



JAVA

ORIENTAÇÃO A OBJETOS

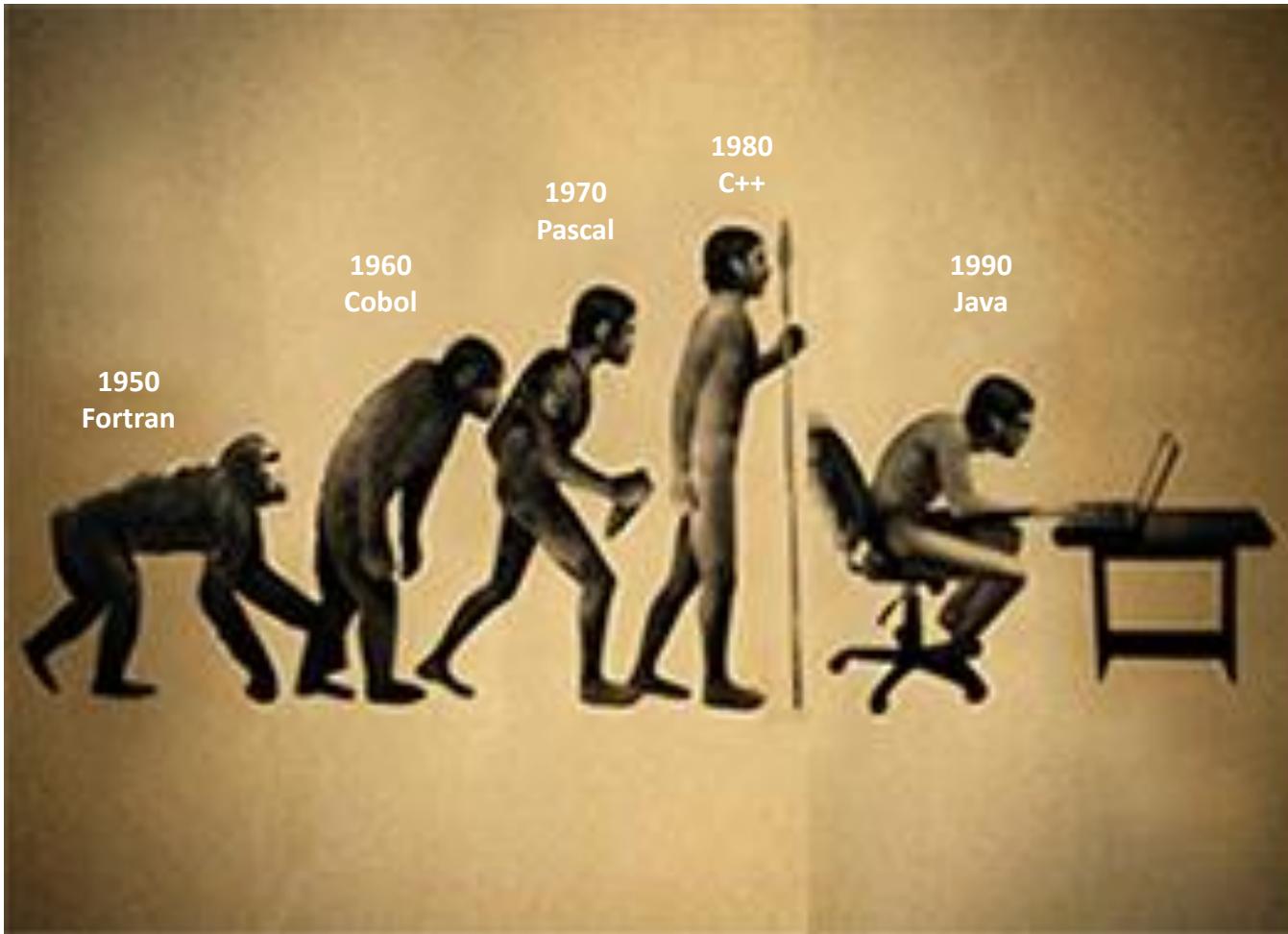
Prof.^a: Flavia Garcia

SENAC RIO

Competências

- Desenvolver estruturas de repetição e de decisão
- Armazenar objetos em estruturas com tamanho fixo ou variável
- Desenvolver, instanciar e manipular classes, aplicando encapsulamento de dados e criando relação de associação e de herança entre as mesmas.
- Criar e utilizar referências polimórficas
- Criar interfaces gráficas para aplicações Desktop, usando componentes visuais da linguagem.
- Conectar a um SGBD e operar consultas SQL utilizando JDBC
- Tratar erros e exceções utilizando os recursos da linguagem.

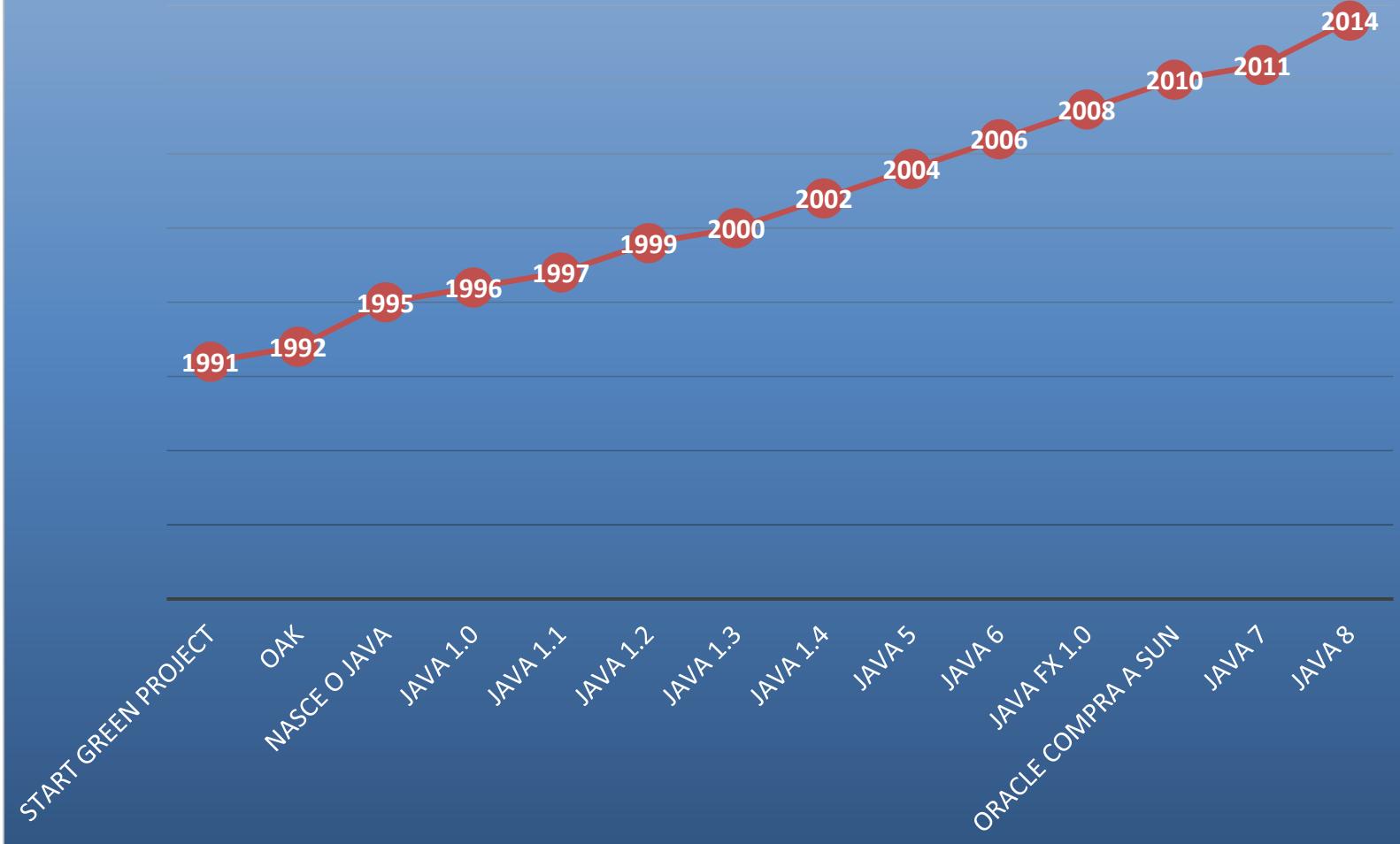
Evolução das Linguagens



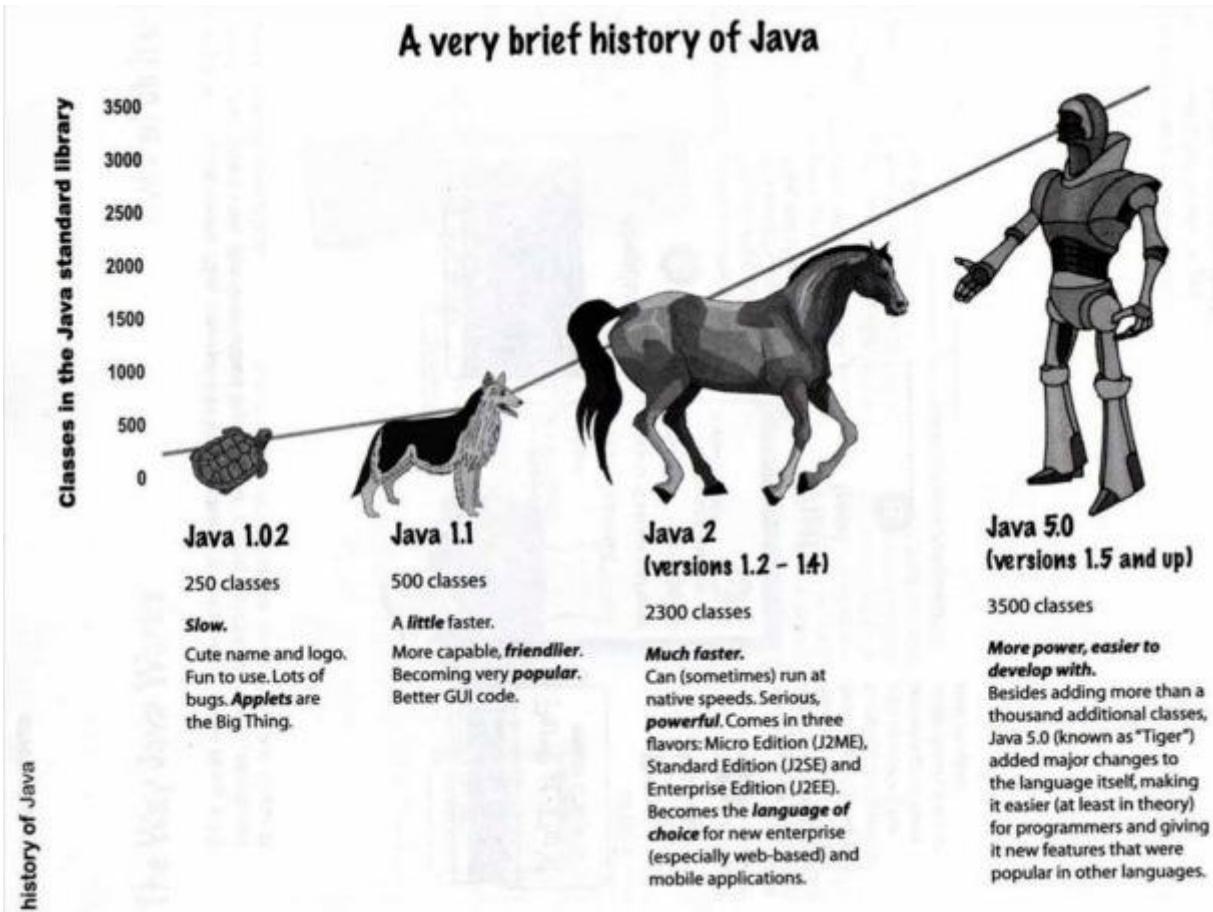
Introdução – Nasce um gigante

- O Java nasceu para solucionar problemas de hardware. Ele já nasceu pra ser SMART.
 - S específico
 - M mensurável
 - A atingível
 - R realizável
 - T temporizável
- Ele era a solução para integrar diversas plataformas de hardware diferentes desenvolvido pela SUN, no início dos anos 90, na época se chamava OAK, mas ela não deu muito certo e foi deixado de lado mas, em 1994 com o grande boom da internet a SUN fez uso da sua tecnologia e a integrou ao mundo WEB.
-

Crescimento do Java



Nunca mais parou de evoluir



Histórias do Java

- <http://oracle.com.edgesuite.net/timeline/java/>
- [Internet das Coisas](#)

- Java: mais que uma linguagem de programação:

- Ambiente de desenvolvimento
- Ambiente de aplicação
- Ambiente de distribuição

Lema do Java : “Write once run anywhere.”
(Escreva uma vez, roda em qualquer lugar!)

Antes do Java

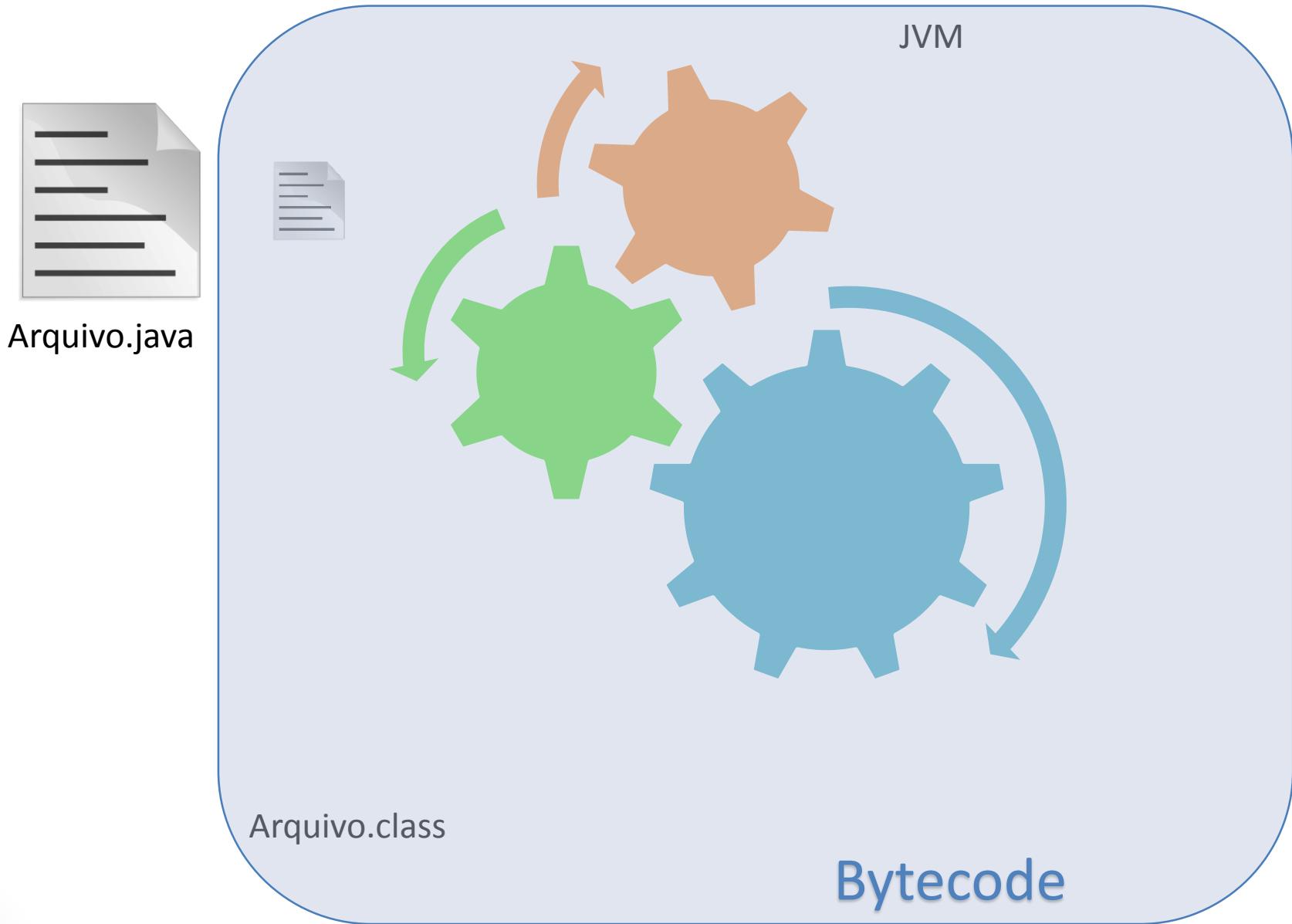
Quando de desenvolvia um algoritmo esse pertencia a plataforma do sistema operacional e não bastava simplesmente utilizar o mesmo arquivo para e utilizar em outros SO's, pois cada um tem suas particularidades de execução. Aí o mesmo código, em grande parte era reutilizado para outras plataformas, alterando somente as tais particularidades. Se tivéssemos 5 SO's diferentes ou alteração no código por conta de melhorias ou alteração nas regras de negócio, o programador deveria reescrever o mesmo código 5 vezes.

Depois do Java

O código que até então era compilado e gerava um arquivo executável e com o Java ele gera um byte code. O byte code nem é hardware e nem software, ele está no meio do caminho que através da JVM pega esse byte code e transforma para cada linguagem de programação. Como isso matamos o problema de plataformas e manutenção de código.

Uma JVM isola totalmente a aplicação do sistema operacional. Se uma JVM termina abruptamente, só as aplicações que estavam rodando nela irão terminar: isso não afetará outras JVMs que estejam rodando no mesmo computador, nem afetará o sistema operacional.

Como funciona em Java



Como funciona a JVM

APP

JVM

S.O.

Hardware

- 1^a camada - Hardware
- 2^a camada - Sistema Operacional
- 3^a camada - Midleware (máquina virtual Java - JVM (Java Virtual Machine))

A máquina virtual não é portável, foi criada uma máquina virtual para cada sistema operacional.
As aplicações Java é que são portáteis.

