

JAVA FUNDAMENTOS

IDES, VARIÁVEIS, CONVERSÕES E **HELLO WORLD**

THIAGO YAMAMOTO



LISTA DE FIGURAS

| | |
|---|----|
| Figura 2.1 – Eclipse IDE for Java EE Developers | 5 |
| Figura 2.2 – Arquivo principal de execução | 6 |
| Figura 2.3 – Escolha da Workspace (diretório de trabalho). | 6 |
| Figura 2.4 – Escolha de Workspace (diretório de trabalho) | 7 |
| Figura 2.5 – IDE Eclipse..... | 8 |
| Figura 2.6 – Criação de um Projeto Java (Parte I). | 8 |
| Figura 2.7 – Criação de um Projeto Java (Parte II). | 9 |
| Figura 2.8 – Criação de um Projeto Java (Parte III). | 10 |
| Figura 2.9 – Resultado da criação do projeto..... | 10 |
| Figura 2.10 – Criando pacotes (parte 1)..... | 12 |
| Figura 2.11 – Criando Pacotes (parte 2) | 13 |
| Figura 2.12 – Pacote no Package Explorer..... | 14 |
| Figura 2.13 – Estrutura de pacotes no Windows Explorer. | 14 |
| Figura 2.14 – Criação de Classe (parte 1)..... | 15 |
| Figura 2.15 – Criação de classe (parte 2). | 16 |
| Figura 2.16 – Arquivo gerado e a o código da classe. | 17 |
| Figura 2.17 – Visão lógica da IDE eclipse..... | 18 |
| Figura 2.18 – Visão detalhada da IDE eclipse..... | 18 |
| Figura 2.19 – Primeira classe Java | 19 |
| Figura 2.20 – Projeto sem salvar..... | 20 |
| Figura 2.21 – Botão para Salvar e Executar o programa Java..... | 20 |
| Figura 2.22 – Executando a classe Java..... | 21 |
| Figura 2.23 – Resultado da execução do programa Java. | 21 |
| Figura 2.24 – Palavras reservadas..... | 23 |
| Figura 2.25 – Tipos primitivos inteiros..... | 24 |
| Figura 2.26 – Tipos primitivos de ponto flutuante..... | 24 |
| Figura 2.27 – Exemplo tipos primitivos..... | 25 |
| Figura 2.28 – Conversões de tipos de dados | 26 |
| Figura 2.29 – Conversões entre tipos primitivos | 26 |
| Figura 2.30 – Conversão utilizando cast | 27 |
| Figura 2.31 – Operadores aritméticos | 28 |
| Figura 2.32 – Operadores de atribuição..... | 29 |
| Figura 2.33 – Exemplo de utilização dos operadores de incremento e decremento . | 30 |
| Figura 2.34 – Operadores aritméticos e de atribuição..... | 31 |
| Figura 2.35 – Resultado da execução | 31 |
| Figura 2.36 – Tabela Operadores de igualdade e relacionais | 32 |
| Figura 2.37 – Fluxo para a instrução if | 35 |
| Figura 2.38 – Fluxo para a instrução if – else..... | 36 |
| Figura 2.39 – Fluxo para if – else aninhados | 37 |
| Figura 2.40 – Exemplo de leitura de dados | 38 |
| Figura 2.41 – Exemplo de leitura de dados..... | 39 |
| Figura 2.42 – Exemplo de leitura de dados..... | 39 |

SUMÁRIO

| | |
|---|----|
| 2 IDES, VARIÁVEIS, CONVERSÕES E HELLO WORLD | 4 |
| 2.1 Introdução | 4 |
| 2.2 Tipos de dados e variáveis..... | 22 |
| 2.2.1 Atribuindo valores às variáveis | 25 |
| 2.3 Operadores | 27 |
| 2.3.1 Operadores de atribuição | 29 |
| 2.3.2 Operadores de incremento e decremento | 30 |
| 2.3.3 Operadores de igualdade e relacionais | 32 |
| 2.3.4 Operadores lógicos | 33 |
| 2.4 Fluxo de controle e escopo de bloco | 34 |
| 2.5 Entrada e saída de dados | 37 |

2 IDES, VARIÁVEIS, CONVERSÕES E HELLO WORLD

2.1 Introdução

Antes de começar, precisamos entender sobre as IDEs – programas de computador que ajudam no desenvolvimento de sistemas. É possível implementar sistemas em Java sem nenhuma ferramenta específica, porém é uma tarefa muito trabalhosa e pouco produtiva.

