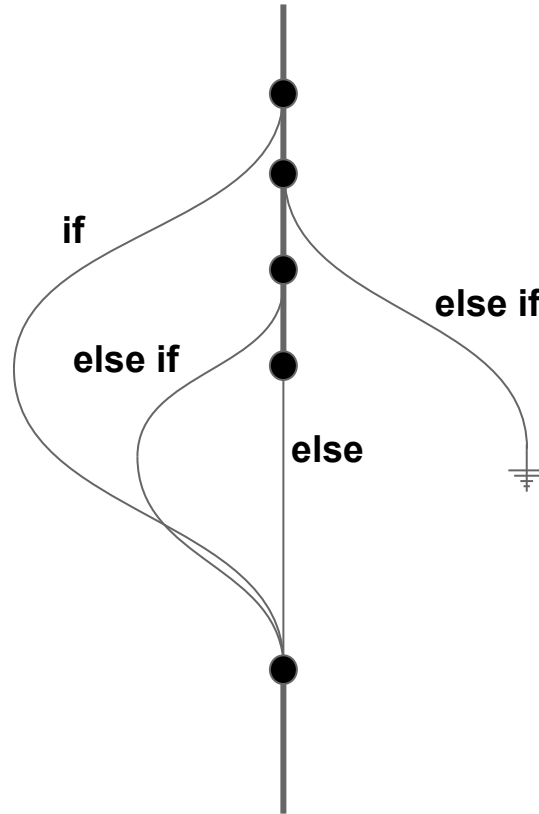
- O comportamento de uma aplicação pode ser influenciado pelos valores de cálculo. Por exemplo, ao pagar um boleto, se a data de vencimento for menor que a atual, deve ser aplicada uma multa. Ou se o saldo da conta for insuficiente, o pagamento deverá ser interrompido.

```
if (valorBoleto == 0) { /* nada a fazer */ }  
else if (saldoConta < valorBoleto) {  
    System.out.println("Saldo insuficiente para efetuar este pagamento");  
    return;  
}  
else if (saldoConta == valorBoleto) { if (confirmaPagamento()) { pagarBoleto(); } }  
else { pagarBoleto(); }
```

- O comando *if* pode opcionalmente ser seguido de um ou mais comandos *else if* e um comando *else* ao final
- Cada comando *if* e *else if* possuirá uma expressão de teste que retorna um valor booleano. O primeiro comando que for testado na ordem e retornar um valor verdadeiro terá o bloco correspondente executado. Caso nenhum retorne, o bloco correspondente da instrução *else* será executado.

## Controle de Fluxo - *if/else*

---



## Controle de Fluxo – *switch/case*

---

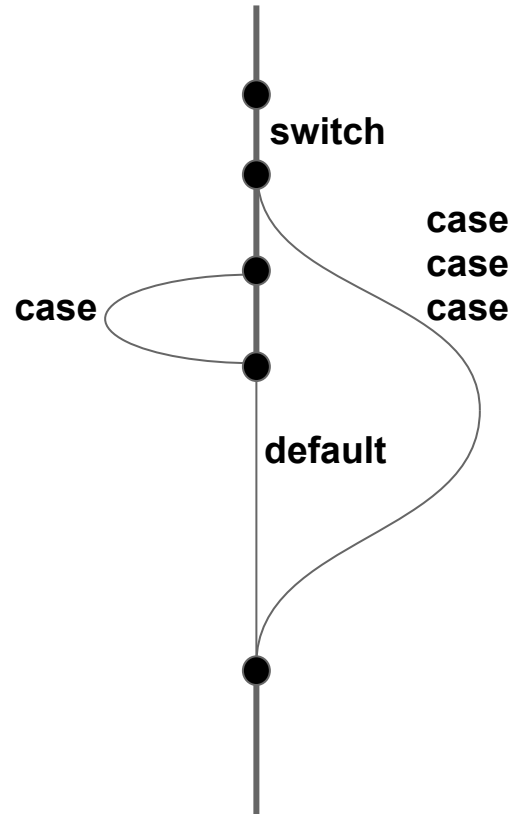
- Em alguns casos, é necessário fazer vários testes de valores para uma mesma variável e, a partir de uma correspondência, executar a ação apropriada. Por exemplo, a classificação de dia útil ou não depende apenas do dia da semana.

```
switch (diaDaSemana) {  
    case 2: case 3: case 4:  
    case 5: case 6:  
        System.out.println("Dia útil");  
        break;  
    case 1: case 7:  
        System.out.println("Fim de semana");  
    default:  
        System.out.println("Não é dia útil"); break;  
}
```

- Até a versão 6 do Java, a variável analisada no *switch* só poderia ser do tipo **byte**, **short**, **char**, **int**, constantes **enum** ou classes wrapper **Byte**, **Short**, **Character** e **Integer**. E a partir da versão 7, a variável também pode ser do tipo **String**.