Performance

- Velocidade
- Arquitetura JVM (computador dentro do outro)
- Atualidade (Java 8)
 - Hotspot – detecta o código que foi processado muitas vezes em sua aplicação.
 - JIT – recompila o código para melhorar o desempenho e o tempo de acesso.
 - Garbage Collections (Gerenciamento de Memória)

Por que usar o Java?

- Mais algumas reflexões antes de começarmos o trabalho:
 - Manutenção, reaproveitamento, escalabilidade, baixo acoplamento, alta coesão;
 - Bibliotecas gratuitas e padronizadas;
 - Especificação vs Implementação da JVM;
 - Posso comprar suporte e performance;
 - É só desplugar o bytecode;
 - Independência de HW, SO e VM;
 - IDEs e programação pura;

Atividade 1 - Pesquisa

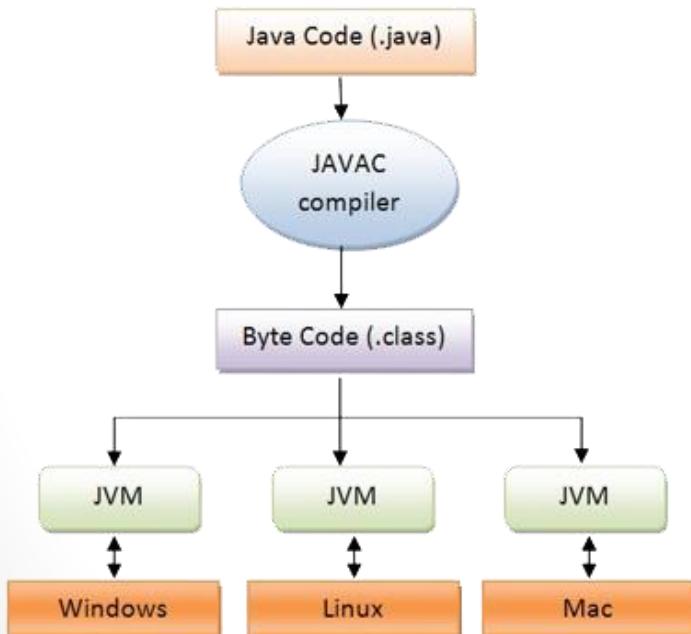
- Acesse, no *site* da Oracle, as áreas sobre Java e descubra por que ele é uma das tecnologias em maior desenvolvimento e aceitação do mundo.
- Pesquise também sobre o Code Conventions
- Convenções de Código para a Linguagem de Programação **Java**

Java – Instalação e configuração

- JRE, JDK o que são?
 - JRE – Java Runtime Environment
 - Ambiente de execução Java, formado pela JVM e bibliotecas, tudo que precisa para executar uma aplicação Java.
 - JDK – Java Development Kit
 - Ferramenta para desenvolvedores, formado pela JRE somado a algumas ferramentas como o compilador Java.

Java – Instalação e configuração

- E JVM o que é?
 - JVM – Virtual Machine (máquina virtual)
 - Vem inclusa no JRE, e é a camada responsável por “traduzir” (mas não apenas isso) o que sua aplicação deseja fazer para as respectivas chamadas do sistema operacional aonde ela está rodando.



Dessa forma, a maneira com a qual você abre uma janela no Linux ou no Windows ou qualquer outro sistema operacional é a mesma. Você ganha independência de sistema operacional. Ou seja, independência de plataforma em geral.

Tecnologia Java

- Visitando o site do Java
 - <http://java.sun.com>
 - <http://www.oracle.com/technetwork/java/index.html>

Java – Instalação e configuração

- Instalar e configurar o ambiente de trabalho
 - Primeiramente precisamos Instalar o JRE e JDK.
 - Para isso acesse o link da [Oracle](#) e baixe o JRE e JDK
 - Após isso, precisamos configurar as variáveis de ambiente do Java
- Primeiro passo:
- abra a propriedades do Sistema e vá em Configurações avançadas do sistema.
Selecione a guia Avançado e clique em Variáveis de Ambiente...

Java – Instalação e configuração

- Instalar e configurar o ambiente de trabalho

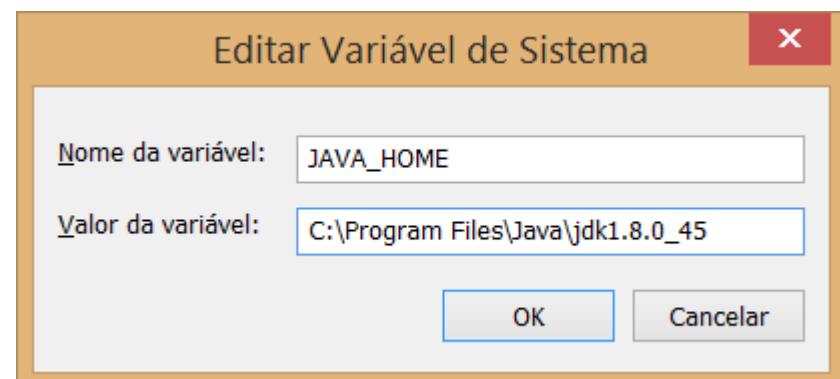
ATENÇÃO: Nesse ponto, é necessário muita atenção para não ocorrer erros, existem processos que podem danificar o sistema operacional.

Em Variáveis do sistema clique em Novo... E adicione as informações:

Nome da variável: JAVA_HOME

Valor da variável: Endereço de instalação do JDK

Após inserir essas informações clique em OK.



Java – Instalação e configuração

- Instalar e configurar o ambiente de trabalho

Vamos criar a variável de ambiente para as classes do Java

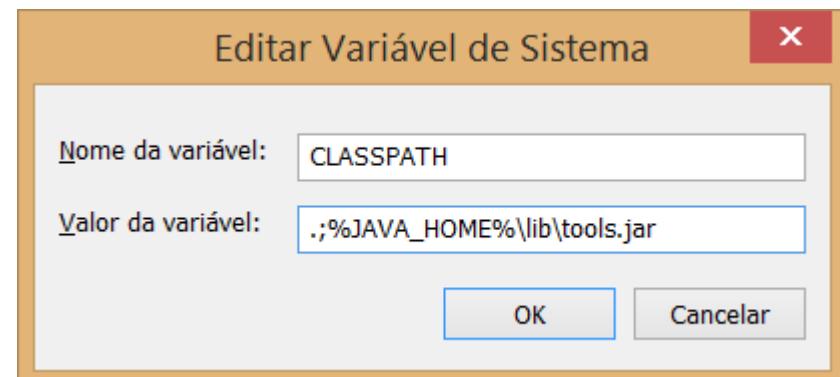
Mesmo processo de como criamos a variável JAVA_HOME

Em Variáveis do sistema clique em Novo... E adicione as informações:

Nome da variável: CLASSPATH

Valor da variável: .;%JAVA_HOME%\lib\tools.jar

Após inserir essas informações clique em OK.



Java – Instalação e configuração

- Instalar e configurar o ambiente de trabalho

ATENÇÃO: Esse passo pode danificar o sistema operacional.

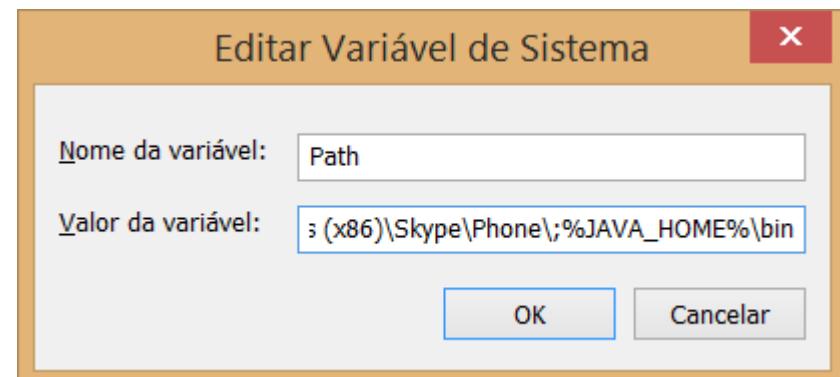
Vamos editar a variável de ambiente path para o sistema reconhecer o Java
Mesmo processo de como criamos a variável JAVA_HOME

Em Variáveis do sistema localize a variável Path

Adicione ao final do valor da variável:

`;%JAVA_HOME%\bin`

Após inserir essa informação clique em OK.



Java – Primeiros contatos

- Instalando e Configurando o Eclipse
 - Precisamos baixar o Eclipse, para acesse o site Eclipse.org e baixe a ultima versão do Eclipse IDE for Java EE Developers e extraia ele em um diretório de seu gosto.
 - Ao abrir o Eclipse, não há necessidade de alterar o caminho do workspace informado por ele, porém, precisamos alterar a perspectiva dele, o Eclipse que foi baixado por você, vem configurado com a versão Enterprise Edition, logo precisamos alterar para Standard Edition.
 - Para fazer isso, vá no canto superior a direita e clique na janela com um símbolo de (+) em amarelo.



Iniciando o uso de Java

- JVM: Máquina Virtual Java, não existe sozinha para download
- JRE: Java SE Runtime Environment (**JVM mais as Bibliotecas**)
- JDK: Java SE ou Java EE Development Kit, JRE com as ferramentas de desenvolvimento como compilador javac, interpretador java (que instancia uma máquina virtual), gerador de documentação javadoc, compactador jar, entre outros, todos no \bin do JAVA_HOME.
- IDE - Sistema de desenvolvimento integrado (Netbeans ou Eclipse)

Java SE – *Compilação e Execução...*

- Verificando o PATH da máquina:
 - c:\>set path
- Verificando a versão instalada do Java:
 - c:\>`java -version`
- Um programa fonte:
 - `NomedaClasse.java`

Java SE – *Compilação e Execução...*

- Compilação:
 - c:\>`javac` `NomedaClasse.java`
- ByteCodes:
 - `NomedaClasse.class`
- Execução (Interpretador):
 - c:\>`java` `NomedaClasse`

Iniciando o uso de Java

- Abra o seu editor preferido e digite o conteúdo abaixo:

```
class MeuPrimeiroPrograma {  
  
    public static void main(String[] args) {  
        System.out.println("O primeiro de muitos!");  
    }  
}
```

Para executá-lo, ainda pelo terminal, digite o comando `java MeuPrimeiroPrograma`. O comando `java` é o responsável por executar a JVM (máquina virtual, ou *Java Virtual Machine*) que irá interpretar o *byte-code* de seu programa. Repare que neste comando não passamos a extensão do arquivo:

```
Administrador: C:\Windows\system32\cmd.exe
3/07/2016  15:43      <DIR>          jdk1.8.0_92
3/03/2017  09:23      <DIR>          jre1.8.0_121
    0 arquivo(s)           0 bytes
    4 pasta(s)   273.447.129.088 bytes disponíveis

::\Program Files\Java>cd jdk1.8.0_92

::\Program Files\Java\jdk1.8.0_92>javac java1.java
javac' não é reconhecido como um comando interno
ou externo, um programa operável ou um arquivo em lotes.

::\Program Files\Java\jdk1.8.0_92>javac
javac' não é reconhecido como um comando interno
ou externo, um programa operável ou um arquivo em lotes.

::\Program Files\Java\jdk1.8.0_92>dir
O volume na unidade C é SISTEMA
O Número de Série do Volume é 76FB-0696

Pasta de C:\Program Files\Java\jdk1.8.0_92

3/07/2016  15:43      <DIR>          .
3/07/2016  15:43      <DIR>          bin
3/07/2016  15:42      <DIR>          COPYRIGHT
1/03/2016  21:31            3.244 db
3/07/2016  15:42      <DIR>          include
3/07/2016  15:42            5.090.296 javafx-src.zip
3/07/2016  15:42      <DIR>          jre
3/07/2016  15:42      <DIR>          lib
3/07/2016  15:42            40 LICENSE
3/07/2016  15:42            159 README.html
3/07/2016  15:42            527 release
1/03/2016  21:31            21.248.736 src.zip
3/07/2016  15:42            110.114 THIRDPARTYLICENSEREADME-JAVAFX.txt
3/07/2016  15:42            177.094 THIRDPARTYLICENSEREADME.txt
    8 arquivo(s)           26.630.210 bytes
    7 pasta(s)   273.447.129.088 bytes disponíveis

::\Program Files\Java\jdk1.8.0_92>cd bin

::\Program Files\Java\jdk1.8.0_92\bin>javac java1.java
javac: file not found: java1.java
usage: javac <options> <source files>
use -help for a list of possible options

::\Program Files\Java\jdk1.8.0_92\bin>cd teste

::\Program Files\Java\jdk1.8.0_92\bin\teste>javac java1.java
javac' não é reconhecido como um comando interno
ou externo, um programa operável ou um arquivo em lotes.

::\Program Files\Java\jdk1.8.0_92\bin\teste>cd..
::\Program Files\Java\jdk1.8.0_92\bin>javac java1.java
::\Program Files\Java\jdk1.8.0_92\bin>java java1
1

::\Program Files\Java\jdk1.8.0_92\bin>
```

Estrutura do Programa em Java

```
public class AloMundo {  
    // Comentário de uma linha  
    /* Comentário de mais de  
       uma linha */  
    /** Comentário de documentação */  
    public static void main (String[] args) {  
        // Código fonte do programa  
    }  
}
```

Nome da Classe

Rotina Principal

Saída de Dados

Para saída dos dados podemos usar um dos comandos:

`System.out.print()`

`System.out.println()`

```
public class AloMundo {  
    / **  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {  
        System.out.println("Alo Mundo");  
    }  
}
```

System.out

É a saída padrão do sistema

A mensagem
(Expressão)

Entrada de Dados

Pode ser usada a classe Scanner do pacote `java.util`

- ▶ Retorna entradas no teclado nos tipos primitivos: String, int, double, float, etc
- ▶ Métodos
 - `next()`
 - `nextInt()`
 - `nextDouble()`
 - `nextFloat()`

Exemplo de Entrada de Dados

```
import java.util.Scanner;           Indica que queremos utilizar a classe Scanner

public class GetInputFromKeyboard {

    public static void main(String[] args) {

        Scanner input = new Scanner(System.in);      Cria uma variável do tipo Scanner

        String nome = "";                            Recebe a entrada do usuário

        System.out.printf("Digite o seu nome " );
        nome = input.next();

        System.out.printf("\nMeu nome é %s", nome);
    }
}
```

Exemplo :

```
class MeuPrimeiroPrograma {  
  
    public static void main(String[] args) {  
        System.out.println("O primeiro de muitos!");  
    }  
}
```

Métodos

- Todos os códigos executáveis dentro de uma aplicação é dividido em pequenos pedaços. Fazendo analogia com algoritmos, são as funções. Em Java, os métodos são controlados por modificadores de acesso.
- Em sua declaração, vale lembrar que, o nome de um método possui letra minúscula, caso o nome seja composto, o primeiro nome com letra minúscula e o segundo com letra maiúscula e identificados com () .
- Os métodos podem ter ou não retorno, possuir ou não passagem de parâmetros.

Método Main

- É utilizado para console de aplicação, ou seja, ele chama os métodos criados nas classes e os executa. Podemos ter diversas classes, mas apenas um main no console. É o chamado método principal, ou seja, na execução o SO o chama primeiro. Sua estrutura é:

```
public static void main (String[] args){  
}
```

- **public** – visibilidade;
- **static** – o método pode ser chamado sem criar um objeto de tipo de classe;
- **void** – método retorna vazio;
- **args** – vetor de Strings representando os argumentos passados.

Algumas regras de convenções

Você pode ter reparado que seguimos algumas regras e convenções até agora. Vamos entendê-las um pouco melhor.

Podemos começar pelo nome de nosso arquivo, `MeuPrimeiroPrograma`. Em Java, o nome de uma classe sempre se inicia com letra maiúscula e, quando necessário, as palavras seguintes também têm seu *case* alterado. Dessa forma “*esse nome de classe*

vira “*EsseNomeDeClasse*

. Essa abordagem é bastante conhecida como *CamelCase*.

Você pode ler mais a respeito dessas e de outras convenções da linguagem no documento oficial *Code Conventions for the Java Programming Language*, disponível no site da Oracle:

<http://www.oracle.com/technetwork/java/index-135089.html>

Convenções para nomes

Embora não seja de uso obrigatório, existe a convenção padrão para atribuir nomes em Java, como:

- Nomes de classes são iniciados por letras maiúsculas;
- Nomes de métodos ,atributos e variáveis são iniciados por letras minúsculas;
- Em nomes compostos, cada palavra do nome é iniciada por letra maiúscula, as palavras não são separadas por nenhum símbolo.

A Regra É Clara!

- Seu nome é sempre com letra maiúscula
- Uma classe possui vários blocos de instrução;
- Os blocos de instrução iniciam e terminam com { }
- Todas as classes possuem métodos. Todos os métodos possuem ();
- Métodos podem ter ou não argumentos (parâmetros de entrada)
- Antes do nome do método temos algumas palavras reservadas que serão explicadas futuramente;
- O modificador de acesso da classe pode ser public, private ou abstract.
- Um arquivo pode conter várias classes, só uma pública e essa deve ter o mesmo nome do arquivo
- Sempre a primeira linha de execução é o main();
- Todas as linhas terminam com ;
- É case sensitive (diferencia maiúscula de minúscula)
- // para comentários de uma linha
- /* */ para comentários de blocos
- /** */ gerar javadoc



ERRO DE COMPILAÇÃO?

Um erro de digitação, a falta de um ponto e vírgula ou uma diferença de *case* em seu código são alguns dos muitos motivos que podem resultar em um erro de compilação. Conhecer e entender esses erros é fundamental, talvez você queira inclusive provocar algum deles para ver como seu código se comporta.

Qual será a mensagem caso eu esqueça de escrever um ponto e vírgula? Escreva um teste simples pra descobrir! Um exemplo:

```
System.out.println("sem ponto-e-virgula")
```

A mensagem de erro será:

```
Syntax error, insert ";" to complete BlockStatements
```

Lembre-se que você pode e deve tirar todas as suas dúvidas no GUJ, em <http://guj.com.br>.