Uma IDE (*Integrated Development Environment* ou Ambiente de Desenvolvimento Integrado) é um programa que visa maximizar a produtividade do desenvolvedor, que tem várias funcionalidades que auxiliam no desenvolvimento, entre elas, as mais comuns são:

- **Editor:** editor de código-fonte específico para cada linguagem de programação suportada pela IDE.
- **Compilador:** compila o código-fonte.
- **Debugger (Depurador):** executa o programa “passo-a-passo”, por meio do qual é possível verificar o que ocorre em cada linha do programa, auxiliando no entendimento do sistema e no processo de encontrar e corrigir problemas.
- **Modelagem:** criação de modelos de classes, objetos, interfaces, associações e interações de forma visual.
- **Geração de código:** geração de código a partir de *templates* de código comumente utilizados para solucionar problemas rotineiros.
- **Deploy (Distribuição):** auxilia no processo de gerar o arquivo final para a instalação do programa desenvolvido.
- **Testes automatizados:** realiza testes no programa de forma automatizada, baseados em scripts ou programas de testes previamente especificados, gerando relatórios que auxiliam na análise do impacto das alterações no código-fonte.

- **Refactoring (Refatoração):** consiste na melhoria constante do código-fonte do programa, seja na construção de código mais otimizado, mais limpo e/ou com melhor entendimento pelos envolvidos no desenvolvimento do software.

Existem várias IDEs disponíveis no mercado, destinadas a linguagens de programação ou plataforma de desenvolvimento específica. São exemplos de IDEs: Visual Studio, da Microsoft, Netbeans, da Oracle etc.

Para a plataforma Java existem várias IDEs como:

- Eclipse
- NetBeans
- JDeveloper

Durante o curso vamos utilizar a IDE Eclipse, líder no mercado de desenvolvimento Java, totalmente gratuito.

O Eclipse não é um software instalável, é necessário simplesmente realizar o download e acessar o arquivo principal de execução.

Para a sua utilização é necessário instalar a JDK do Java. (Link para *download*: <http://www.oracle.com/technetwork/java/javase/downloads/index.html>).

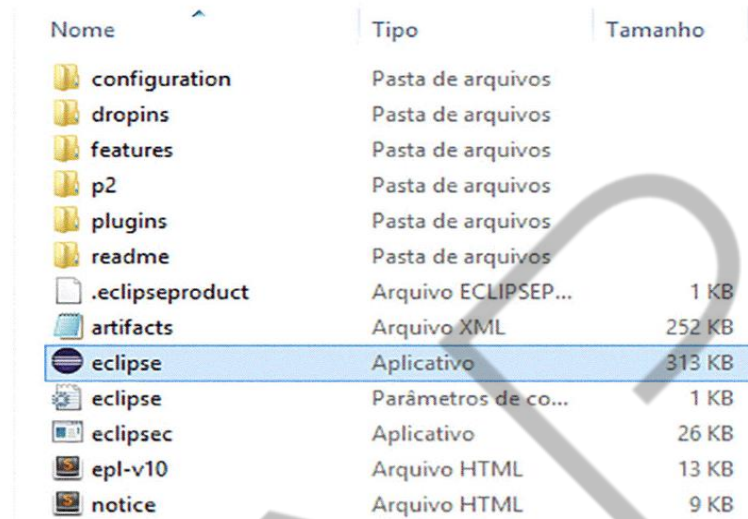
O eclipse possui várias versões. Vamos utilizar o “**Eclipse IDE for Java EE Developers**”, pois ela dá suporte ao Java Enterprise Edition, ou seja, além de toda a parte básica do Java, podemos utilizar a parte Enterprise, incluindo projetos Java Web.