## Controle de Fluxo – *switch/case*

---



## Controle de Fluxo – *for*

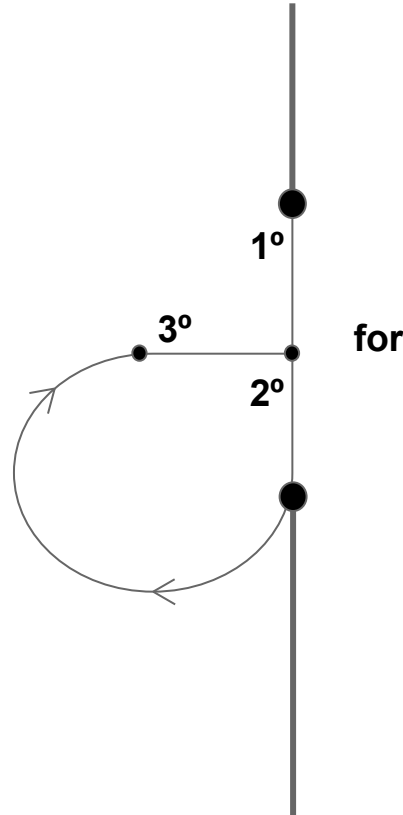
---

- Em alguns casos, a aplicação necessita realizar a mesma operação mais de uma vez. Por exemplo, para calcular juros compostos (juros sobre juros), é necessário calcular o novo valor de cobrança a partir do anterior, sucessivamente.

```
valorBoleto = 200; JurosAoMes = 2; mesesDeAtraso = 3;  
for (int i = 1; i <= mesesDeAtraso; i++) {  
    valorBoleto += valorBoleto * JurosAoMes / 100; // adicionando 2% do valor atualizado do boleto  
}
```

- O comando *for* recebe três parâmetros:
  - O primeiro é executado em primeiro lugar, sendo normalmente utilizado para inicializar a variável de controle
  - O segundo deve possuir uma expressão booleana que é testada para saber se o bloco deverá ser executado (mesmo que duas ou mais vezes) ou não
  - O terceiro é executado ao fim de cada execução do bloco, sendo normalmente utilizado para incrementar a variável de controle
- Usualmente o *for* é utilizado para repetir um conjunto de operações um número previamente conhecido de vezes

## Controle de Fluxo - *for*





## Controle de Fluxo – *while*

---

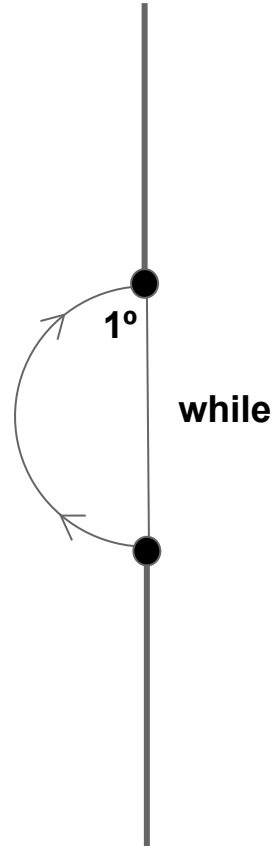
- Pode ser que a aplicação necessite realizar a mesma operação até que uma determinada situação aconteça. Por exemplo, um programa pessoal de e-mails, como o MS Outlook, permite recuperar e-mails novos, finalizando ao chegar no mais recente.

```
while (existemNovosEmails()) {  
    emails = recuperarNovosEmails();  
    armazenarEmails(emails);  
}
```

- O comando *while* recebe um parâmetro, uma expressão booleana que é testada para saber se o bloco deverá ser executado (mesmo que duas ou mais vezes) ou não
- Usualmente o *while* é utilizado para repetir um conjunto de operações enquanto um evento externo ao programa ainda não tiver acontecido

## Controle de Fluxo - *while*

---



## Controle de Fluxo – *do/while*

---

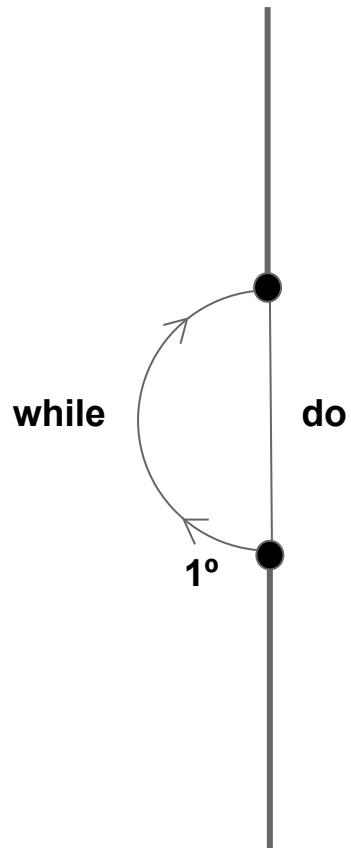
- Similar aos casos anteriores, pode ser necessário repetir a mesma operação ao menos uma vez. Por exemplo, ao recuperar e-mails, mas já partindo do entendimento que existem e-mails novos na caixa de correio.

```
do {  
    emails = recuperarNovosEmails();  
    armazenarEmails(emails);  
} while (existemNovosEmails());
```