Trabalhando com a IDE

- A IDE ou interface de desenvolvimento foram criadas para tornarem a codificação mais produtiva, oferecendo recursos como atalhos e *auto complete*. Adotaremos o Eclipse para o desenvolvimento da parte core e para o desenvolvimento das telas o NetBeans.
- *Os principais atalhos são:*
 - **Control + espaço** = ajuda no *auto complete*, inclusive com sugestões de uso métodos e atributos.
 - **Control + 1** = sugestões quando algo há algum erro de compilação (*quickfix*), indicado pela pequena lâmpada, como no tratamento de exceções checked ou para criar classes que ainda não existem.
 - Mas o uso mais interessante é na criação de variáveis e atributos. Quando vamos criar uma variável, começamos declarando-a, mas isso não é necessário. Podemos simplesmente digitar new ContaCorrente() e pressionar Control + 1.
 - **Control + 3** = Ele é o atalho que busca um comando ou opção de menu baseado no que você escreve.

Outros Atalhos

- Control + Shift + O = importar ou remover bibliotecas.
- Control + Shift + F = organizar o código de forma endentada.
- Control + M = Modifica o ambiente de desenvolvimento
- Control + F11 = Executa o código
- F11 = debug
- Control + L = exibe a lista de atalhos
- Alt + Shift + N = novo

Essas ferramentas são chamadas de IDE (*Integrated Development Environment*) e podem tornar seu desenvolvimento muito mais produtivo e interessante, oferecendo-lhe recursos como *syntax highlight* e *auto complete* das instruções de seu código.

No decorrer deste livro, vamos utilizar o *Eclipse*, você pode fazer o download de sua versão mais recente em:

<https://www.eclipse.org/downloads/>

Essa é uma IDE gratuita e open source, sem dúvida uma das preferidas do mercado e instituições de ensino. Seus diversos atalhos e templates prontos são grandes diferenciais.

Existem diversas outras boas opções no mercado, como por exemplo o *NetBeans* da Oracle, ou o *IntelliJ IDEA*, sendo este último pago.

Criando seu primeiro projeto no Eclipse

Vamos criar nosso projeto! Para isso, você pode por meio do menu escolher as opções `File > New > Java Project`. Vamos chamar o projeto que desenvolveremos durante o curso de `livraria`.

Repare que, depois de concluir, esse projeto será representando em seu sistema operacional como uma pasta, dentro do `workspace` que você escolheu. Por enquanto, dentro desse diretório você encontrará a pasta `src`, onde ficará todo o seu código `.java` e também a pasta `bin`, onde ficará o `bytecode` compilado.

Agora que já criamos o projeto, vamos criar nossa primeira classe pelo Eclipse. Você pode fazer isso pelo menu File > New > Class. Para conhecer um pouco da IDE, vamos criar novamente a classe MeuPrimeiroPrograma.

Depois de preencher o nome, selecione a opção *finish* e veja o resultado:

```
public class MeuPrimeiroProgramma {  
}
```

A estrutura de sua classe já está pronta, vamos agora escrever seu método main. Para fazer isso, escreva a palavra main dentro de sua classe e pressione o atalho Control + Espaço.

Esse é o atalho de *code completion* da IDE. Se tudo correu bem, seu código ficou assim:

```
public class MeuPrimeiroPrograma {  
  
    public static void main(String[] args) {  
  
    }  
}
```

Interessante, não acha? Além de termos mais produtividade ao escrever, evitamos que erros de digitação aconteçam.

Vamos além, agora dentro do método `main`, digite `sys` e pressione `Control + Espaço` para fazer *code completion* novamente.

Você pode e deve usar e abusar desse recurso!

```
public class MeuPrimeiroProgramma {  
  
    public static void main(String[] args) {  
        System.out.println("O primeiro de muitos!");  
    }  
}
```

Agora, com o código pronto, as próximas etapas seriam compilar e executá-lo, mas a compilação já está pronta!

Isso mesmo, conforme você vai escrevendo seu código, a IDE já cuida de compilá-lo. Repare que se você apagar o ponto e vírgula, por exemplo, essa linha ficará sublinhada em vermelho. Essa é uma indicação visual de que seu código não está compilando.

Para executar o código, você pode clicar com o botão direito do mouse em sua classe e selecionar as opções `Run As > Java Application`. Ou por atalho, pressionando `Control + F11`.

Repare que a saída de seu código vai aparecer na aba `Console`.

O primeiro de muitos!

Java SE – *Um programa Java...*

- Sintaxe básica:

`<NomeDoPrograma>.java`

- Exemplo:

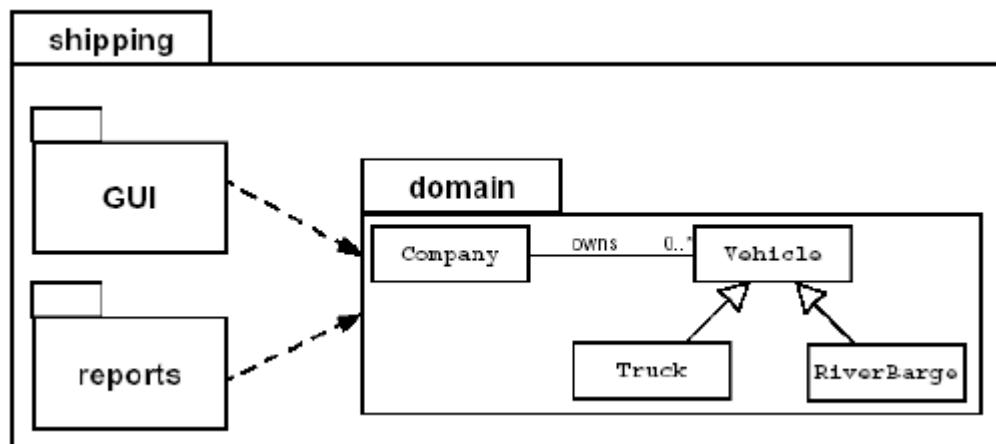
`MeuPrograma.java`

`IncluirServicoServlet.java`

`Utilitarios.java`

- **Regras:**
 1. O nome do arquivo deve ser igual ao nome da classe.
 2. Um arquivo pode ter **n** classes, *mas apenas uma é pública.*
 3. O nome do arquivo deve ser igual ao nome da classe.

- *Pacote:*
 - Sistemas são grandes -> Agrupar classes em pacotes.
 - Facilitar a manutenção e o gerenciamento.
 - Notação UML:



Java SE – A Instrução Package...

- **Regras:**
 1. Só pode existir uma (1) declaração de pacote por programa fonte.
 2. Os nomes dos pacotes devem ser hierárquicos e separados por ponto.
 3. Se não houver declaração de pacotes no programa fonte, todas as classes pertencerão a um pacote *default* que não possui nome.
 4. Quando um arquivo Java contendo uma instrução Package é compilado, o arquivo *.class* resultante é colocado no diretório especificado pela instrução package.

Java SE – A Instrução Import...

- Indica ao Compilador como achar as classes dentro de um pacote e portanto, usar os métodos e atributos destas classes.

- Sintaxe básica:

```
import <pkg_name>[.<sub_pkg_name>].<class_name>;  
import <pkg_name>[.<sub_pkg_name>].*;
```

- Regras:

1. As declarações *import* devem preceder todas as declarações de classe.

Java SE – *Desafiando...*

- Criar o projeto *PrjDesafio2*.
 - Criar a classe executável *Desafio2*.
 - Criar um método **somar** na classe **Desafio2** que receba dois parâmetros, **x** e **y**, do tipo inteiro.
 - O método deve retornar o valor da soma destes dois parâmetros.
 - Imprimir o resultado da execução do método **somar** no console.

Variáveis

- Uma variável é um espaço na memória usado para armazenar o estado de um objeto e os associamos a identificadores (*nomes*)
- Ao declarar uma variável, você deve escrever o tipo da variável seguido pelo seu nome, que deve seguir as regras para os identificadores
 - Podem ser iniciados com letras de A-Z ou a-z, underscore “_” e sinal de cifrão “\$”
 - Os demais caracteres do identificador podem ainda ser numéricos 0-9
 - Não podem ser usados como identificadores palavras reservadas do Java

O acesso as variáveis pode se dar de duas formas:

- **Por valor:** onde o estado (dado) é armazenado no espaço de memória onde a variável está.
 - Exemplo: int i = 10;
- **Por referência:** onde é armazenado uma referência para o endereço de memória onde o estado (dado) está armazenado
 - Exemplo: int i = 10;
int j = i;

- O Java possui 4 tipos de dados primitivos:
- Booleano ou lógico:
 - boolean
- Caracter:
 - char (16 bits)
- Número inteiro:
 - byte (8 bits),
 - short (16 bits),
 - int (32 bits)
 - long (64 bits)
- Número com ponto flutuante:
 - float (32 bits)
 - double (64 bits)

- O Java é uma linguagem fortemente tipada, assim, toda variável deve ter um tipo e este não pode ser mudado
- Uma variável em Java pode ser declarada em qualquer lugar do código e com o operador de atribuição “=” pode ser inicializada com o valor desejado
- A declaração e a inicialização podem ser feitas na mesma linha

- O tipo booleano tem domínio lógico, no caso de Java true ou false:

```
13 ...
14 boolean resultado;
15 resultado = false;
16 ...
17 ...
```

OU

```
4 ...
5 boolean resultado = false;
6 ...
```

```
32 ...
33 char c = 'f';
34
35 byte b = 5;
36
37 short s = 10;
38
39 int i = 15;
40
41 long x = 100L;
42
43 double d = 1000.0;
44
45 float f = 1500.0f;
46
47 System.out.println(i);
48
49 System.out.println("O valor de i é: "+x);
50
51
```

Faça uma classe para testar os tipos primitivos acima

Casting

- É o ato de moldar um tipo de dado para outro, também conhecido como *type casting*. Pode ser explícito ou automático sua lógica está associada principalmente ao tamanho do tipo em bits.

- Relembrando:
- Caracter: char (16 bits)
- Número inteiro:
 - byte (8 bits), short (16 bits),
 - int (32 bits) , long (64 bits).
- Número com ponto flutuante:
 - float (32 bits) , double (64 bits).

- Se você tentar compilar o código abaixo receberá um erro, pois está atribuindo um valor double a um int.

```
4  double d = 10.50;  
5  
6  int i = d;  
7
```

- Neste caso, apesar do número atribuído à variável double ser um inteiro, ele foi armazenado inicialmente num espaço de memória cujo espaço é de um double, sendo assim, não cabe num espaço reservado para um int.
- Ou seja, double pode conter um int, pois é menor que um double, mas um int não pode conter um double.
- Caso você precise armazenar um número com ponto flutuante num inteiro, você até pode fazer isso, mas ele será arredondado.

- Isso é o processo de ***casting***, porém você deve, antes de atribuir, dizer explicitamente que a variável passará a ser um inteiro, da seguinte maneira:

```
4      double d = 10.50;
5      int i = (int) d;
6      System.out.println(i);
7  }
8 }
```

Constantes

- São identificadores que não podem ter seu valor alterado. Em java eles são controlados pelo prefixo ***final*** e o nome da constante deve ter todas as letras maiúsculas.
 - ***Exemplo:***
 - ***final double PI = 3.14159;***
 - ***final int ANO = 2016;***

Trabalhando com Caracteres e Strings

- Os métodos que trabalham com caracteres retornam booleanos

Métodos	Descrição
isDigit	Retorna verdadeiro se for um digito
isLetter	Retorna verdadeiro se for uma letra
isLetterOrDigit	Retorna verdadeiro se for um digito ou uma letra
isSpace	Retorna verdadeiro se for um espaço
toLowerCase	Retorna o caractere minúsculo
toUpperCase	Retorna o caractere maiúsculo
replace	Substitui um caractere por outro

```
public class TesteStrings {  
    public static void main(String args[]) {  
        System.out.println(Character.isDigit('c'));  
        System.out.println(Character.isDigit('5'));  
  
        System.out.println(Character.isLetter('c'));  
        System.out.println(Character.isLetter('5'));  
  
        System.out.println(Character.isLetterOrDigit('c'));  
        System.out.println(Character.isLetterOrDigit('5'));  
  
        System.out.println(Character.isUpperCase('c'));  
        System.out.println(Character.isUpperCase('C'));  
  
        System.out.println(Character.isLowerCase('c'));  
        System.out.println(Character.isLowerCase('C'));  
  
        System.out.println(Character.isWhitespace(' '));  
  
        String a = "testejava.html";  
        a = a.replaceAll("html", "jsp");  
        System.out.print(a);  
    }  
}
```

Método	Propósito
equals	Compara duas strings e retorna true se forem equivalentes
equalsIgnoreCase	Compara duas strings enquanto ignoram casos de letras
startsWith	Retorna true se a palavra inicia com a sequencia especifica de caracteres
endsWith	Retorna true se a palavra termina com a sequencia especifica de caracteres
compareTo	Retorna -1 se a primeira string tiver precedência sobre a segunda. 0 se forem iguais. 1 se a primeira segue a segunda.

Métodos	Propósitos
length	Retorna o tamanho da string
charAt	Retorna a posição de um caractere fornecido de uma string
substring	Retorna partes de uma string
indexOf	Retorna a posição da primeira ocorrência de um char ou string
lastIndexOf	Retorna a posição da última ocorrência de um char ou string

Operadores Aritméticos

- $a + b \Rightarrow$ soma o valor de a e b
- $a - b \Rightarrow$ subtrai o valor b de a
- $a * b \Rightarrow$ multiplica o valor de a por b
- $a / b \Rightarrow$ divide o valor de a por b
- $a \% b \Rightarrow$ resto inteiro da divisão de a por b
- Faça uma classe para testar os operadores acima

Operadores Relacionais

- Operadores relacionais: comparam dois valores e determinam seu relacionamento, sendo que sua saída é sempre um resultado lógico true ou false.

Operador	Uso	Descrição
>	v1 > v2	v1 é maior do que v2
>=	v1 >= v2	v1 é maior ou igual a v2
<	v1 < v2	v1 é menor do que v2
<=	v1 <= v2	v1 é menor ou igual a v2
==	v1 == v2	v1 é igual a v2
!=	v1 != v2	v1 é diferente de v2

Operadores Lógicos

Expressão lógica é qualquer expressão que retorne *true* ou *false* e podem ser concatenadas com operadores lógicos

E	OU	Negação
&		!
&&		
(and)	(or)	

Vamos criar uma nova classe Java cujo objetivo será calcular o valor total de livros do nosso estoque na livraria. Podemos chamá-la de CalculadoraDeEstoque.

```
public class CalculadoraDeEstoque {  
    public static void main(String[] args) {  
        }  
    }
```

Precisamos agora armazenar dentro de seu método `main` o valor de nossos livros. Uma forma simples de fazer isso seria:

```
public class CalculadoraDeEstoque {  
    public static void main(String[] args) {  
        double livroJava8;  
        double livroTDD;  
    }  
}
```

Praticando...

Dessa forma, estamos pedindo ao Java que reserve regiões da memória para futuramente armazenar esses valores. Essa instrução que criamos é conhecida como uma variável.

Repare que essas duas variáveis declaradas possuem um tipo `double` (número com ponto flutuante). Logo, conheceremos os outros tipos existentes, mas é importante reconhecer desde já que toda variável em Java precisaria ter um.

Utilizamos o sinal `=` (igual) para atribuir um valor para estas variáveis que representam alguns de nossos livros:

```
double livroJava8;  
double livroTDD;  
  
livroJava8 = 59.90;  
livroTDD = 59.90;
```

Mas, neste caso, como já sabemos o valor que será atribuído no momento de sua declaração, podemos fazer a atribuição de forma direta no momento em que cada variável é declarada:

```
double livroJava8 = 59.90;  
double livroTDD = 59.90;
```

Agora que já temos alguns valores de livros, vamos calcular a sua soma e acumular em uma nova variável. Isso pode ser feito da seguinte forma:

```
double soma = livroJava8 + livroTDD;
```

Pronto, isso é tudo que precisamos por enquanto para completar a nossa CalculadoraDeEstoque. Após somar esses valores, vamos imprimir o valor do resultado no *console*:

```
public class CalculadoraDeEstoque {  
    public static void main(String[] args) {  
        double livroJava8 = 59.90;  
        double livroTDD = 59.90;  
  
        double soma = livroJava8 + livroTDD;  
  
        System.out.println("O total em estoque é " + soma);  
    }  
}
```

Escreva e execute esse código para praticar! Neste exemplo, o resultado será:

O total em estoque é 119.8

Vamos praticar ?

Operadores aritméticos, entrada e saída de dados

1. Desenvolva um programa que escreva o seu nome completo, colocando uma palavra em cada linha.
2. Desenvolva um programa que leia um número inteiro e mostre seu sucessor
3. Desenvolva um programa que lê dois números inteiros, X e Y, e mostre o resultado da multiplicação de x por y
4. Desenvolva um programa que lê o salário de um funcionário, reajusta o salário em 7% e mostra o resultado.
5. Desenvolva um programa que lê um valor real em dólar, e converte o valor para reais. Considere que a cotação é US\$ 1 = R\$ 1,82.
6. Desenvolva um programa que leia as variáveis inteiros x e y e troque o valor destas variáveis. Isto é, x deve ficar com o valor de y e y deve ficar com o valor de x. Mostre os valores depois da troca.