Figura 2.1 – Eclipse IDE for Java EE Developers

Fonte: elaborado pelo autor (2018)

Baixe o arquivo e descompacte. Dentro da pasta descompactada, encontre e clique duas vezes no arquivo de inicialização da ferramenta:



| Nome | Tipo | Tamanho |
|-----------------|---------------------|---------------|
| configuration | Pasta de arquivos | |
| dropins | Pasta de arquivos | |
| features | Pasta de arquivos | |
| p2 | Pasta de arquivos | |
| plugins | Pasta de arquivos | |
| readme | Pasta de arquivos | |
| .eclipseproduct | Arquivo ECLIPSEP... | 1 KB |
| artifacts | Arquivo XML | 252 KB |
| eclipse | Aplicativo | 313 KB |
| eclipse | Parâmetros de co... | 1 KB |
| eclipseec | Aplicativo | 26 KB |
| epl-v10 | Arquivo HTML | 13 KB |
| notice | Arquivo HTML | 9 KB |

Figura 2.2 – Arquivo principal de execução

Fonte: Elaborado pelo autor (2018)

O próximo passo é definir um diretório para gravar os arquivos gerados de dentro do eclipse. Esse diretório é conhecido como *workspace* (diretório de trabalho).

Para isso, clique no botão “browser” e escolha uma pasta qualquer.

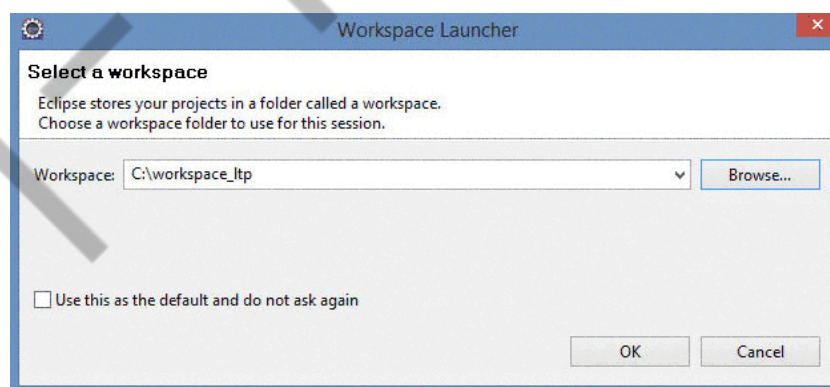


Figura 2.3 – Escolha da Workspace (diretório de trabalho).

Fonte: Elaborado pelo autor (2018)

Como a pasta estava vazia, o eclipse exibe uma página de boas-vindas, com opções para visualizar tutoriais, exemplos e até um overview da ferramenta.

Vamos fechar essa página, clique no “X” na aba “Welcome”.



Figura 2.4 – Escolha de Workspace (diretório de trabalho)

Fonte: Elaborado pelo autor (2018)

O Eclipse é formado por várias janelas. Toda a composição da tela pode ser alterada pelo usuário, assim as janelas podem ser fechadas ou abertas, posicionadas conforme as preferências do desenvolvedor.

Vamos utilizar duas janelas neste momento:

- **Project explorer:** está no lado esquerdo do eclipse, é onde a estrutura e organização dos projetos e arquivos podem ser visualizados e acessados.
- **Source editor:** situado na área central (em cinza), é onde visualizamos e editamos o código fonte.

Vamos explorar as outras abas na medida em que avançamos na disciplina.