- O comando *do/while* recebe um parâmetro similar ao *while*
- Usualmente o *do/while* é utilizado nos mesmos casos que o *while*, considerando que o bloco será executado no mínimo uma vez

## Controle de Fluxo - *do/while*

---



## Exercício 1 – Imprimir os Primeiros 50 Números Inteiros Positivos

---

- Crie uma classe chamada `ImprimeInteiros`
- Crie um método `main`
- Escreva o código necessário para imprimir os números de 1 a 50
  - Escolha o comando de controle de fluxo mais conveniente para este caso em particular:
    - *if/else*
    - *switch/case*
    - *for*
    - *while*
    - *do/while*

## Exercício 2 - Imprimir de Acordo com Números Pares ou Ímpares

---

- Crie uma classe chamada `ImprimeParesImpares`
- Crie um método `main`
- Escreva o código necessário para percorrer os números de 1 a 20 e, quando o número for par, imprimir "PAR", e quando for ímpar, imprimir "ÍMPAR"

### Exercício 3 – Calcular o Próximo Número Primo

---

- Crie uma classe chamada CalculaProximoNumeroPrimo
- Crie um método main
- Escreva o código necessário para encontrar o número primo que vem após o número 113
- **Obs.:** para que um número seja primo, deve atender às seguintes regras:
  - O número deve ser inteiro e maior que 1
  - Só pode ser divisível (resto 0) por ele mesmo e por 1

## Exercício 4 - Imprimir o Nome de Todos os Meses (Janeiro, ..., Dezembro)

---

- Crie uma classe chamada `ImprimeNomeMeses`
- Crie um método `main`
- Escreva o código necessário para imprimir, por extenso, todos os meses do ano
  - Janeiro
  - Fevereiro
  - ...
  - Novembro
  - Dezembro



## Exercício 5 - Imprimir a sequência de Fibonacci

---

- Crie uma classe chamada Fibonacci
- Crie um método main
- Escreva o código necessário para imprimir a sequência de Fibonacci:
  - 0 1 1 2 3 5 8 13 21 34 55 ...
  - Ou seja, começando em 0 e depois 1, o próximo número sempre corresponderá à soma dos dois números anteriores
  - DESAFIO: fazer o mesmo algoritmo, mas utilizando apenas duas variáveis

## Resposta do Exercício 1

---

```
public class ImprimeInteiros {  
    public static void main(String[] args) {  
        int contador;  
  
        for (contador = 1; contador <= 50; contador++) {  
            System.out.println(contador);  
        }  
    }  
}
```

## Resposta do Exercício 2

---

```
public class ImprimeParesImpares {  
    public static void main(String[] args) {  
        int contador;  
  
        for (contador = 1; contador <= 20; contador++) {  
            if ((contador % 2) == 0) {  
                System.out.println("PAR");  
            }  
            else {  
                System.out.println("ÍMPAR");  
            }  
        }  
    }  
}
```

## Resposta do Exercício 3

---

```
public class CalculaProximoNumeroPrimo {  
    public static void main(String[] args) {  
        boolean isPrimo;  
        int numeroAtual = 113;  
  
        System.out.println("Procurando próximo número primo a partir de " + numeroAtual);  
        do {  
            isPrimo = true; numeroAtual++;  
            for (int i = 2; i <= numeroAtual - 1; i++) {  
                if (numeroAtual % i == 0) {  
                    isPrimo = false; break;  
                }  
            }  
        } while (!isPrimo);  
        System.out.println("Encontrado ==> " + numeroAtual);  
    }  
}
```

## Resposta do Exercício 4

---

```
public class ImprimeNomeMeses {  
    public static void main(String[] args) {  
        for (int mes = 1; mes <= 12; mes++) {  
            switch (mes) {  
                case 1: System.out.println("Janeiro"); break;  
                case 2: System.out.println("Fevereiro"); break;  
                case 3: System.out.println("Março"); break;  
                case 4: System.out.println("Abril"); break;  
                case 5: System.out.println("Maio"); break;  
                ...  
                case 11: System.out.println("Novembro"); break;  
                case 12: System.out.println("Dezembro"); break;  
                default: System.out.println("---"); break;  
            }  
        }  
    }  
}
```

## Resposta do Exercício 5

---

```
public class Fibonacci {  
    public static void main(String[] args) {  
        int num, proxNum, temp;  
  
        num = 0;  
        System.out.print(num);  
        proxNum = 1;  
        System.out.print(" " + proxNum);  
        for (int i = 3; i <= 20; i++) {  
            temp = proxNum;  
            proxNum = num + proxNum;  
            num = temp;  
  
            System.out.print(" " + proxNum);  
        }  
    }  
}
```