Operadores Condicionais

- É uma forma condensada a estrutura *if*. Sua estrutura é resumida em uma única expressão:

```
package testes;
public class Condicionais {
    public static void main(String[] args) {
        String resultado = "";
        int i = 20;
        resultado = (i>=20) ? "sim" : "nao" ;
        System.out.println(resultado);
    }
}
```

Estruturas de Controle por Decisão

- Executam um bloco de instruções de acordo com um tipo de crítica ou avaliação. São elas:
 - IF
 - IF – ELSE
 - IF – ELSE – IF
 - SWITCH

IF

```
float media = 10;  
if (media >= 7){  
    System.out.println("Aprovado !");  
}
```

IF - ELSE

```
float media = 10;  
if (media >= 7){  
    System.out.println("Aprovado !");  
} else {  
    System.out.println("Reprovado !");  
}
```

IF – ELSE -IF

```
float media = 10;  
if (media >= 7){  
    System.out.println("Aprovado ! ");  
} else if (media <= 5){  
    System.out.println("Reprovado ! ");  
} else {  
    System.out.println("Recuperação! ");  
}
```

Switch

```
Int num = 92;  
switch (num){  
    case 100:  
        System.out.println("Excelente!");  
        break;  
    case 90:  
        System.out.println("Bom trabalho!");  
        break;  
    case 80:  
        System.out.println("Pode melhorar!");  
        break;  
    default:  
        System.out.println("Desculpe, você falhou!");  
}
```

Incremento e decremento

- Podem ser pré ou pós operados:
 - `i++` => avalia a expressão antes do valor ser acrescido
 - `i--` => avalia a expressão antes do valor ser decrescido
 - `++i` => o valor é acrescido e depois a expressão é avaliada
 - `--i` => o valor é decrescido e depois a expressão é avaliada
- Faça os testes dos códigos abaixo linha-a-linha:

```
56  
57 int i = 5;  
58  
59 int j = i++;  
60  
62 int i = 10;  
63  
64  
65 int j = ++i;  
66
```

- Qual o valor das variáveis ao final do código abaixo?

```
67  
68 int i = 10;  
69  
70 int j = 5;  
71  
72 int k = j++ + i;  
73
```

```
74  
75 int i = 10;  
76  
77 int j = 5;  
78  
79 int k = ++j + i;  
80
```

- Adicionando Condicionais

Nossa CalculadoraDeEstoque precisa de uma nova funcionalidade. Se o valor total de livros for menor que 150 reais, precisamos ser alertados de que nosso estoque está baixíssimo, caso contrário, devemos mostrar uma mensagem indicando de que está tudo sob controle!

Em Java, podemos fazer essa condicional de uma forma bem comum, utilizando um `if` e `else`. Observe:

```
public class CalculadoraDeEstoque {  
  
    public static void main(String[] args) {  
  
        double livroJava8 = 59.90;  
        double livroTDD = 59.90;  
  
        double soma = livroJava8 + livroTDD;  
  
        System.out.println("O total em estoque é " + soma);  
  
        if (soma < 150) {  
            System.out.println("Seu estoque está muito baixo!");  
        } else if (soma >= 2000) {  
            System.out.println("Seu estoque está muito alto!");  
        } else {  
            System.out.println("Seu estoque está bom");  
        }  
    }  
}
```

Como já é esperado, esse código só vai imprimir a mensagem de estoque baixo se o valor da soma for menor (`<`) que 150. Caso contrário, irá executar o conteúdo de dentro do bloco `else`.

Passamos uma condição como argumento fazendo uma comparação entre o valor da variável `soma` com o valor `150`. Essa condição vai resultar em um valor `true` quando verdadeira, ou `false` caso contrário. Esse tipo de condição é conhecido como **expressão booleana**. Seu resultado sempre será do tipo `boolean`:

```
boolean resultado = soma < 150;
```

Você pode usar qualquer um dos seguintes operadores relacionais pra construir uma expressão booleana: `>` (maior), `<` (menor), `>=` (maior ou igual), `<=` (menor ou igual), `==` (igual sim, são dois iguais! Lembre-se que um único igual significa atribuição) e, por fim, `!=` (diferente).

Há ainda a alternativa de encadear mais condições em nosso `if`. Por exemplo, para receber uma mensagem indicando que o estoque está muito alto, podemos adicionar a seguinte condição:

```
public class CalculadoraDeEstoque {  
  
    public static void main(String[] args) {  
  
        double livroJava8 = 59.90;  
        double livroTDD = 59.90;  
  
        double soma = livroJava8 + livroTDD;  
  
        System.out.println("O total em estoque é "+ soma);  
  
        if (soma < 150) {  
            System.out.println("Seu estoque está muito baixo!");  
        } else if (soma >= 2000) {  
            System.out.println("Seu estoque está muito alto!");  
        } else {  
            System.out.println("Seu estoque está bom");  
        }  
    }  
}
```

Exercícios Usando Condicional

Comandos de Decisão

Desenvolva um programa que leia um número do usuário e informe se o número é positivo ou negativo.

Escreva um algoritmo que leia uma letra que represente o sexo de uma pessoa (M para Masculino e F para feminino). Se for masculino, mostra a mensagem “Seja bem-vindo, Senhor!”, se for feminino, mostra a mensagem “Seja bem-vinda, Senhora!”.

Desenvolva um programa que receba do usuário um número qualquer e verifique se esse é múltiplo de 5

Desenvolva um programa que, dada uma temperatura em graus célsius, exiba uma mensagem informando o tipo do clima, de acordo com as seguintes condições: se a temperatura estiver até 18 graus, o clima é frio; se a temperatura estiver entre 19 e 23 graus, o clima é agradável; se a temperatura estiver entre 24 e 35 graus, o clima é quente; se a temperatura estiver acima de 35 graus, o clima é muito quente

Desenvolva um programa que leia do usuário o salário e exiba uma mensagem de acordo com as seguintes condições: se o salário for até R\$ 645, escreva a mensagem “Até 1 salário mínimo; se o salário for acima de R\$ 645 e até R\$ 1935, escreva a mensagem “Até 3 salários mínimos”; se for acima de R\$ 1935 e abaixo de R\$ 3225, escreva a mensagem “Até 5 salários mínimos”; se for acima de R\$ 3225, escreva a mensagem “Acima de 5 salários mínimos”

Estruturas de Controle por Repetição

- Estruturas capazes de repetir uma parte do código numa quantidade definida ou não de vezes são chamadas de estruturas de repetição ou laços de repetição.
- As estruturas de repetição são divididas em:
 - Laços while
 - Laços do-while
 - Laços for

Laço while

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

Laço for

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

Laço do while

```
int x = 0;  
do {  
    System.out.println ( x );  
    x++;  
} while (x < 10);
```

Laço for aninhado

```
for (int i = 0; i<4;i++){  
    System.out.print ("\n");  
    for (int j= 0; j<4;j++)  
        System.out.print ( j );  
}
```

- Loopings e mais loopings

Considerando que todos os livros têm o preço 59.90, seria muito trabalhoso criar tantas variáveis para que o valor da soma ultrapasse 2000 reais. Para nos ajudar na construção desses livros podemos criar uma estrutura de repetição (um looping). Uma das formas de se fazer isso seria utilizando o while:

```
double soma = 0;  
int contador = 0;  
  
while (contador < 35) {  
    double valorDoLivro = 59.90;  
    soma = soma + valorDoLivro;  
    contador = contador + 1;  
}
```

O `while` é bastante parecido com o `if`. A grande diferença é que, enquanto sua expressão booleana for `true`, seu código continuará sendo executado. Neste exemplo estamos criando uma variável `soma` para acumular o valor total dos livros e também uma variável `contador` para controlar a quantidade de vezes que queremos iterar.

Note que estamos adicionando 35 livros, já que a condição é `contador < 35` e que a cada iteração incrementamos 1 ao valor do `contador`.

Nosso código completo fica da seguinte forma:

```
public class CalculadoraDeEstoque {

    public static void main(String[] args) {

        double soma = 0;
        int contador = 0;

        while (contador < 35) {
            double valorDoLivro = 59.90;
            soma = soma + valorDoLivro;
            contador = contador + 1;
        }
    }
}
```

```
System.out.println("O total em estoque é "+ soma);

if (soma < 150) {
    System.out.println("Seu estoque está muito baixo!");
} else if (soma >= 2000) {
    System.out.println("Seu estoque está muito alto!");
} else {
    System.out.println("Seu estoque está bom");
}
}
```

Ao executá-lo a saída será:

```
O total em estoque é 2096.500000000014
Seu estoque está muito alto!
```

Nosso código pode ficar ainda mais enxuto utilizando uma outra forma de *looping* extremamente conhecida e utilizada, o `for`.

Observe que a estrutura que utilizamos no `while` foi parecida com:

```
// inicialização do contador  
  
while (condição) {  
  
    // atualização do contador  
}
```

Com o `for`, podemos fazer isso de uma forma ainda mais direta:

```
for(inicialização; condição; atualização) {  
  
}
```

Sem dúvida, é uma forma mais direta. Você muitas vezes verá essa variável contador que criamos ser chamada de `i` (*index*, ou índice). Seguindo esse padrão, nosso código completo fica assim:

```
public class CalculadoraDeEstoque {  
  
    public static void main(String[] args) {  
  
        double soma = 0;  
  
        for (double i = 0; i < 35; i++) {  
            soma += 59.90;  
        }  
  
        System.out.println("O total em estoque é "+ soma);  
  
        if (soma < 150) {  
            System.out.println("Seu estoque está muito baixo!");  
        } else if (soma >= 2000) {  
            System.out.println("Seu estoque está muito alto!");  
        } else {  
            System.out.println("Seu estoque está bom");  
        }  
    }  
}
```

Exercícios Estruturas de repetição

Estruturas de Repetição

Desenvolva um programa que calcule a soma dos números de 1 a 15. Utilize o comando de repetição (While).

Desenvolva um programa que leia 10 números do usuário e calcule a soma desses números. Utilize o comando de repetição Repita (Do...While)

Desenvolva um programa que leia 15 números do usuário. Ao final exiba a média dos 15 números. Utilize o comando de repetição Para (For).

Desenvolva um programa que leia o nome e a idade de 10 pessoas e exiba: o nome e a idade da pessoa mais nova.

Desenvolva um programa que leia a quantidade de funcionários em uma empresa e, para cada funcionário leia seu nome e seu tempo de serviço (em meses). Se o funcionário possuir mais de 12 meses na empresa, escreva a mensagem “<NOME> tem direito a férias”. Caso contrário, escreva a mensagem “<NOME> não tem direito a férias”. Ao final, exiba quantos funcionários possuem direito a férias e quantos não possuem.

Arrays

- Utilizamos Arrays para facilitar o uso do código de forma que não seja necessário criar várias variáveis.
 - Exemplo:

```
32 int nota1;
33 int nota2;
34 int nota3;
35
36 nota1 = 5;
37 nota2 = 6;
38 nota3 = 8;
```

- Pra isso criamos os Arrays, que foram herdadas das linguagens procedurais, porém temos atualmente estruturas ainda mais especializadas como os *collections*, que estudaremos futuramente.
- Os arrays em Java armazenam múltiplos itens de dado do mesmo tipo em um bloco contínuo de memória

- Esses blocos são indexados, sendo o primeiro índice iniciado com 0 (zero)
- A declaração de Array pode ser feita através da concatenação de um par de colchetes “[]” após o tipo dos elementos do Array ou após o identificador nome do Array:

```
41 int[] notas;  
42 ou  
43 int notas[];
```

- Após declarado, o Array deve ser criado através do operador new, sendo seu tamanho passado como parâmetro no construtor

- Exemplo de declaração e construção de um array para armazena cem notas:

```
5 //declaracao do array
6 int notas[];
7 //construcao do array
8 notas = new int[100];
9
10 //ou declaracao e construcao na mesma linha
11 int notas[] = new int[100];
12
13
```

- Obs: o Array é um objeto, já que utilizamos o operador new
- Um Array também pode ser construído ao ser inicializado com dados:

```
53
54 int notas[] = {5, 6, 7, 8, 9};
55

56
57 boolean resultados[] = { true, false, true, false };
58
59 String dias[] = {"Seg", "Ter", "Qua", "Qui", "Sex", "Sab", "Dom"};
60
61
```

- Para acessar um elemento do array devemos utilizar o número de seu índice
- Os índices de um array vão de 0 até o número de elementos-1
- O código abaixo imprime a 4^a nota armazenada no Array

```
61  
62 System.out.println(notas[3]);  
63
```

- Para tipos numéricos, o Array é inicializado com zero
- Arrays de objetos devem ser inicializados explicitamente
- Imprimir todos os elementos de um Array:

```
public class ArraySample{  
    public static void main( String[] args ) {  
        int[] ages = new int[100];  
        for( int i=0; i<100; i++ ){  
            System.out.print( ages[i] );  
        }  
    }  
}
```

- Quando não se sabe o tamanho de um array podemos utilizar o atributo `length` de um array

```
public class ArraySample {  
    public static void main( String[] args ){  
        int[] ages = new int[100];  
        for( int i=0; i < ages.length; i++ ){  
            System.out.print( ages[i] );  
        }  
    }  
}
```

- Arrays multidimensionais são implementados como arrays dentro de arrays, inserindo-se mais um par de colchetes para cada dimensão

```
// array inteiro de 512 x 128 elementos
int[][] twoD = new int[512][128];

// array de caracteres de 8 x 16 x 24
char[][][] threeD = new char[8][16][24];

// array de String de 4 linhas x 2 colunas
String[][] dogs = {{ "terry", "brown" },
                   { "Kristin", "white" },
                   { "toby", "gray" },
                   { "fido", "black" }
};
```

- O acesso é feito nos mesmos moldes do unidimensional, mas passando quantos valores forem a dimensão do array, para duas dimensões, por exemplo, passando a linha e a coluna
- Assim, para acessarmos a primeira linha segunda coluna devemos programar da seguinte forma

```
64
65 nomeDoArray[0][1];
66
67 |-----|
```

Exercícios Array

TRABALHANDO COM **CONTINUE** E **BREAK**

Você pode utilizar a palavra-chave `continue` para pular uma iteração de seu looping e forçar a execução do próximo laço. O código a seguir vai imprimir todos os números de 0 a 10, mas vai pular o número 7:

```
for(int i = 0; i <= 10; i++) {  
    if (i == 7) {  
        continue;  
    }  
    System.out.println(i);  
}
```

Outra possibilidade comum é parar a execução de um looping dada uma determinada condição. Para isso, utilizamos a palavra-chave `break`:

```
for(int i = 0; i <= 10; i++) {  
    if (i == 7) {  
        break;  
    }  
    System.out.println(i);  
}
```

Neste caso, apenas os números de 0 a 6 serão impressos.

Break

- Uma declaração break pode ser não rotulada ou rotulada (em inglês, unlabeled e labeled break) ;
- O unlabeled break (não rotulado) encerra a estrutura de repetição na qual está inserido, passando o fluxo de controle para a próxima instrução ;
- O labeled break (rotulado) encerra o laço exterior identificado pelo label (rótulo) ;
- O fluxo de controle passa para a próxima instrução após a que está o label .

Exemplo Break

```
String names[] = {"Beah", "Bianca", "Lance", "Belle", "Nico", "Yza",
    "Gem", "Ethan"};
String searchName = "Yza";
boolean foundName = false;
for (int i=0; i < names.length; i++) {
    if (names[i].equals(searchName)) {
        foundName = true;
        break;
    }
}
if (foundName)
    System.out.println(searchName + " found!");
else
    System.out.println(searchName + " not found.");
```

Continue

- Da mesma forma que o break, o continue pode ser rotulado ou não rotulado (labeled ou unlabeled continue)
- O continue não rotulado, ao ser encontrado no código, interrompe o fluxo e retorna ao início do seu laço para avaliar a expressão do próximo fluxo, ignorando o restante do código deste laço
- O continue rotulado, ao ser encontrado no código, interrompe o fluxo e retorna ao início do laço indicado pelo rótulo para avaliar a expressão do próximo fluxo

Exemplo Continue

```
String names[] = {"Beah", "Bianca", "Lance", "Beah"};
int count = 0;
for (int i=0; i<names.length; i++) {
    if (!names[i].equals("Beah")) {
        continue; // interrompe o fluxo
    }
    count++;
}
System.out.println(count + " Beahs in the list");
```

CAIXAS DE DIÁLOGO COM JOPTIONPANE

- Os conceitos estruturais principais de linguagem, a idéia de orientação a objeto e as classes que mais utilizaremos como [String](#) e [Numbers](#), passaremos agora a focar numa programação mais voltada a práticas possíveis, demonstrando classes com funções mais avançadas e importantes, além de abandornamos um pouco o visual de console e partirmos para a programação visual de Java.
- Para introduzir esta nova fase, iniciaremos descrevendo um pouco sobre **JOptionPane** do pacote visual **Swing**.
- A classe JOptionPane nos proporciona uma série de métodos estáticos que ao serem invocados criam caixas de diálogos simples e objetivas.
- Para usar JOptionPane temos sempre que importar o pacote **javax.swing.JOptionPane** primeiro.

CAIXAS DE DIÁLOGO - *OPTIONPANE* - *Input Text*

- As caixas de diálogo de entrada de texto ou Input Text Dialog servem para fazer uma requisição de algum dado ao usuário de forma bem simples e direta.
- O que é digitado pelo usuário é retornado pelo método em forma de string.
- Existem mais de 10 métodos sobrecarregados para invocar uma caixa de diálogo Input Text, mas, a princípio, usaremos a mais simples.
- O método showInputDialog recebe um argumento que é a string contendo a informação desejada, o que na maioria das vezes é uma pergunta ou pedido.

CAIXAS DE DIÁLOGO - *OPTIONPANE* - *Input Text*

Para usarmos as caixas de diálogo precisamos importar uma classe do pacote (package) javax.swing, que é um pacote que usaremos bastante em nossas aplicações gráficas Java.

Essa classe é estática e se chama JOptionPane.

Se acostume com esses nomes começados com J: JLabel, JButton, JTextField...