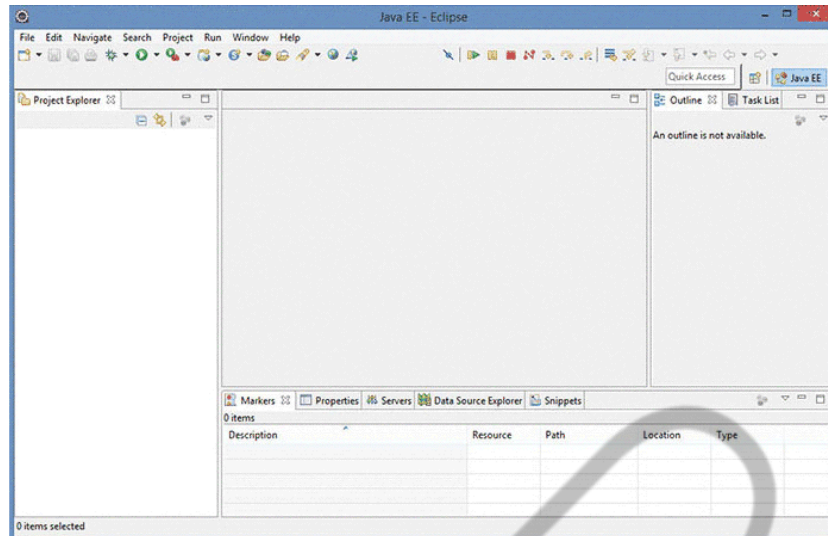


Figura 2.5 – IDE Eclipse

Fonte: Elaborado pelo autor (2018)

Agora, vamos criar o nosso primeiro projeto Java:

Para isso, vá ao menu – **File – New – Project**, conforme a figura abaixo:

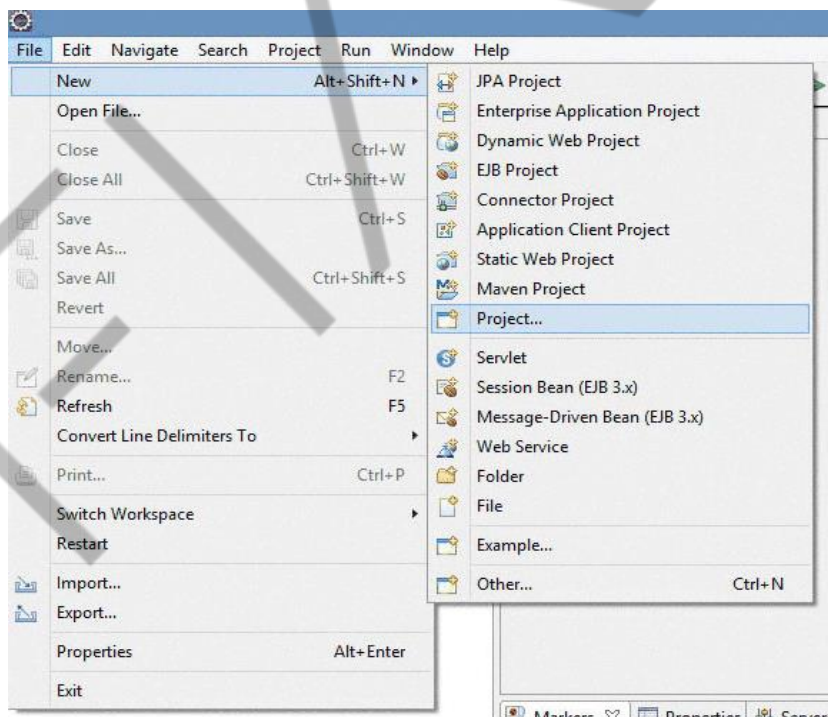


Figura 2.6 – Criação de um Projeto Java (Parte I).

Fonte: Elaborado pelo autor (2018)

Escolha a primeira opção, **Java Project** e clique em **Next**.