## Resposta do DESAFIO do Exercício 5

---

```
public class Fibonacci {  
    public static void main(String[] args) {  
        int num, proxNum;  
  
        num = 0;  
        System.out.print(num);  
        proxNum = 1;  
        System.out.print(" " + proxNum);  
        for (int i = 3; i <= 20; i++) {  
            proxNum = num + proxNum;  
            num = proxNum - num;  
  
            System.out.print(" " + proxNum);  
        }  
    }  
}
```



**UNIVERSIDADE PAULISTA**

Professor: Me. Pablo I. Gandulfo

Data: 01/08/2020

Versão: 1.1


## INTRODUÇÃO - O. O., Modelo, Classes, Instâncias e Referências

---

- Orientação a Objetos
- Classe (Estrutura de Dados Complexa)
- Pacote
- 4 Pilares da O. O.
- Abstração
- Encapsulamento
- Herança
- Polimorfismo
- Exercícios



# Programação Orientação a Objetos - Uma Nova Forma de Pensar e Programar

- Uma aplicação pode ser desenvolvida para automatizar as atividades que determinado através dos atributos e envolvidos
  - Anteriormente modelados e dados e
- 
- ou seja: quais dados estão envolvidos? De onde vem? Para onde vão? E quais cálculos ou processamentos eles sofrem?



## Programação Orientação a Objetos – Uma Nova Forma de Pensar e Programar

- Mais recentemente, o paradigma da orientação a objetos (e uma nova geração de linguagens de programação) permitiu que a modelagem e codificação das aplicações partisse de representações mais próximas do mundo real: os objetos



CLASSE



OBJETOS

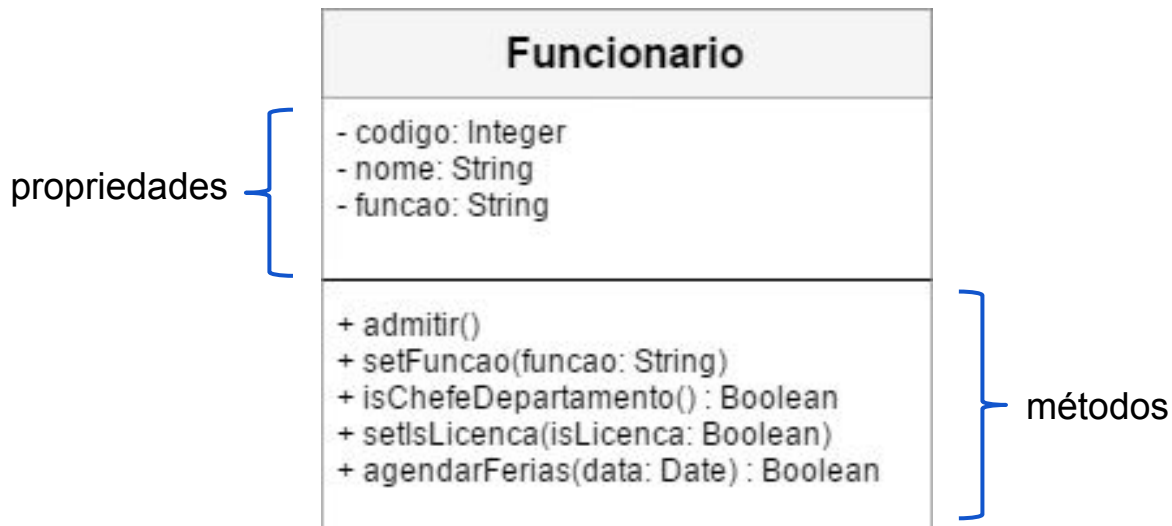
## Programação Orientação a Objetos – Entidade, Objeto e Classe

---

- Em programação, um objeto permite representar uma entidade, seja ela física (ex.: caminhão), conceitual (ex.: processo químico) ou de software (ex.: lista encadeada), tanto quanto seu estado (atributos e relações) e comportamento (operações ou métodos)
- Cada objeto é definido por uma classe que o representa. Uma classe pode ser vista como um projeto de um avião, que pode servir para construir vários aviões, inclusive com características diferentes: número de identificação, cor, data de fabricação, custo envolvido, etc..
- Cada objeto possui uma identidade única

## Classe (Estrutura de Dados Complexa)

- Uma classe é uma descrição de um conjunto de objetos que compartilham as mesmas propriedades (ou atributos), métodos (ou operações), relações e semântica
- Representação na UML:



## Propriedades de Classe

---

- A propriedade de uma classe representa uma característica que pode estar associado à própria classe, ou ao objeto instanciado
- Uma propriedade possui:
  - Modificadores:
    - De Acesso: *private* (classe), *protected* (pacote e herança), *<default>* (pacote) e *public* (todos)
    - De Instância ou Não: *static* (classe) e *<default>* (instância)
    - Variável ou Não: *final* (constante) e *<default>* (variável)
  - Nome
  - Tipo
  - Valor Inicial
- Exemplo ...