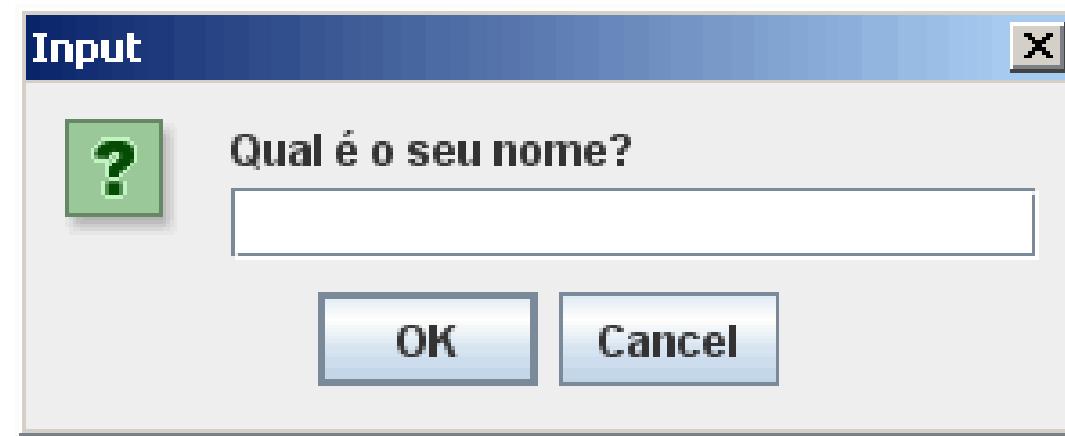
A importação, portanto, é:

import javax.swing.JOptionPane;

O exemplo abaixo demonstra um programa pedindo para que digite seu nome.

```
1. import javax.swing.JOptionPane;
2.
3. public class CaixasDeDialogo {
4.     public static void main(String[] args) {
5.         JOptionPane.showInputDialog("Qual é o seu nome?");
6.     }
7. }
```

Será apresentada uma janela a seguir.



PRIMEIRA JANELA DE DIÁLOGO

```
import javax.swing.JOptionPane; // Importa a Classe JOptionPane

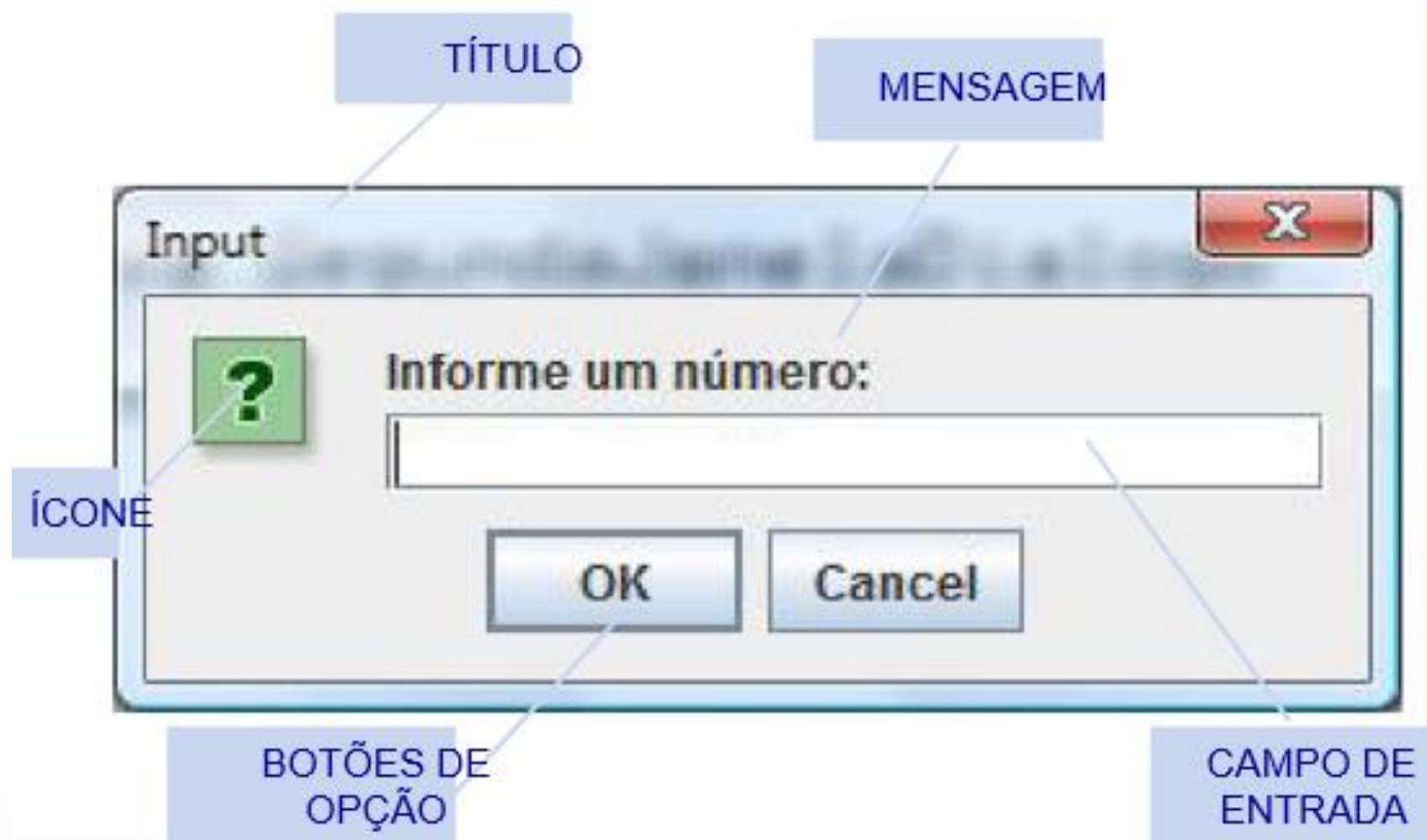
public class PrimeiraJanelaDialogo {
    public static void main(String [] args) {
        // Mostra uma mensagem na tela
        // Primeiro parâmetro é sempre null
        // Segundo parâmetro é a mensagem que se deseja exibir

        JOptionPane.showMessageDialog (null, "Para continuar clique
no botão \"OK\"!");

        System.exit(0); // Termina o programa
    }
}
```

A Classe *System* faz parte do pacote *java.lang*, que é automaticamente importado em todo programa Java.

APARÊNCIA DE UMA CAIXA DE DIÁLOGO



TIPOS DE CAIXA DE DIÁLOGO

| A Classe *JOptionPane* possui os seguintes métodos:

y **showConfirmDialog**

| Solicita uma confirmação do tipo SIM/NÃO/CANCELAR

y **showInputDialog**

| Solicita alguma entrada

y **showMessageDialog**

| Mostra algum aviso ao usuário

y **showOptionDialog**

| É uma unificação dos três métodos anteriores

Caixas de Diálogo Confirm

Outra caixa de diálogo simples e objetiva do JOptionPane é a caixa de diálogo de confirmação ou Confirm Dialog.

A Confirm Dialog (caixa de confirmação) consiste de uma caixa contendo uma mensagem, um ícone e três botões: sim, não e cancelar.

Apesar deste ser o aspecto padrão, esta caixa, como qualquer outra de JOptionPane, pode ser facilmente configurada (assunto que será tratado com mais detalhes nas próximas páginas).

No método padrão chamado showConfirmDialog usamos dois argumentos:

1.O primeiro é a dependência ou frame pai, ou seja, de qual janela esta confirmação está sendo gerada. Como nossos exemplos iniciais não possuem nenhuma dependência,

3. então, sempre usaremos null neste argumento.

2.O segundo argumento é a mensagem que desejamos mostrar ao usuário.

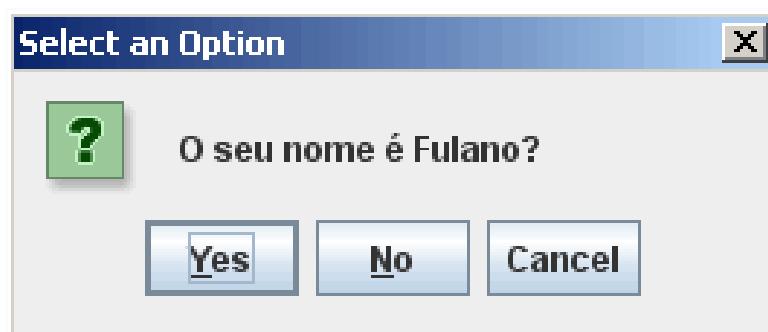
O método `showConfirmDialog` sempre retorna uma constante que é a resposta clicada pelo usuário, que são:

Valor	Nome da Constante	Equivale
0	YES_OPTION	ao clicar no botão Yes (sim)
1	NO_OPTION	ao clicar no botão No (não)
2	CANCEL_OPTION	ao clicar no botão Cancel (cancelar)

Melhorando o exemplo anterior ficaria assim.

```
01. import javax.swing.JOptionPane;
02.
03. public class CaixasDeDialogo {
04.     public static void main(String[] args) {
05.         String nome = null;
06.         nome = JOptionPane.showInputDialog("Qual é o seu nome?");
07.         JOptionPane.showConfirmDialog(null, "O seu nome é " + nome + "?");
08.     }
09. }
```

A caixa de confirmação pareceria da seguinte forma:



Caixa de Diálogo de Mensagem

A caixa de diálogo de mensagem é uma caixa que serve apenas para emitir uma mensagem.

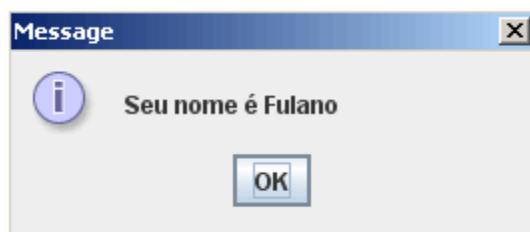
Esta caixa também é muito configurável e versátil, pois serve para muitas situações distintas como uma mensagem de erro, um alerta, ou simplesmente uma informação.

O método `showMessageDialog` é responsável em trazer a caixa de mensagem, o qual pode ter muitos argumentos, porém, vamos nos ater ao mais simples. Assim como o método `showConfirmDialog`, usaremos `null` como valor de nosso primeiro argumento, pois, por enquanto, não há dependência de outras janelas em nosso programa. O segundo argumento é a mensagem que desejamos emitir.

Para finalizar nosso exemplo, incluiremos as caixas de mensagem de acordo com as respostas.

```
01. import javax.swing.JOptionPane;
02.
03. public class CaixasDeDialogo {
04.     public static void main(String[] args) {
05.         String nome = null;
06.         int resposta;
07.         nome = JOptionPane.showInputDialog("Qual é o seu nome?");
08.         resposta = JOptionPane.showConfirmDialog(null, "O seu nome é " + nome + "?");
09.         if (resposta == JOptionPane.YES_OPTION) {
10.             // verifica se o usuário clicou no botão YES
11.             JOptionPane.showMessageDialog(null, "Seu nome é " + nome);
12.         } else {
13.             JOptionPane.showMessageDialog(null, "Seu nome não é " + nome);
14.         }
15.     }
16. }
```

Abaixo está um exemplo de como irá ficar a caixa de mensagem caso clique no botão YES.



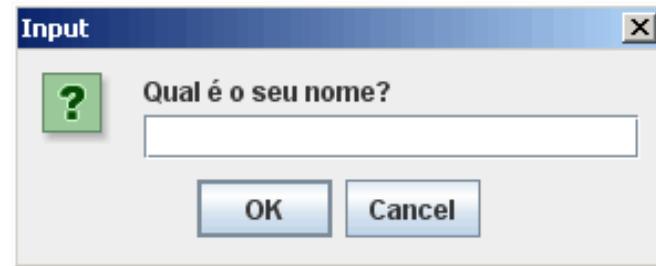
Caixas de Entrada de Dados

Caixas de diálogo de entrada de dados são importantes para obter informações ou requisitar algo do usuário.

No objeto `JOptionPane`, há o método `showInputDialog()` que é responsável em criar uma caixa de diálogo requisitando uma entrada de dado. Este método é sobre carregado de várias maneiras. A forma mais simples de seus argumentos é:

1. A mensagem que deve ser exibida para o usuário.

Com apenas este argumento é possível criar uma caixa de diálogo com o título *Input*, um ícone de interrogação, uma caixa de texto, uma mensagem e dois botões. Igual a figura abaixo:



Caixas de Entrada de Dados

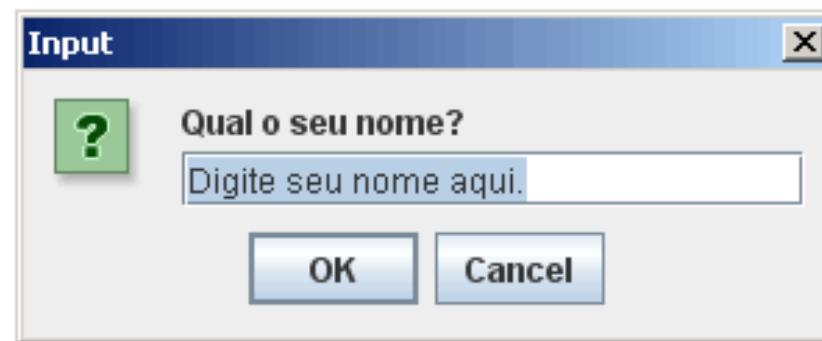
Porém, há como alterarmos a aparência dessa caixa, costumizando-a completamente. Outra forma é utilizar dois argumentos, sendo:

- 1.A mensagem que deve ser exibida ao usuário.
- 2.O valor inicial da caixa de texto.

O valor inicial da caixa de texto é a string que deve aparecer dentro do campo onde digitamos nossa entrada. Assim que aparece a caixa, seu campo está preenchido com um valor inicial já selecionado.

Ex.:

O código `JOptionPane.showInputDialog("Qual o seu nome?", "Digite seu nome aqui.")` geraria a seguinte caixa:



Caixas de Entrada de Dados

Uma das formas mais completas desse método inclui alterar, inclusive, o título da caixa. Assim, usa-se 4 argumentos:

1. De qual frame a caixa de diálogo é dependente, ou seja, qual a janela principal que chamou a caixa Input Dialog. Caso a caixa de diálogo não dependa de nenhum frame ou janela principal, basta utilizarmos o valor null para este argumento.
2. A mensagem que deve ser exibida ao usuário.
3. O título da caixa de texto.
4. Que tipo de mensagem é. O tipo de mensagem define qual o ícone será utilizado, podendo ser utilizados os números inteiros representados pelas constantes:

- PLAIN_MESSAGE (valor: -1): Mensagem limpa, sem nenhum ícone.
- ERROR_MESSAGE (valor: 0): Mensagem de erro.
- INFORMATION_MESSAGE (valor: 1): Mensagem informativa.
- WARNING_MESSAGE (valor: 2): Mensagem de alerta.
- QUESTION_MESSAGE (valor: 3): Mensagem de requisição ou pergunta. Esta é a opção padrão do método showInputDialog().

Caixas de Entrada de Dados

•

1. De qual frame a caixa de diálogo é dependente, ou seja, qual a janela principal que chamou a caixa Input Dialog. Caso a caixa de diálogo não dependa de nenhum frame ou janela principal, basta utilizarmos o valor null para este argumento.
2. A mensagem que deve ser exibida ao usuário.
3. O título da caixa de texto.
4. Que tipo de mensagem é. O tipo de mensagem define qual o ícone será utilizado, podendo ser utilizados os números inteiros representados pelas constantes:

- PLAIN_MESSAGE (valor: -1): Mensagem limpa, sem nenhum ícone.
- ERROR_MESSAGE (valor: 0): Mensagem de erro.
- INFORMATION_MESSAGE (valor: 1): Mensagem informativa.
- WARNING_MESSAGE (valor: 2): Mensagem de alerta.
- QUESTION_MESSAGE (valor: 3): Mensagem de requisição ou pergunta. Esta é a opção padrão do método showInputDialog().

Caixas de Entrada de Dados

Ex.: O código `JOptionPane.showInputDialog(null, "Qual o seu Nome?", "Pergunta", JOptionPane.PLAIN_MESSAGE)`geraria a seguinte caixa:



Obter valor de showInputDialog

O método `showInputDialog` pode retornar dois valores: ou uma string ou null.

Se o botão OK for clicado a string contida na caixa de texto será retornada, se o botão Cancel for clicado o valor null será retornado. Sabendo disso, podemos usar uma variável string para obter o valor e tratarmos da forma que quisermos. Vejamos o exemplo abaixo:

```
01. import javax.swing.JOptionPane;
02.
03. public class CaixasDeInput {
04.     public static void main(String[] args) {
05.         String nome = null;
06.         while (nome == null || nome.equals("")) {
07.             nome = JOptionPane.showInputDialog("Qual o seu nome?");
08.             if (nome == null || nome.equals("")) {
09.                 JOptionPane.showMessageDialog(null,
10.                     "Você não respondeu a pergunta.");
11.             }
12.         }
13.         JOptionPane.showMessageDialog(null, "Seu nome é " + nome);
14.     }
15. }
```

Input Dialog com lista de opções

Outra forma de caixa de diálogo de entrada de dados é a Input Dialog com lista de opções.

É o mesmo método `showInputDialog`, mas com mais argumentos, sendo um deles uma lista de objetos. Esta lista de objetos fará com que a caixa de diálogo venha com um *combo box* ao invés de um campo de texto.

Para criar um Input Dialog com um *combo box* devemos usar os seguintes argumentos na respectiva ordem:

1. De qual frame a caixa de diálogo é dependente, ou seja, qual a janela principal que chamou a caixa Input Dialog. Caso a caixa de diálogo não dependa de nenhum frame ou janela principal, basta utilizarmos o valor *null* para este argumento.
2. A mensagem que deve ser exibida ao usuário.
3. O título da caixa de texto.
4. Que tipo de mensagem é. O tipo de mensagem define qual o ícone será utilizado, podendo ser utilizados os números inteiros representados pelas constantes da mesma forma como foi mostrada anteriormente.
5. O quinto argumento é representado pelo objeto `Icon`, que é um ícone que podemos criar a partir de um jpg, gif, png, etc. O objeto `Icon` será comentado com mais detalhes nos próximos artigos.
6. O segredo do *combo box* está neste argumento. Aqui virá um [array \(vetor\)](#) de objetos que serão nossos valores pré-definidos.
7. O último argumento serve apenas para indicar qual elemento do array (vetor) deve vir selecionado no início. Caso não desejarmos que um ítem seja selecionado no início basta utilizarmos *null*.

O array (vetor) de objetos deve ser genérico, portanto, utilizamos a classe **Object** para criar este array.