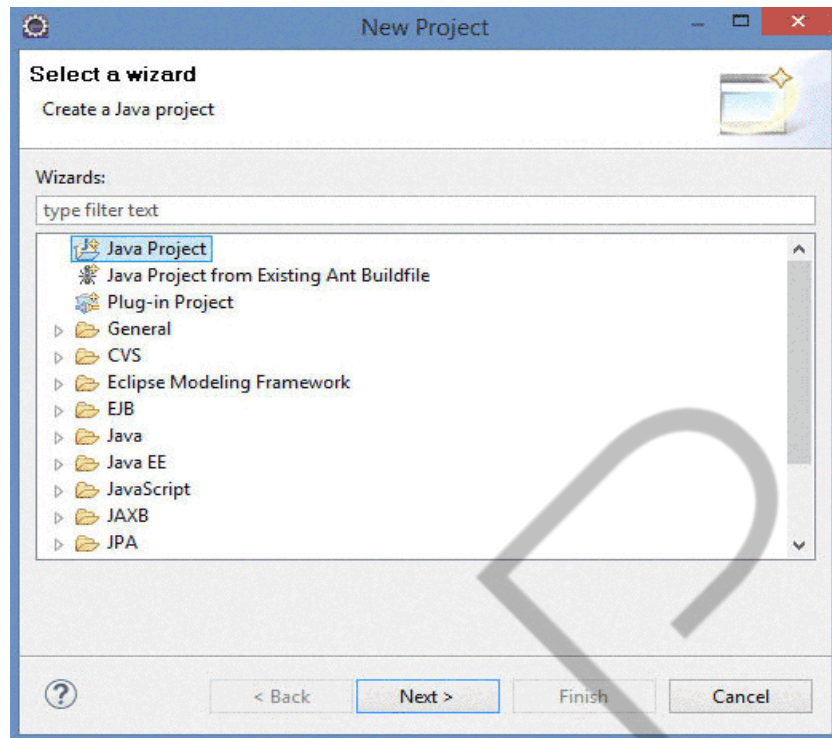


Figura 2.7 – Criação de um Projeto Java (Parte II).

Fonte: Elaborado pelo autor (2018)

Agora é a hora de definir o nome do projeto. Como sugestão, dê o nome de **01-OlaMundo** e clique em **Finish**.