## Métodos de Classe

---

- O método de uma classe representa um comportamento ou serviço associado à própria classe ou ao objeto instanciado que pode ser executado
- Um método possui:
  - Modificadores:
    - De Acesso: *private* (classe), *protected* (pacote e herança), *<default>* (pacote) e *public* (todos)
    - De Instância ou Não: *static* (classe) e *<default>* (instância)
    - Abstrato ou Não: *abstract* (abstrato) e *<default>* (concreto)
    - Pode Ser Sobrescrito ou Não: *final* (não pode) e *<default>* (pode)
  - Valor de Retorno: *void* (nenhum) ou o *<tipo>*
  - Nome
  - Parâmetros (opcional)
- Exemplo ...

## Pacote

---

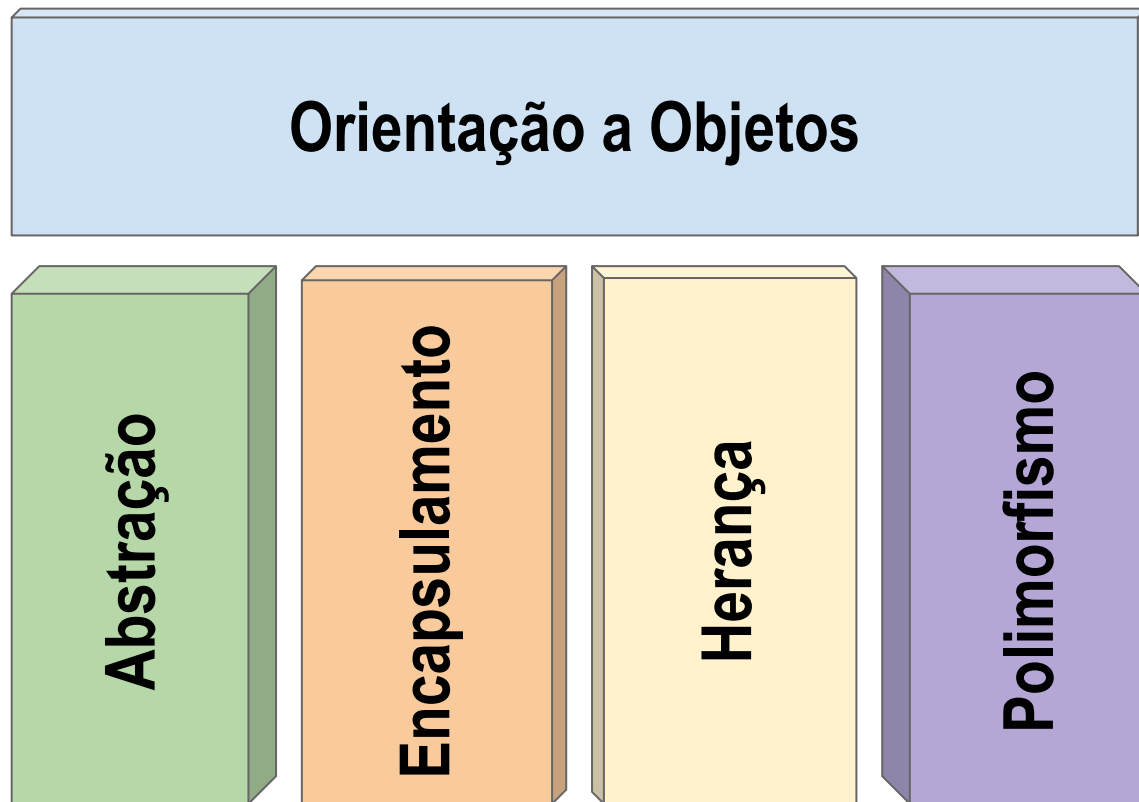
- Um pacote é um mecanismo de propósito geral para organizar elementos, dentre eles, classes, em grupos
- É um elemento de programação que pode conter outros elementos



- A referência à uma classe dentro de um pacote segue o padrão [nome do pacote].[nome do pacote filho].....[nome da classe]. Além disso, existe uma convenção de utilizar como prefixo dos pacotes o domínio de internet da corporação, na ordem inversa.  
Ex.: **org.hibernate**.connection.C3P0ConnectionProvider