O método `showInputDialog` com *combo box* se diferencia do `showInputDialog` com caixa de texto pelo seguinte fato: o que é retornado dessa vez não será uma string, mas um objeto. Isso faz sentido se percebermos que agora estamos escolhendo um item dentro de uma lista de objetos. Portanto, o que será retornado será um objeto dessa lista, não uma string como acontecia com o Input Dialog com caixa de texto.

Então, se quisermos utilizar o objeto genérico como algum outro tipo de dado, devemos antes fazer uma [indução de tipo ou typecasting](#).

Input Dialog com lista de opções

O array (vetor) de objetos deve ser genérico, portanto, utilizamos a classe **Object** para criar este array.

O método **showInputDialog** com *combo box* se diferencia do **showInputDialog** com caixa de texto pelo seguinte fato: o que é retornado dessa vez não será uma string, mas um objeto. Isso faz sentido se percebermos que agora estamos escolhendo um item dentro de uma lista de objetos. Portanto, o que será retornado será um objeto dessa lista, não uma string como acontecia com o Input Dialog com caixa de texto.

Então, se quisermos utilizar o objeto genérico como algum outro tipo de dado, devemos antes fazer uma [indução de tipo ou typecasting](#).

```
01. import javax.swing.JOptionPane;
02.
03. public class CaixaComComboBox {
04.     public static void main(String[] args) {
05.         Object[] opcoes = { "sim", "não" };
06.         Object resposta;
07.         do {
08.             resposta = JOptionPane.showInputDialog(null,
09.                                         "Deseja finalizar o programa?",
10.                                         "Finalização",
11.                                         JOptionPane.PLAIN_MESSAGE,
12.                                         null,
13.                                         opcoes,
14.                                         "não");
15.         } while (resposta == null || resposta.equals("não"));
16.     }
17. }
```

```
01. import javax.swing.JOptionPane;
02.
03. public class JogoDeAdivinhar {
04.     public static void main(String[] args) {
05.         // define um número qualquer entre 0 e 10
06.         int rndNr = (int) Math.ceil(Math.random() * 10);
07.         // lista de opções para o combo box da caixa de diálogo
08.         Object[] opcoes = { "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "10" };
09.         // string onde será retornado o resultado
10.         String resposta;
11.         while (true) {
12.             // loop para evitar que o jogo feche depois da primeira resposta
13.             resposta = (String) JOptionPane.showInputDialog(null,
14.                 "Em que número estou pensando?", "Jogo de Adivinhar",
15.                 JOptionPane.QUESTION_MESSAGE, null, opcoes, null);
16.             if (resposta == null) {
17.                 /*
18.                  * se clicar no botão Cancel, mostrar uma mensagem de Game Over
19.                  * e sair do loop para finalizar o programa
20.                 */
21.                 JOptionPane.showMessageDialog(null,
22.                     "Game Over!\nVocê desistiu do jogo!");
23.                 break;
24.             }
25.             if (Integer.valueOf(resposta) > rndNr) {
26.                 /*
27.                  * Interpreta string como inteiro e compara com o número sorteado
28.                  * para ver se é maior
29.                 */
30.                 JOptionPane.showMessageDialog(null,
31.                     "Errado!\nO número que eu pensei é menor.");
32.             } else if (Integer.valueOf(resposta) < rndNr) {
33.                 /*
34.                  * Interpreta string como inteiro e compara com o número sorteado
35.                  * para ver se é maior
36.                 */
37.                 JOptionPane.showMessageDialog(null,
38.                     "Errado!\nO número que eu pensei é maior.");
39.             } else {
40.                 /*
41.                  * se não for nem maior e nem menor, então é igual.
42.                  * Finaliza o jogo saindo do loop
43.                 */
44.                 JOptionPane.showMessageDialog(null,
45.                     "Parabéns\nVocê adivinhou!\n"
46.                     + "Eu realmente pensei no número " + rndNr);
47.                 break;
48.             }
49.         }
50.     }
51. }
```

Orientação a objetos

A linguagem Java tem como forte característica ter como paradigma a **orientação a objetos**. Esse paradigma existe desde a década de 70, mas foi depois do surgimento do Java que ficou bastante famoso e que passou a ser levado mais a sério.

Repare que nossa `CalculadoraDeEstoque` está fazendo todo o trabalho dentro de seu método `main`, ainda de forma muito procedural. A orientação a objetos propõe uma maneira diferente de fazer isso, você passa a trabalhar de um jeito mais próximo à realidade humana. Para cada necessidade importante teremos objetos que interagem entre si e que são compostos por estado (atributos) e comportamento (métodos). Quer um exemplo? Observe como estamos representando o preço de nossos livros:

```
double soma = 0;  
  
for (double i = 0; i < 35; i++) {  
    soma += 59.90;  
}
```

O valor 59.90 está fazendo isso. Ele representa o valor do livro; mas, e quanto ao seu nome, descrição e demais informações? Todas essas informações representam o que um livro tem e são extremamente importantes para nosso sistema. O grande problema do paradigma procedural é que não existe uma forma simples de conectar todos esses elementos, já na orientação a objetos podemos fazer isso de um jeito muito simples! Assim como no contexto real, podemos criar um objeto para representar tudo o que um livro tem e o que ele faz.

CRIANDO UM MOLDE DE LIVROS

Vamos criar uma nova classe Java chamada `Livro`. Essa classe será um molde, que representará o que um livro deve ter e como ele deve se comportar em nosso sistema.

```
public class Livro {  
}
```

Para deixar essa classe mais interessante, vamos adicionar alguns campos para representar o que um livro tem. Esses são os atributos de nossa classe:

```
public class Livro {  
  
    String nome;  
    String descricao;  
    double valor;  
    String isbn;  
}
```

Repare que esses atributos são muito parecidos com variáveis que podemos criar dentro de nosso método `main`, por exemplo. Mas eles não estão dentro de um bloco e, sim, dentro do escopo da classe, por isso recebem o nome diferenciado de atributo.

Nosso molde já está pronto para uso. Por enquanto, um livro terá um nome, descrição, valor e ISBN (um número de identificação, *International Standard Book Number*).

Esses campos não serão populados na classe `Livro`, ela é apenas o molde! O que precisamos é criar um objeto a partir desse molde. Para fazer isso, utilizamos a palavra-chave `new`.

```
Livro livro = new Livro();
```

Observe que a variável `livro` tem um tipo, assim como qualquer variável em Java, mas diferente de um `int`, `double` ou `String` como já estamos acostumados, agora seu tipo é a própria classe `Livro`.

Ao criar um objeto de `Livro` e atribuir a uma variável `livro`, estamos estabelecendo uma forma de nos referenciar a esse objeto que até então não tem nenhum valor populado.

Populando os atributos do livro

Agora que fizemos isso, a partir da variável `livro`, podemos acessar o objeto que foi criado em memória e popular os seus atributos. Um exemplo seria:

```
Livro livro = new Livro();
livro.nome = "Java 8 Prático";
livro.descricao = "Novos recursos da linguagem";
livro.valor = 59.90;
livro.isbn = "978-85-66250-46-6";
```

Repare que utilizamos um `.` (ponto) para acessar os atributos desse objeto em específico. Você também pode recuperar as informações adicionadas nos seus objetos acessando seus atributos da seguinte forma:

```
System.out.println("O nome do livro é " + livro.nome);
```

Neste caso, a saída será:

```
O nome do livro é Java 8 Prático
```

Vamos criar uma nova classe chamada `CadastroDeLivros` em nosso projeto. Ela deve ter um método `main` que cria um novo livro, preenche alguns de seus atributos e depois imprime os seus valores. Algo como:

```
public class CadastroDeLivros {  
  
    public static void main(String[] args) {  
  
        Livro livro = new Livro();  
        livro.nome = "Java 8 Prático";  
        livro.descricao = "Novos recursos da linguagem";  
        livro.valor = 59.90;  
        livro.isbn = "978-85-66250-46-6";  
  
        System.out.println(livro.nome);  
        System.out.println(livro.descricao);  
        System.out.println(livro.valor);  
        System.out.println(livro.isbn);  
    }  
}
```

Execute a classe para ver a saída! Deverá ser:

```
Java 8 Prático  
Novos recursos da linguagem  
59.9  
978-85-66250-46-6
```

CLASSE X OBJETO

Uma classe é apenas um molde. Uma especificação que define para a máquina virtual o que um objeto desse tipo deverá ter e como ele deve se comportar. Nossa livraria poderá ter milhares de livros (objetos), mas existirá apenas uma classe `Livro` (molde). Cada objeto que criarmos do tipo `Livro` terá seus próprios valores, ou seja, cada livro terá o seu próprio nome, sua descrição, um valor e um número de ISBN.

CRIANDO UM NOVO MÉTODO

A todo momento que criamos um novo objeto do tipo `Livro`, estamos imprimindo seus valores. Essa é uma necessidade comum entre todos os livros de nosso sistema, mas da forma como estamos fazendo, toda hora repetimos as mesmas 4 linhas de código:

```
System.out.println(livro.nome);  
System.out.println(livro.descricao);  
System.out.println(livro.valor);  
System.out.println(livro.isbn);
```

A única coisa que muda é o nome da variável, de `livro` para `outroLivro`.

Pense na seguinte situação, em breve nosso cadastro terá cerca de 100 livros. Para cada livro, teremos 4 linhas de código imprimindo os seus 4 atributos. Ou seja, teremos 400 linhas de código muito parecidas, praticamente repetidas.

Essa repetição de código sempre tem um efeito colateral desagradável, que é a dificuldade de manutenção. Quer ver? O que acontece se adicionarmos um novo atributo no livro, por exemplo a data de seu lançamento? Para imprimir a data toda vez que criar um livro, teremos que mudar 100 partes de nosso código, adicionando o `System.out.println(livro.dataDeLancamento)`.

No lugar de deixar essa lógica de impressão dos dados do livro toda espalhada, podemos isolar esse comportamento comum entre os livros na classe `Livro`! Para isso, criamos um método. Uma forma seria:

```
void mostrarDetalhes() {  
    System.out.println(nome);  
    System.out.println(descricao);  
    System.out.println(valor);  
    System.out.println(isbn);  
}
```

Esse método define um comportamento para classe `Livro`. Repare que a sintaxe de um método é um pouco diferente do que estamos acostumados, sua estrutura é:

```
tipoDeRetorno nomeDoComportamento() {  
  
    // código que será executado  
}
```

Nesse caso, não estamos retornando nada e, sim, apenas executando instruções dentro do método, portanto, seu tipo de retorno é `void`. Essa palavra reservada indica que um método não tem retorno.

Um método também pode ter variáveis declaradas, como por exemplo:

```
void mostrarDetalhes() {  
    String mensagem = "Mostrando detalhes do livro ";  
    System.out.println(mensagem);  
    System.out.println(nome);  
    System.out.println(descricao);  
    System.out.println(valor);  
    System.out.println(isbn);  
}
```

A variável `mensagem` foi declarada dentro do método, logo esse será seu escopo, ou seja, ela só existirá e poderá ser utilizada dentro do método `mostrarDetalhes`.

Nossa classe Livro ficou desta forma:

```
public class Livro {  
  
    String nome;  
    String descricao;  
  
    double valor;  
    String isbn;  
  
    void mostrarDetalhes() {  
        String mensagem = "Mostrando detalhes do livro ";  
        System.out.println(mensagem);  
        System.out.println(nome);  
        System.out.println(descricao);  
        System.out.println(valor);  
        System.out.println(isbn);  
    }  
}
```

Agora que cada livro possui o comportamento de exibir os seus detalhes, podemos remover as linhas que faziam esse trabalho no método `main` e passar a invocar esse novo método. Repare:

```
public class CadastroDeLivros {

    public static void main(String[] args) {

        Livro livro = new Livro();
        livro.nome = "Java 8 Prático";
        livro.descricao = "Novos recursos da linguagem";
        livro.valor = 59.90;
        livro.isbn = "978-85-66250-46-6";

        livro.mostrarDetalhes();

        Livro outroLivro = new Livro();
        outroLivro.nome = "Lógica de Programação";
        outroLivro.descricao = "Crie seus primeiros programas";
        outroLivro.valor = 59.90;
        outroLivro.isbn = "978-85-66250-22-0";

        outroLivro.mostrarDetalhes();
    }
}
```

Ao isolar esse comportamento dentro da classe `Livro` já tivemos um ganho evidente. Tivemos que escrever menos linhas em nosso `main`, o código ficou com menos repetições pois teve maior reaproveitamento. Mas essa não é a única vantagem em isolar um comportamento: mesmo que eu tenha 100 livros cadastrados e use o método `mostrarDetalhes` 100 vezes, em quantas partes do meu código eu terei que mexer para modificar a forma como os livros são exibidos? A resposta é: uma!

Agora que o comportamento está isolado, só precisaremos alterar o método, um único ponto do nosso código, para refletir a mudança em todos os lugares que o usam. Vamos colocar isso em prática, basta mudar o método `mostrarDetalhes` para:

```
void mostrarDetalhes() {  
    System.out.println("Mostrando detalhes do livro ");  
    System.out.println("Nome: " + nome);  
    System.out.println("Descrição: " + descricao);  
    System.out.println("Valor: " + valor);  
    System.out.println("ISBN: " + isbn);  
    System.out.println("--");  
}
```

Execute agora o `main` da classe `CadastroDeLivros`. Sem precisar mudar nenhuma linha de seu código, o resultado será:

```
Mostrando detalhes do livro
```

```
Nome: Java 8 Prático
```

```
Descrição: Novos recursos da linguagem
```

```
Valor: 59.9
```

```
ISBN: 978-85-66250-46-6
```

```
--
```

```
Mostrando detalhes do livro
```

```
Nome: Lógica de Programação
```

```
Descrição: Crie seus primeiros programas
```

```
Valor: 59.9
```

```
ISBN: 978-85-66250-22-0
```

```
--
```

Nosso código agora tem uma manutenibilidade muito maior! Sempre devemos criar métodos de forma genérica e reaproveitável, assim será muito mais fácil e produtivo evoluir o código no futuro.

OBJETOS PARA TODOS OS LADOS!

Falando em evoluir código, precisamos saber mais informações sobre nossos livros. Por exemplo, quem escreveu o livro? Qual o e-mail do autor? E quando foi a sua data de publicação? Todas essas informações são relevantes para nossa livraria e também para nossos clientes. Podemos adicionar essas e outras informações na classe Livro:

```
public class Livro {  
  
    String nome;  
    String descricao;  
    double valor;  
    String isbn;  
    String nomeDoAutor;  
    String emailDoAutor;  
    String cpfDoAutor;  
  
    void mostrarDetalhes() {  
        System.out.println("Mostrando detalhes do livro ");  
        System.out.println("Nome: " + nome);  
        System.out.println("Descrição: " + descricao);  
        System.out.println("Valor: " + valor);  
        System.out.println("ISBN: " + isbn);  
        System.out.println("---");  
    }  
}
```

Mas repare que todas essas novas informações pertencem ao autor do livro e não necessariamente ao livro. Se autor é um elemento importante para nosso sistema, ele pode e deve ser representado como um objeto! Vamos fazer essa alteração, basta criar a classe Autor e declarar seus atributos:

```
public class Autor {  
  
    String nome;  
    String email;  
    String cpf;  
}
```

Portanto, podemos adicionar na classe Livro um atributo do tipo Autor, que acabamos de criar. Uma classe pode ter outra classe como atributo, esse é um processo natural conhecido como composição. Nosso código fica assim:

```
public class Livro {  
  
    String nome;  
    String descricao;  
    double valor;  
    String isbn;  
    Autor autor;  
  
    void mostrarDetalhes() {  
        System.out.println("Mostrando detalhes do livro ");  
        System.out.println("Nome: " + nome);  
        System.out.println("Descrição: " + descricao);  
        System.out.println("Valor: " + valor);  
        System.out.println("ISBN: " + isbn);  
        System.out.println("--");  
    }  
}
```

Vamos agora criar alguns autores no `CadastroDeLivros` e associar ao seu devido livro. Uma forma de fazer isso seria:

```
Autor autor = new Autor();
autor.nome = "maria";
autor.email = "maria.pilar@gmail.com";
autor.cpf = "123.456.789.10";
```

Nesse código, apenas criamos o `autor` e populamos os seus atributos. Agora precisamos associar esse objeto ao seu livro. Podemos simplesmente fazer:

```
livro.autor = autor;
```

Vamos fazer isso com os dois livros, nosso código [completo](#) deve ficar parecido com:

Referência a objetos

É fundamental perceber que, quando instanciamos um novo objeto com a palavra reservada `new`, um `Autor` por exemplo, guardamos em sua variável uma referência para esse objeto, e não seus valores. Ou seja, a variável `autor` não guarda o valor de um `nome`, `email` e outros atributos da classe `Autor`, mas sim uma forma de acessar esses atributos do `autor` em memória. Muito diferente de quando trabalhamos com tipos primitivos que guardam uma cópia do valor.

NOMES AMBÍGUOS E O `THIS`

O que aconteceria se o nome do parâmetro do método fosse igual ao nome do atributo da classe? Veja como ficaria o nosso código:

```
public void aplicaDescontoDe(double valor) {  
    valor -= valor * valor;  
}
```

No lugar de uma `porcentagem`, chamamos o parâmetro de `valor`, assim como o nome do atributo da classe `Livro`. Como o Java vai saber qual `valor` queremos atualizar? A resposta é: ele não vai. Nossa código vai subtrair e multiplicar o `valor` do parâmetro por ele mesmo, já que este tem um escopo menor que o atributo da classe. Ou seja, mesmo com a ambiguidade neste caso o código vai compilar, mas o `valor` considerado será o que possui o menor escopo.

Para evitar esse problema, podemos utilizar a palavra reservada `this` para mostrar que esse é um `atributo da classe`. Ainda que seja opcional, é sempre uma boa prática usar o `this` em atributos para evitar futuros problemas de ambiguidade e também para deixar claro que este é um atributo da classe, e não uma simples variável.