## 4 Pilares da O. O.

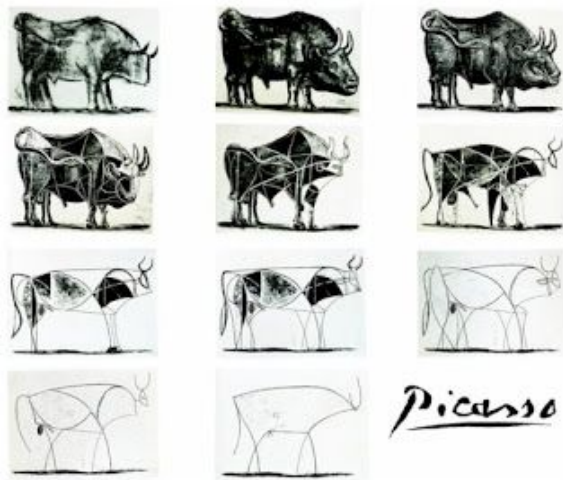
---





## 1º Pilar - Abstração

- É o princípio de ignorar aspectos não relevantes da entidade para concentrar nos principais, ao desenhar e implementar as classes
- É uma forma de lidar com a complexidade das entidades do mundo real, procurando torná-las mais simples



Pablo Picasso, Bull (plates I - XI) 1945

## 2º Pilar – Encapsulamento de Dados

---

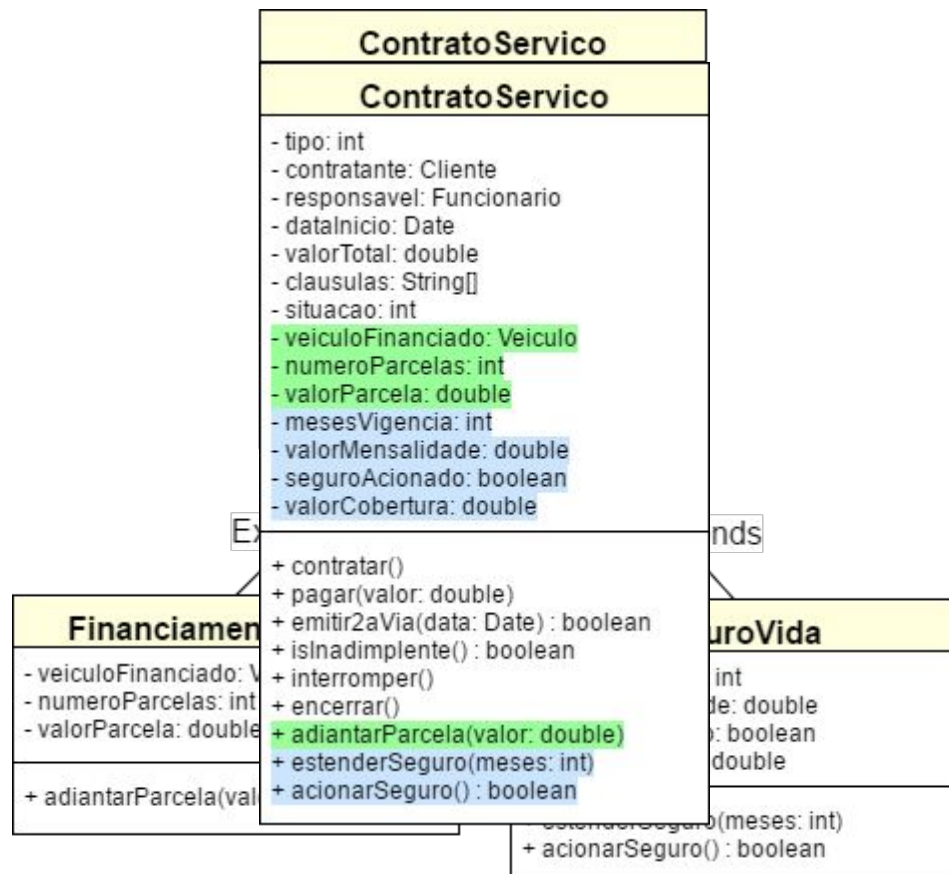
- O encapsulamento de dados ocorre nas propriedades da classe, que são protegidas do acesso externo (à classe), ao definí-las com escopo privado (ou pelo menos mais restrito que público), e criar métodos públicos de leitura (*getters*) e escrita (*setters*)
- O objetivo do encapsulamento é garantir que os dados só possam ser lidos ou escritos de uma forma controlada, inclusive permitindo implementar regras de negócio associadas a esses acessos
- Esse controle permite:
  - Proteger as propriedades de valores inválidos ou indesejados
  - Limitar o acesso a propriedades para leitura ou escrita apenas
  - Implementar regras de negócio relacionadas com a leitura ou escrita das propriedades
- Exemplo

### 3º Pilar – Herança

---

- Diferentes entidades podem estar intimamente relacionadas, sugerindo uma proximidade através de um ancestral comum
- Por exemplo, um banco pode oferecer serviços de Financiamento de Veículos e Seguro de Vida, com ambos compartilhando alguns atributos e comportamentos em comum:
  - Atributos: contratante, funcionário responsável, data de início, valor total do contrato, cláusulas do contrato, situação
  - Comportamentos: contratar, pagar parcela/mensalidade, emitir 2ª via, consultar inadimplência, interromper, encerrar
- Por outro lado, cada serviço possui atributos e comportamentos específicos