Com isso, o código de nosso método `aplicaDescontoDe` fica assim:

```
public void aplicaDescontoDe(double porcentagem) {  
    this.valor -= this.valor * porcentagem;  
}
```

E a classe `Livro` completa:

```
public class Livro {  
  
    String nome;  
    String descricao;  
    double valor;  
    String isbn;  
    Autor autor;  
  
    void mostrarDetalhes() {  
        System.out.println("Mostrando detalhes do livro ");  
        System.out.println("Nome: " + nome);  
        System.out.println("Descrição: " + descricao);  
        System.out.println("Valor: " + valor);  
        System.out.println("ISBN: " + isbn);  
        autor.mostrarDetalhes();  
        System.out.println("--");  
    }  
  
    public void aplicaDescontoDe(double porcentagem) {  
        this.valor -= this.valor * porcentagem;  
    }  
}
```

Métodos com retorno

Nossa classe `Livro` já possui dois métodos, mas os dois são `void`. Vimos que o `void` representa a ausência de um retorno, mas há situações em que precisaremos retornar algo. Por exemplo, como saber se um livro tem ou não um autor? Um alternativa seria criar um método `temAutor`, tendo um `boolean` como retorno. Observe um exemplo de uso:

```
if(livro.temAutor()) {  
    System.out.print("O nome do autor desse livro é ");  
    System.out.println(livro.autor.nome);  
}
```

A implementação desse método é bem simples, ele não receberá nenhum parâmetro e terá o tipo de retorno `boolean`:

```
boolean temAutor(){  
    // o que fazer aqui?  
}
```

Mas como saber se o `autor` do livro existe ou não? Na verdade, é bem simples: quando um objeto não foi instanciado, ele não tem nenhuma referência, portanto seu valor será `null`. Sabendo disso, tudo o que precisamos

fazer no corpo do método é retornar um boolean, informando se o atributo autor é diferente de null. Para retornar um valor, utilizamos a palavra reservada `return`:

```
boolean temAutor(){  
    boolean naoEhNull = this.autor != null;  
    return naoEhNull;  
}
```

Podemos fazer isso de uma forma ainda mais simples, retornando diretamente a expressão booleana sem a criação da variável `naoEhNull`:

```
boolean temAutor(){  
    return this.autor != null;  
}
```

Agora que o Livro ganhou esse novo comportamento, vamos tirar proveito dele dentro do método `mostrarDetalhes` da própria classe. Podemos mostrar os detalhes do autor apenas quando ele existir. Uma forma de fazer isso seria:

```
void mostrarDetalhes() {  
    System.out.println("Mostrando detalhes do livro ");  
    System.out.println("Nome: " + nome);  
    System.out.println("Descrição: " + descricao);  
    System.out.println("Valor: " + valor);  
    System.out.println("ISBN: " + isbn);  
  
    if (this.temAutor()) {  
        autor.mostrarDetalhes();  
    }  
  
    System.out.println("---");  
}
```

Pronto! Ao imprimir os detalhes de um livro que não tem autor, apenas os dados do livro serão exibidos.

ENTENDENDO A CONSTRUÇÃO DE UM OBJETO

Agora que já conhecemos um pouco sobre objetos, podemos entender melhor como funciona seu processo de construção. Repare na sintaxe utilizada para criar um novo `Livro`:

```
Livro livro = new Livro();
```

Quando escrevemos a instrução `Livro()` seguida da palavra reservada `new`, estamos pedindo para a JVM procurar a classe `Livro` e invocar o seu construtor, que se parece com:

```
public Livro(){  
}
```

Um construtor é bastante parecido com um método comum, mas ele não é um. Diferente dos métodos, um construtor tem o mesmo nome da classe e não tem um retorno declarado.

Mas, se nunca escrevemos esse construtor, quem o fez? Sempre que você não criar um construtor para suas classes, o compilador fará isso para você.

Quer uma prova? Vamos utilizar o programa `javap` que já vem no JDK para ver o código compilado. Acesse pelo terminal do seu sistema operacional a pasta `bin` de seu projeto e rode o comando `javap Livro`. A saída deverá se parecer com:

```
public class Livro {  
    java.lang.String nome;  
    java.lang.String descricao;  
    double valor;  
    java.lang.String isbn;  
    public Livro();  
    void mostrarDetalhes();  
    public void aplicaDescontoDe(double);  
}
```

Note que, mesmo sem criar o construtor vazio, ele está presente em nosso código compilado:

```
public Livro();
```

E, sim, todo código que estiver declarado dentro do construtor será executado quando um objeto desse tipo for criado. Por exemplo, o código a seguir imprime uma mensagem sempre que criarmos um novo Livro:

```
public class Livro {  
  
    String nome;  
    String descricao;  
    double valor;  
    String isbn;  
    Autor autor;  
  
    public Livro() {  
        System.out.println("novo livro criado");  
    }  
  
    // demais métodos da classe  
}
```

Para testar, podemos criar alguns livros dentro de um método `main` e executar esse código:

Para testar, podemos criar alguns livros dentro de um método `main` e executar esse código:

```
public static void main(String[] args) {  
    Livro livro1 = new Livro();  
    Livro livro2 = new Livro();  
    Livro livro3 = new Livro();  
    Livro livro4 = new Livro();  
}
```

Como já é esperado, a saída será:

```
novo livro criado  
novo livro criado  
novo livro criado  
novo livro criado
```

Agora que já temos um construtor, o compilador não vai criar mais nenhum. Ele só faz isso quando a sua classe não tem nenhum construtor definido.

Veremos mais adiante que construtores também podem receber parâmetros e inicializar os atributos de suas classes.

ENCAPSULAMENTO

LIMITANDO DESCONTO DO LIVRO

Quando criamos o método `aplicaDescontoDe` na classe `Livro`, não determinamos a porcentagem total de desconto que um livro pode ter, mas isso é muito importante para nossa regra de negócio. Um livro pode ter no máximo 30% de desconto.

Uma forma simples de evoluir nosso método seria:

```
public boolean aplicaDescontoDe(double porcentagem) {  
    if (porcentagem > 0.3) {  
        return false;  
    }  
    this.valor -= this.valor * porcentagem;  
    return true;  
}
```

Repare que agora estamos retornando um boolean para indicar ao usuário se o desconto foi aplicado ou não. No momento de usá-lo podemos fazer algo como:

```
if (!livro.aplicaDescontoDe(0.1)){  
    System.out.println("Desconto não pode ser maior do que 30%");  
}
```

CÓDIGO FLEXÍVEL E REAPROVEITÁVEL

O método `aplicaDescontoDe` poderia, sim, no lugar de retornar um boolean já imprimir a mensagem de validação do limite de desconto, como por exemplo:

```
public void aplicaDescontoDe(double porcentagem) {
    if (porcentagem > 0.3) {
        System.out.println("Desconto não pode ser
                           maior do que 30%");
    }
    this.valor -= this.valor * porcentagem;
}
```

Com isso, não seria necessário escrever o `if` em nosso método `main`, mas repare que estamos perdendo a flexibilidade de customizar nossas mensagens. Será que todos que forem utilizar esse método querem mostrar a mensagem *Desconto não pode ser maior do que 30%*? E se eu quiser mudar esse texto dependendo do usuário ou simplesmente não mostrar a mensagem de validação em algum momento?

Lembre-se sempre de evitar deixar informações tão específicas em seus moldes, além de não sobrecarregá-los com comportamentos que não deveriam ser de sua responsabilidade.

Pronto! Nossa lógica já está bem definida, agora um livro não pode ter um desconto maior do que 30%. Mas vamos testá-la para ter certeza. Para isso, basta criar a classe `RegrasDeDesconto` com o seguinte cenário:

```
public class RegrasDeDesconto {  
    public static void main(String[] args) {  
  
        Livro livro = new Livro();  
        livro.valor = 59.90;  
  
        System.out.println("Valor atual: " + livro.valor);  
  
        if (!livro.aplicaDescontoDe(0.1)){  
            System.out.println("Desconto não pode ser  
                maior do que 30%");  
        } else {  
            System.out.println("Valor com desconto: "  
                + livro.valor);  
        }  
    }  
}
```

Ao executá-la, teremos o resultado:

Valor atual: 59.9

Valor com desconto: 53.91

E, mudando o valor do desconto para 0.4, a saída será:

Valor atual: 59.9

Desconto não pode ser maior do que 30%

Excelente! É exatamente o comportamento que estamos esperando. Mas há um problema grave. Nada obriga o desenvolvedor a utilizar o método `aplicaDescontoDe`, ele poderia perfeitamente escrever o código desta forma:

```
public class RegrasDeDesconto {  
  
    public static void main(String[] args) {  
  
        Livro livro = new Livro();  
        livro.valor = 59.90;  
  
        System.out.println("Valor atual: " + livro.valor);  
  
        livro.valor -= livro.valor * 0.4;  
        System.out.println("Valor com desconto: " + livro.valor);  
    }  
}
```

Ele consegue aplicar o desconto de 40% ou mais, independente de nossa regra de negócio.

O problema é que o atributo `valor` da classe `Livro` pode ser acessado diretamente e modificado pelas outras lógicas do nosso sistema. Isso é muito ruim! Como vimos, ao expor nossos dados, estamos abrindo caminho para que os objetos possam ser modificados independente das condições verificadas em seus métodos.

Apesar de ser um problema muito sério, a solução é bastante simples. Basta modificar a visibilidade do atributo `valor`, que até então é `default` (padrão). Podemos restringir o acesso para esse atributo para que fique acessível apenas pela própria classe `Livro`. Repare:

```
public class Livro {  
  
    private double valor;  
  
    // outros atributos e métodos  
}
```

Como estamos utilizando o modificador de visibilidade `private`, ninguém mais além da própria classe `Livro` conseguirá acessar e modificar esse valor. Portanto, a seguinte linha da classe `RegrasDeDesconto` não irá compilar.

```
livro.valor -= livro.valor * 0.4;
```

O erro será `The field Livro.valor is not visible`. Ótimo, isso fará com que a única forma de se aplicar um desconto em um livro seja passando pela nossa regra de negócio, que está isolada no método `aplicaDescontoDe`.

MODIFICADORES DE VISIBILIDADE

Além do `private` e `default` (padrão), existem outros modificadores de visibilidade. São eles: `public` e `protected`. No momento certo entenderemos a fundo cada um deles, mas tenha em mente desde já que todos eles são diferentes e possuem regras bem específicas.

ISOLANDO COMPORTAMENTOS

Ao alterar a visibilidade do atributo `valor` para `private` todos os lugares que acessavam esse atributo passaram a não compilar, já que apenas a própria classe `Livro` pode fazer isso. Por exemplo, observe como estávamos passando o valor do livro:

```
livro.valor = 59.90;
```

Como esse acesso não pode mais ser feito, precisaremos criar um comportamento (método) na classe `Livro` para fazer a atribuição desse valor. Poderíamos chamá-lo de `adicionaValor`.

```
livro.adicionaValor(59.90);
```

Sua implementação será bem simples, observe:

```
void adicionaValor(double valor) {  
    this.valor = valor;  
}
```

Há outro erro de compilação que precisaremos resolver. Repare na forma como estamos imprimindo o valor do Livro:

```
System.out.println("Valor atual: " + livro.valor);
```

Bem, você já pode ter imaginado que a solução será parecida. Preciso criar um método retornaValor que não receberá nenhum argumento e retornará o atributo double valor. Algo como:

```
double retornaValor() {  
    return this.valor;  
}
```

E como esse método não recebe parâmetros, o uso do this será opcional.

Agora podemos imprimir dessa forma:

```
System.out.println("Valor atual: " + livro.retornaValor());
```

Quando usar o private?

Isso resolveu o problema do valor do livro, mas e quanto aos demais atributos? Quando devemos deixar um atributo de classe com visibilidade private? Você deve estar pensando: *“quando não quero que ninguém acesse esse atributo diretamente”*

, e é verdade. Mas a questão é: quando eu quero que alguém acesse algum atributo de forma direta?

O nome do livro está com visibilidade default, portanto estamos adicionando da seguinte forma:

```
livro.nome = "Java 8 Prático";
```

Considere que depois de alguns meses apareça a necessidade de validar que a `String` passada possui ao menos duas letras, caso contrário não será um nome válido. Ou seja, nosso código precisará fazer algo como:

```
livro.nome = "Java 8 Prático";
if(livro.nome.length() < 2) {
    System.out.println("Nome inválido");
}
```

Repare que o método `length` foi usado para retornar o tamanho de uma `String`. Além desse há diversos outros métodos úteis definidos na classe `String`, que conheceremos melhor adiante. O importante agora é perceber que, se acessamos o atributo `nome` diretamente em mil partes de nosso código, precisaremos replicar essa mudança em mil lugares. Nada fácil.

código, precisaremos replicar essa mudança em mil lugares. Nada fácil.

Já vimos que todo comportamento da classe `Livro` deveria ser isolado em um método, assim evitamos repetição de código possibilitando o reúso desse método por toda a aplicação. O ideal seria desde o começo ter criado um método `adicionaNome` com esse comportamento.

Mas apenas criar o método `adicionaNome` não teria resolvido o problema, percebe? Se a visibilidade do atributo ainda fosse `default`, haveria duas formas de se adicionar um nome ao livro e nossa lógica de adicionar nome ainda poderia estar espalhada pela aplicação.

Como garantir que isso nunca aconteça? Usando `private` sempre! Todo atributo de classe deve ser privado, assim garantimos que ninguém os acesse diretamente e viole as nossas regras de negócio.

CÓDIGO ENCAPSULADO

Essa mudança que iniciamos em nossa classe faz parte de um conceito muito importante da OO conhecido como **encapsulamento**. A ideia é simplesmente esconder todos os atributos de suas classes (deixando-os `private`) e *encapsular* seus comportamentos em métodos. Além disso, encapsular é esconder como funcionam suas regras de negócio, os seus métodos.

Um bom termômetro para descobrir se o seu código está bem encapsulado é fazendo as duas perguntas:

- O que esse código faz?
- Como esse código faz?

Veja por exemplo quando aplicamos o desconto acessando o atributo diretamente:

```
livro.valor -= livro.valor * 0.4;
```

Olhando para esse código, nós conseguimos responder as duas perguntas: o que e como é feito, mas em um código bem encapsulado você só deveria conseguir responder a primeira.

Observe este segundo exemplo:

```
livro.aplicaDescontoDe(0.1);
```

Repare que fica claro o que o método `aplicaDescontoDe` está fazendo, mas apenas olhando para ele não conseguimos responder **como** isto será feito. Se em algum momento no futuro resolvermos mandar um e-mail sempre que algum desconto for aplicado, não precisaremos mudar milhares de partes de nosso código, mas sim esse único método que está bem encapsulado. Portanto, o encapsulamento deixa nosso código muito mais passível de mudanças.

INTERFACE DA CLASSE

Os métodos públicos existentes em sua classe são comumente chamados de *interface da classe*, afinal em um código encapsulado eles são a única maneira de você interagir com os objetos dessa classe.

Pensar em algum contexto do mundo real pode ajudar a entender encapsulamento e interface da classe. Por exemplo, quando você vai escrever um texto em um computador, o que importa é o teclado (a interface que você usa para fazer isso). Pouco importa **como** ele funciona internamente. Quando você mudar de computador ou simplesmente de tipo de teclado, não precisará reaprender a usar um teclado.

Por mais diferente que eles funcionem internamente, isso não fará diferença para um usuário final. Ninguém precisa desmontar a lateral de um computador e manualmente mudar os circuitos para que o computador possa interpretar os sinais enviados e mostrar os dados na tela, afinal esse comportamento está muito bem encapsulado na interface do teclado.

GETTERS E SETTERS

Se todos os atributos de nossas classes forem `private`, precisaremos criar um método sempre que quisermos que alguém consiga adicionar um valor ao atributo e o mesmo quando quisermos que alguém consiga ler e exibir este valor. Assim como fizemos com os métodos `adicionaValor` e `retornaValor`.

A convenção de nome para esses dois métodos é utilizar o prefixo `get`

(pegar) e `set` (atribuir). Por exemplo, no lugar de chamar os métodos de `adicionaValor` e `retornaValor`, podemos chamá-los de `setValor` e `getValor`, respectivamente.

Nosso código fica assim:

```
public class Livro {  
  
    private double valor;  
  
    // demais atributos e métodos  
  
    public double getValor() {  
        return valor;  
    }  
  
    public void setValor(double valor) {  
        this.valor = valor;  
    }  
}
```

Os getters e setters possibilitem o acesso dos atributos de forma controlada. Mas é muito importante perceber que nem todo atributo deve ter um getter e setter, você só deve criar esses métodos quando houver uma real necessidade. Há um post do Paulo Silveira muito interessante ilustrando essa situação:

<http://blog.caelum.com.br/2006/09/14/nao-aprender-oo-getters-e-setters/>

Depois de criar os getters e setters necessários, nossa classe Livro deve ficar assim:

```
public class Livro {  
  
    private String nome;  
    private String descricao;  
    private double valor;  
    private String isbn;  
    private Autor autor;  
  
    // demais métodos omitidos
```

```
public double getValor() {  
    return valor;  
}  
  
public void setValor(double valor) {  
    this.valor = valor;  
}  
  
public String getNome() {  
    return nome;  
}  
  
public void setNome(String nome) {  
    this.nome = nome;  
}  
  
public String getDescricao() {  
    return descricao;  
}
```

```
public void setDescricao(String descricao) {  
    this.descricao = descricao;  
}  
  
public String getIsbn() {  
    return isbn;  
}  
  
public void setIsbn(String isbn) {  
    this.isbn = isbn;  
}  
  
public Autor getAutor() {  
    return autor;  
}  
  
public void setAutor(Autor autor) {  
    this.autor = autor;  
}  
}
```

Precisamos fazer o mesmo para nossa classe Autor:

```
public class Autor {  
  
    private String nome;  
    private String email;  
    private String cpf;  
  
    // demais métodos omitidos  
  
    public String getNome() {  
        return nome;  
    }  
  
    public void setNome(String nome) {  
        this.nome = nome;  
    }  
}
```

```
public String getEmail() {
    return email;
}

public void setEmail(String email) {
    this.email = email;
}

public String getCpf() {
    return cpf;
}

public void setCpf(String cpf) {
    this.cpf = cpf;
}

}
```

E, por fim, será necessário modificar a classe `CadastroDeLivros` para que deixe de acessar os atributos diretamente. Seu código final deve ficar parecido com:

```
public class CadastroDeLivros {  
  
    public static void main(String[] args) {  
  
        Autor autor = new Autor();  
        autor.setNome("Rodrigo Turini");  
        autor.setEmail("rodrigo.turini@caelum.com.br");  
        autor.setCpf("123.456.789.10");  
  
        Livro livro = new Livro();  
        livro.setNome("Java 8 Prático");  
        livro.setDescricao("Novos recursos da linguagem");  
        livro.setValor(59.90);  
        livro.setIsbn("978-85-66250-46-6");  
  
        livro.setAutor(autor);
```

```
livro.mostrarDetalhes();

Autor outroAutor = new Autor();
outroAutor.setNome("Paulo Silveira");
outroAutor.setEmail("paulo.silveira@caelum.com.br");
outroAutor.setCpf("123.456.789.10");

Livro outroLivro = new Livro();
outroLivro.setNome("Lógica de Programação");
outroLivro.setDescricao("Crie seus primeiros programas");
outroLivro.setValor(59.90);
outroLivro.setIsbn("978-85-66250-22-0");

outroLivro.setAutor(outroAutor);

outroLivro.mostrarDetalhes();
}

}
```

```
livro.mostrarDetalhes();

Autor outroAutor = new Autor();
outroAutor.setNome("Paulo Silveira");
outroAutor.setEmail("paulo.silveira@caelum.com.br");
outroAutor.setCpf("123.456.789.10");

Livro outroLivro = new Livro();
outroLivro.setNome("Lógica de Programação");
outroLivro.setDescricao("Crie seus primeiros programas");
outroLivro.setValor(59.90);
outroLivro.setIsbn("978-85-66250-22-0");

outroLivro.setAutor(outroAutor);

outroLivro.mostrarDetalhes();
}

}
```

HERANÇA



Herança

Enquanto programamos em Java, há a necessidade de trabalharmos com várias classes. Muitas vezes, classes diferentes tem características comuns, então, ao invés de criarmos uma nova classe com todas essas características usamos as características de um objeto ou classe já existente.

Ou seja, herança é, na verdade, uma classe derivada de outra classe.

Para simplificar de uma forma mais direta, vejamos:

Vamos imaginar que exista uma classe chamada Eletrodomestico, e nela estão definidos os seguintes atributos: ligado (boolean), voltagem (int) e consumo (int).

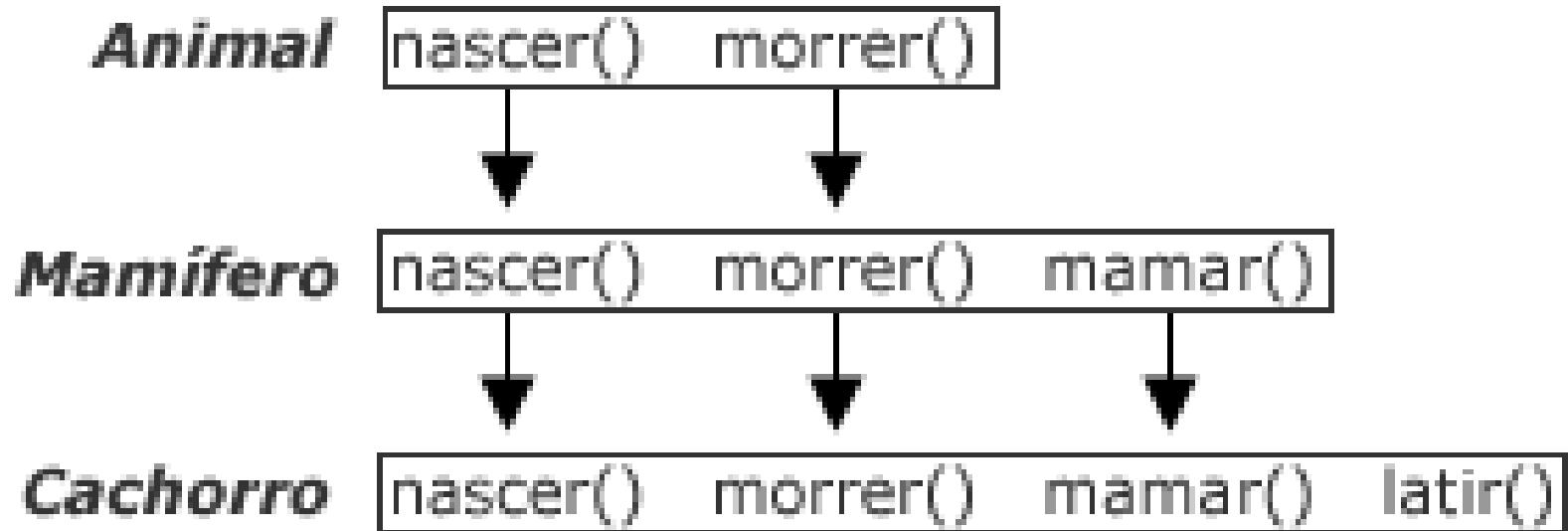
Se levarmos em conta a classe TV que estamos usando de exemplo até agora, podemos dizer que TV deriva de Eletrodomestico. Ou seja, a classe TV possui todas as características da classe Eletrodomestico, além de ter suas próprias características.

Extends e Super

Para fazermos uma classe herdar as características de uma outra, usamos a palavra reservada `extends` logo após a definição do nome da classe. Dessa forma:

```
class NomeDaClasseASerCriada extends NomeDaClasseASerHerdada
```

Importante: Java permite que uma classe herde apenas as características de uma única classe, ou seja, não pode haver heranças múltiplas. Porém, é permitido heranças em cadeias, por exemplo: se a classe Mamifero herda a classe Animal, quando fizermos a classe Cachorro herdar a classe Mamifero, a classe Cachorro também herdará as características da classe Animal.



Como estamos tratando de herança de classes, toda classe tem seu método construtor. Portanto, se estamos trabalhando com duas classes, temos dois métodos construtores. Para acessarmos o método construtor da classe que está sendo herdada usamos o `super()`.

Podemos usar o `super` para qualquer construtor da classe pai, pois o Java consegue diferenciar os construtores por causa da sobrecarga de métodos.

Para finalizar, iremos mostrar o exemplo citado mais acima da classe Eletrodomestico e TV.

Classe 1: Eletrodomestico

```
public class Eletrodomestico {  
    private boolean ligado;  
    private int voltagem;  
    private int consumo;  
  
    public Eletrodomestico(boolean ligado, int voltagem, int consumo) {  
        this.ligado = ligado;  
        this.voltagem = voltagem;  
        this.consumo = consumo;  
    }  
    // (...)  
}
```

Classe 2: TV

```
public class TV extends Eletrodomestico {  
    private int canal;  
    private int volume;  
    private int tamanho;  
  
    public TV(int voltagem, int consumo, int canal, int volume, int tamanho) {  
        super(false, voltagem, consumo);  
        this.canal = canal;  
        this.volume = volume;  
        this.tamanho = tamanho;  
    }  
    // (...)  
}
```

Classe que mostra a instanciação de TV.

```
public class ExemploHeranca {  
    public static void mostrarCaracteristicas(TV obj) {  
        System.out.print("Esta TV tem as seguintes caracteristicas:\n"  
            + "Tamanho: " + obj.getTamanho() + "\n"  
            + "Voltagem Atual: " + obj.getVoltagem() + "V\n"  
            + "Consumo/h: " + obj.getConsumo() + "W\n");  
        if (obj.isLigado()) {  
            System.out.println("Ligado: Sim\n"  
                + "Canal: " + obj.getCanal() + "\n"  
                + "Volume: " + obj.getVolume() + "\n");  
        } else {  
            System.out.println("Ligado: Não\n");  
        }  
    }  
  
    public static void main(String args[]) {  
        TV tv1 = new TV(110, 95, 0, 0, 21);  
        TV tv2 = new TV(220, 127, 0, 0, 29);  
        tv2.setLigado(true);  
        tv2.setCanal(3);  
        tv2.setVolume(25);  
        mostrarCaracteristicas(tv1);  
        mostrarCaracteristicas(tv2);  
    }  
}
```

O código irá mostrar o seguinte resultado:

Esta TV tem as seguintes características:

Tamanho: 21"

Voltagem Atual: 110V

Consumo/h: 95W

Ligado: Não

Esta TV tem as seguintes características:

Tamanho: 29"

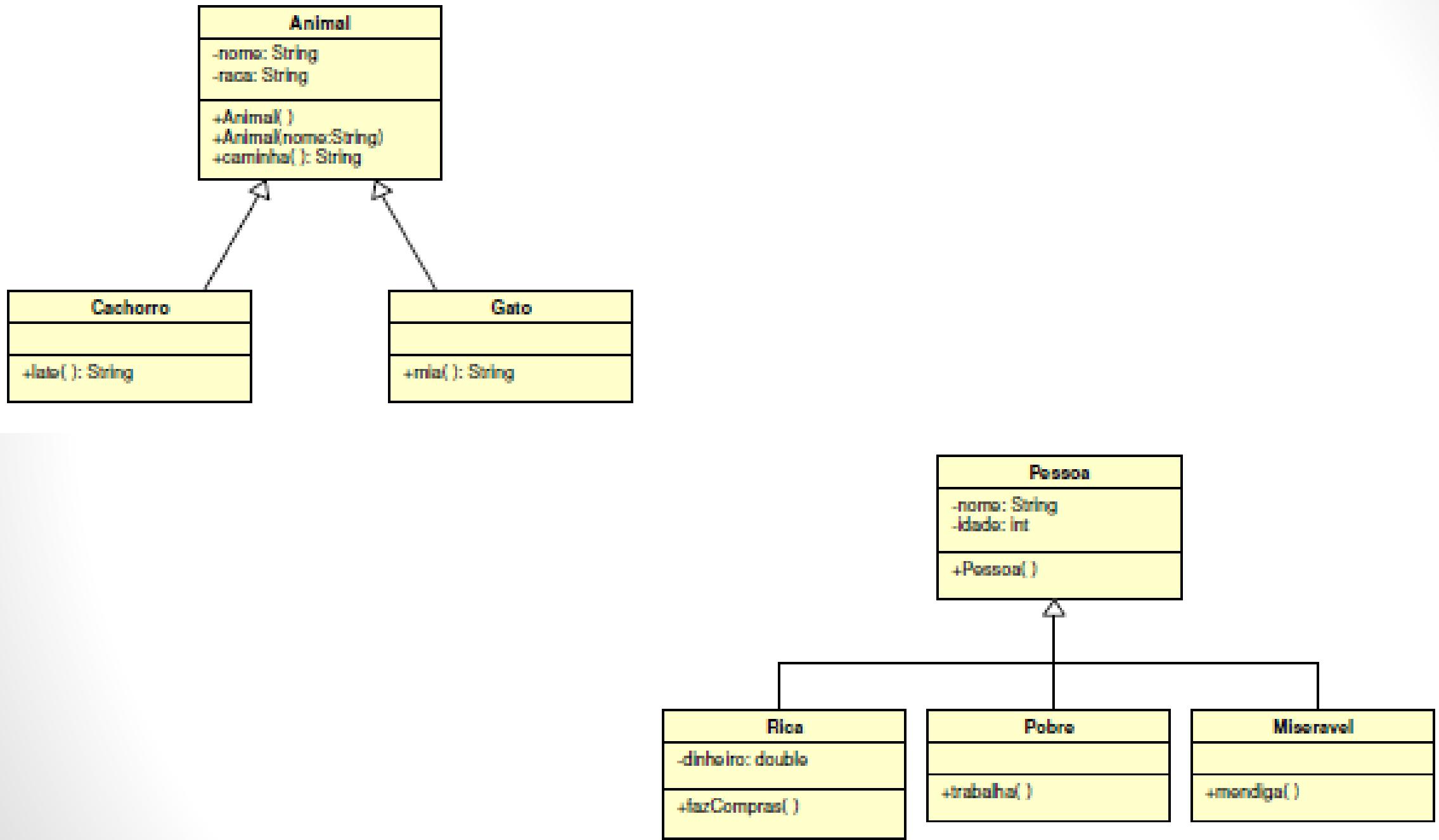
Voltagem Atual: 220V

Consumo/h: 127W

Ligado: Sim

Canal: 3

Volume: 25



Além disso, existem alguns comportamentos e atributos que só servem para um Ebook. Um deles é o watermark (marca d'água). Essa é a forma de identificar discretamente o nome e e-mail do dono daquele livro digital, normalmente no rodapé das páginas.

Se Ebook é um elemento importante, possui comportamentos e atributos específicos, ele deveria ser representado como um Objeto! Podemos criar uma classe Ebook definindo os atributos e comportamentos específicos desse novo tipo.

```
public class Ebook {  
  
    private String nome;  
    private String descricao;  
    private double valor;  
    private String isbn;  
    private Autor autor;  
    private String waterMark;  
  
    public void setWaterMark(String waterMark) {  
        this.waterMark = waterMark;  
    }  
  
    public String getWaterMark() {  
        return waterMark;  
    }  
  
    // getters, setters e outros métodos  
}
```

Nosso código já está um pouco mais interessante, afinal não estamos mais sobrecarregando a classe `Livro` com atributos e métodos que serão utilizados apenas quando o tipo do livro for um *ebook*. Mas há muito código repetido aqui: além dos comportamentos do `Ebook`, temos todos os atributos e métodos já escritos na classe `Livro`.

Para evitar toda essa repetição de código, podemos ensinar ao compilador que o `Ebook` é um tipo de `Livro`, ou seja, além de seus próprios atributos e métodos, essa classe possui tudo o que um `Livro` tem. Para fazer isto, basta `Ebook` dizer na declaração da classe que ela é um `Livro`, que é uma **extensão** dessa classe:

```
public class Ebook extends Livro {  
  
    private String waterMark;  
  
    public Ebook(Autor autor) {  
        super(autor);  
    }  
  
    public void setWaterMark(String waterMark) {  
        this.waterMark = waterMark;  
    }  
  
    public String getWaterMark() {  
        return waterMark;  
    }  
}
```

Como a classe `Livro` tinha um construtor obrigando a passagem de um `Autor` como parâmetro, ao herdar de um `Livro`, a classe `Ebook` também herdou essa responsabilidade. Repare que utilizamos a palavra `super` para delegar a responsabilidade para a *superclasse* que já tem esse comportamento bem definido.

```
public Ebook(Autor autor) {  
    super(autor);  
}
```

Ao utilizar a palavra reservada `extends`, estamos dizendo que um `Ebook` (*subclasse*) **herda** tudo o que a classe `Livro` (*superclasse*) tem. Portanto, mesmo sem ter nenhum desses métodos declarados diretamente na classe `Ebook`

Como a classe `Livro` tem os *setters* para o atributo `nome` declarados, um `Ebook` também terá.

HERANÇA MÚLTIPLA

Uma regra importante da herança em Java é que nossas classes só podem herdar diretamente de **uma** classe pai. Ou seja, não há herança múltipla como na linguagem C++. Mas sim, uma classe pode herdar de uma classe que herda de outra e assim por diante. Você pode encadear a herança de suas classes, no entanto, veremos mais à frente que essa estratégia não é muito interessante por aumentar de mais o acoplamento entre suas classes.

POLIMORFISMO

Polimorfismo, que vem do grego "muitas formas". É o termo definido em linguagens orientadas a objeto - como o Java - para a possibilidade de se usar o mesmo elemento de forma diferente. Especificamente em Java, polimorfismo se encontra no fato de podemos modificar totalmente o código de um método herdado de uma classe diferente, ou seja, sobrescrevemos o método da classe pai.

Portanto, polimorfismo está intimamente ligado a herança de classes.

Um pequeno exemplo para simplificar essa característica segue abaixo:

- a. Classe 1 possui 2 métodos: métodoA() e métodoB().
- b. Classe 2 herda a classe 1.
- c. Classe 2 reescreve todo o métodoA() que pertence a classe 1.

Levando isso a um patamar mais prático.

Sabemos que toda classe em Java herda implicitamente a classe Object.

A classe Object possui alguns métodos, dentre eles o método `toString()`.

O método `toString()` original, descreve qual instância de objeto está sendo utilizada.

Resumidamente, ele cria um texto com o nome da classe mais um código hexadecimal que cada instância possui diferente de outra, chamado hash code (é como um RG daquela instância).

Então, se tivéssemos a classe TV dentro do pacote senac e usássemos o comando:
System.out.println (tv1.toString()). O que apareceria no console seria algo como:

[senac.TV@c17124.](#)

Então o que faremos para melhorar será usar o polimorfismo para sobrescrever o método `toString()`, colocando o texto da forma que desejarmos.

```
public class TV {  
    private String marca;  
    private String modelo;  
    private int tamanho;  
    private int canal;  
    private int volume;  
    private boolean ligada;  
  
    public TV(String marca, String modelo, int tamanho) {  
        this.marca = marca;  
        this.modelo = modelo;  
        this.tamanho = tamanho;  
        this.canal = 1;  
        this.volume = 0;  
        this.ligada = false;  
    }  
  
    // métodos getters e setters  
  
    public String toString() {  
        return "TV" +  
            "\nMarca: " + this.marca +  
            "\nModelo: " + this.modelo +  
            "\nTamanho: " + this.tamanho;  
    }  
  
    public static void main(String args[]) {  
        TV tv1 = new TV("Marcal", "SDX-22", 29);  
        System.out.println(tv1.toString());  
    }  
}
```

Agora, o resultado da linha System.out.println (tv1.toString()) não será mais:

senac.TV@c17124

Agora, será:

TV

Marca: Marca1

Modelo: SDX-22

Tamanho: 29

É possível que dependendo do seu JRE e compilador seja necessário o uso da annotation @override antes de sobrescrever o método toString();