### 3º Pilar – Herança (cont.)



### 3º Pilar – Herança em Detalhes

---

- Representa uma relação do tipo “<objeto> **é um tipo de** <classe ancestral>”
- Quanto mais profunda a classe estiver na hierarquia, mais especializada será (generalização / especialização)
- A classe filha herda todas as características (propriedades / atributos) e comportamentos (métodos / operações) da classe ancestral, mesmo que sejam privados (apesar de que não poderá vê-los), além de poder estender a ancestral acrescentando características ou comportamentos novos, e/ou podendo sobrescrever comportamentos (não estáticos) herdados
- O cliente do objeto filho pode adotar uma visão parcial desse objeto, através da classe ancestral, para acionar apenas as características e comportamentos herdados (parecido com interfaces)

## Herança - Exemplo

---

```
public class ContratoServico {  
    protected String dataInicio;  
    protected double valorTotal;  
  
    public void contratar() {  
        dataInicio = new java.util.Date().toString();  
    }  
}
```

## Herança - Exemplo (cont.)

---

```
public class FinanciamentoVeiculo extends ContratoServico {  
    private int numeroParcelas;  
    private double valorParcela;  
  
    public FinanciamentoVeiculo(int numeroParcelas, double valorParcela) {  
        this.numeroParcelas = numeroParcelas;  
        this.valorParcela = valorParcela;  
    }  
  
    public void contratar() {  
        super.contratar();  
        valorTotal = numeroParcelas * valorParcela;  
    }  
}
```

## Herança - Exemplo (cont.)

---

```
public class SeguroVida extends ContratoServico {  
    private int mesesVigencia;  
    private double valorMensalidade;  
  
    public SeguroVida(int mesesVigencia, double valorMensalidade) {  
        this.mesesVigencia = mesesVigencia;  
        this.valorMensalidade = valorMensalidade;  
    }  
  
    public void contratar() {  
        super.contratar();  
        valorTotal = mesesVigencia * valorMensalidade;  
    }  
}
```



## Herança - Exemplo (cont.)

---

```
public class ContratacaoExemplo {  
    public static void main(String[] args) {  
        FinanciamentoVeiculo fv = new FinanciamentoVeiculo(20, 1500);  
        SeguroVida sv = new SeguroVida(36, 800);  
  
        contratarServico(fv);  
        contratarServico(sv);  
    }  
  
    public static void contratarServico(ContratoServico cs) {  
        cs.contratar();  
    }  
}
```

## Interface

- Entre os fabricantes de televisões, os protocolos de conversação utilizados entre o controle remoto e a tv costumam divergir:



## Interface

---

- Mas para um mesmo fabricante, não é incomum existirem controles remotos universais que funcionam com vários modelos diferentes de televisão:



## Interface

---

- Para que isso seja possível, todas as televisões do fabricante devem aderir à uma determinada especificação ou padrão de comunicação
- Mesmo que alguns modelos possuam funcionalidades específicas, as funcionalidades em comum são projetadas para responder da mesma forma
- A idéia de interface em O. O. é muito parecida: é uma forma de definir um padrão, ou contrato, de comportamento (métodos ou operações) que as classes concordam em implementar
- Para que um cliente possa acionar esse comportamento comum, ele faz uso da interface. Ou seja, ao invés de lidar com o objeto diretamente, o cliente adota uma visão parcial desse objeto, através da interface, para acionar apenas esse comportamento comum.

## Interface – Exemplo

---

```
public interface ControleUniversal {  
    public void ligar();  
    public void desligar();  
}
```

```
public class TvSamsungModelo4K implements ControleUniversal {  
    public final int ESTADO_DESLIGADO = 0;  
    public final int ESTADO_LIGADO = 1;  
    private int estado = ESTADO_DESLIGADO;  
    public void ligar() { estado = ESTADO_LIGADO; }  
    public void desligar() { estado = ESTADO_DESLIGADO; }  
}
```

```
public class TvSamsungModelo3D implements ControleUniversal {  
    private boolean ligado = false;  
    public void ligar() { ligado = true; }  
    public void desligar() { ligado = false; }  
}
```

## Interface – Exemplo (cont.)

---

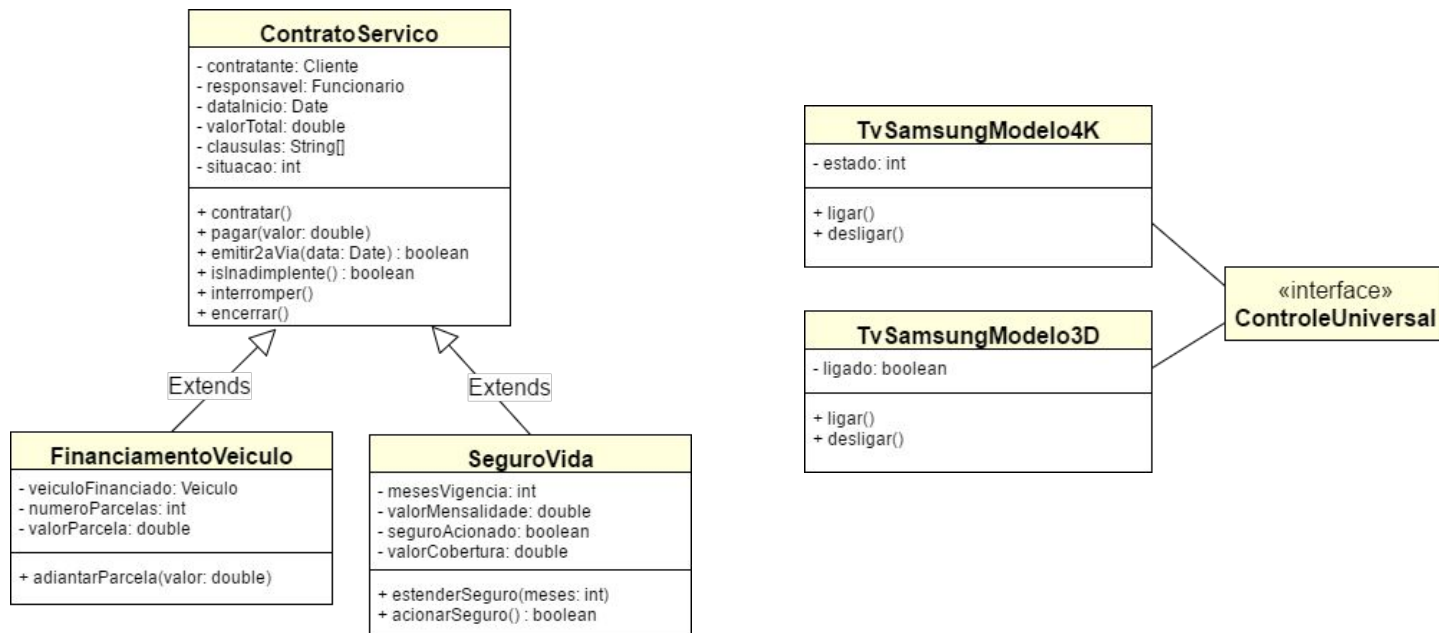
```
public class ClienteControleRemoto {  
    public static void main(String[] args) {  
        TvSamsungModelo4K modelo4K = new TvSamsungModelo4K();  
        TvSamsungModelo3D modelo3D = new TvSamsungModelo3D();  
        ligar(modelo4K);  
        ligar(modelo3D);  
        desligar(modelo4K);  
        desligar(modelo3D);  
    }  
  
    public static void ligar(ControlUniversal tvSamsung) {  
        tvSamsung.ligar();  
    }  
  
    public static void desligar(ControlUniversal tvSamsung) {  
        tvSamsung.desligar();  
    }  
}
```

## 4º Pilar - Polimorfismo

---

- Polimorfismo é a habilidade de esconder diferentes implementações através de uma única interface. Portanto, através do Polimorfismo, é possível tratar objetos criados a partir de classes específicas como objetos de uma classe genérica.
- Existem duas formas de implementá-lo:
  - Através de Herança: a classe ancestral permite representar características e comportamentos de um conjunto de objetos provenientes de classes filhas
  - Através de Interface: a interface permite representar comportamentos comuns de um conjunto de objetos que a implementam

## Polimorfismo - Exemplos





## Exercício 1

---

- Faça a abstração de uma classe que represente uma **lâmpada**, a qual possui o estado **acesa** ou **apagada**
- As operações sobre ela são **ligar** e **desligar**
- Crie uma lâmpada e imprima seu estado após acendê-la e também após apagá-la

## Exercício 2

---

- Crie uma classe ancestral para representar uma Conta, contendo as seguintes operações:
  - depositar, contendo o parâmetro valor
  - sacar, contendo o parâmetro valor
  - consultarSaldo, retornando o valor do saldo atual
- Crie uma classe ContaCorrente descendente dessa classe, e que tenha:
  - Um atributo saldo
  - Uma constante contendo o valor 0,45 de taxa por operação
  - Os métodos exigidos da interface, sendo que todos eles devem debitar o valor da taxa (para depósitos, essa dedução ocorre após a transação; nas outras transações, a dedução ocorre antes)

## Exercício 2 (cont.)

---

- Crie uma classe ContaPoupanca que seja similar à classe ContaCorrente, mas que não realize débitos de taxa por operação

**Obs.:** se houverem atributos em comum das classes descendentes, coloque-os na classe ancestral

- Crie uma classe TestaContas que utiliza ambos os tipos de conta, testando operações de depósito, saque e consulta de saldo

### Exercício 3

---

- Implemente o exercício 2 utilizando interface na entidade Conta
- Teste essa implementação da mesma forma que foi feito no exercício 2