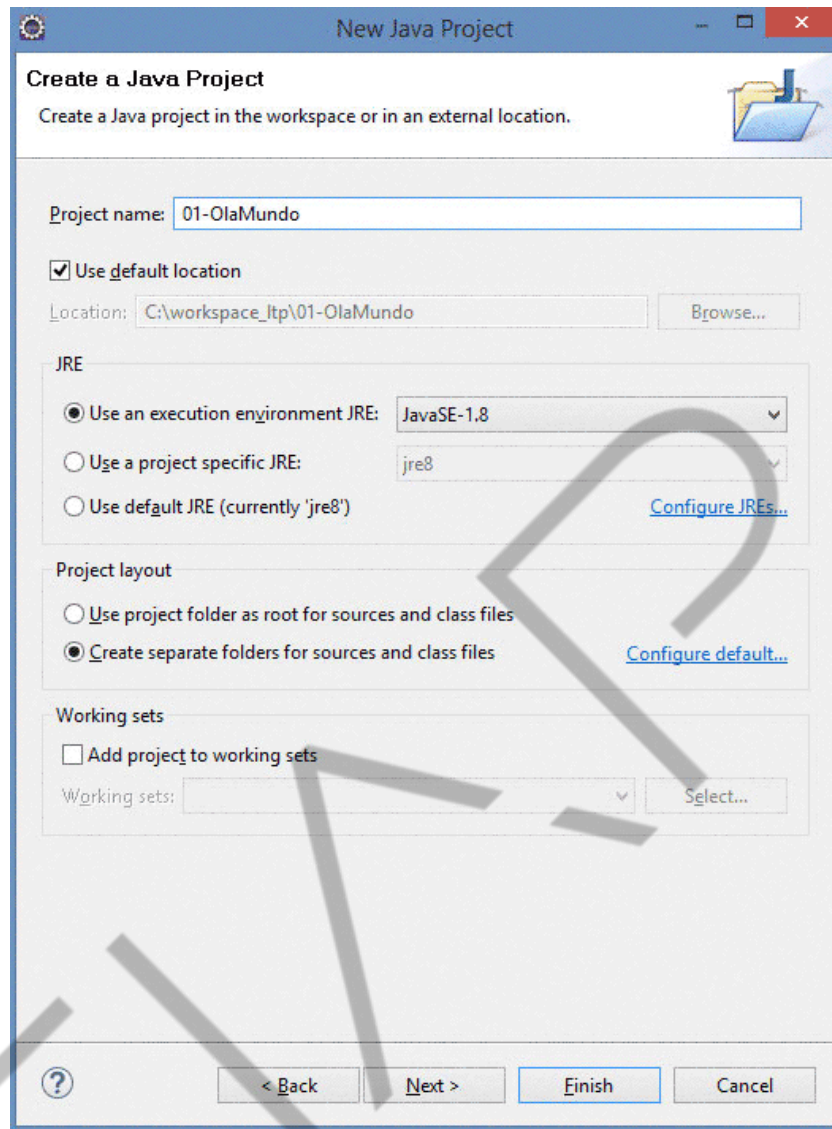


Figura 2.8 – Criação de um Projeto Java (Parte III).

Fonte: Elaborado pelo autor (2018)

O resultado pode ser visualizado na aba de **Package Explorer**, localizado na janela à esquerda no eclipse.

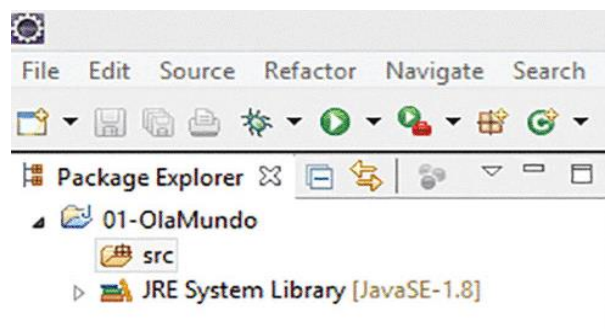


Figura 2.9 – Resultado da criação do projeto.

Fonte: Elaborado pelo autor (2018)

A pasta **src** é onde devemos criar os arquivos Java. Abaixo dela, estão localizadas as bibliotecas de classes básicas do Java. (JRE System Library).

Depois de definir a **workspace** e criar um **projeto**, está na hora de criar a nossa primeira classe **Java**. Para isso, vamos primeiro definir o diretório em que ele será criado. A hierarquia de diretórios para a organização dos arquivos de um programa em Java é denominada **Pacotes**.

Imagine um sistema corporativo, como um Sistema de um Banco ou de Telefonia que precisa processar milhares de informações para seus clientes, como: processar as transações financeiras de uma conta ou calcular a fatura de telefone.

Quantas classes são necessárias para desenvolver um sistema desses? Centenas. Agora imagine se todas as classes estivessem no mesmo diretório. Seria muito difícil de gerenciá-las. Por isso, devemos sempre criar uma estrutura de diretório (pacotes) para organizar os arquivos do sistema. Dessa forma, podemos agrupar as classes em coleções e separá-las das bibliotecas de classes fornecidas por outras empresas.

Os pacotes em Java seguem uma hierarquia. Assim como podemos ter diretórios e subdiretórios no disco rígido, podemos organizar os pacotes em níveis hierárquicos.

Além de organizar, utilizar pacotes garante a singularidade dos nomes das classes. Em um sistema não podemos ter classes com o mesmo nome dentro do mesmo pacote. Porém, não há conflito se as classes estiverem em pacotes diferentes. Assim, podemos ter duas classes com o nome Cliente no sistema, desde que estejam em pacotes diferentes.

Para garantir um nome de pacote único, é recomendado utilizar o nome de domínio da internet da empresa (que é único) escrito ao contrário. E depois especificar mais ainda os pacotes com nome do projeto, tipos de classes etc.

Por exemplo, para a faculdade FIAP, que possui o domínio **fiap.com.br** e está desenvolvendo um projeto para *e-commerce*, o pacote será definido como **br.com.fiap.ecommerce**.

Vamos criar um pacote para o nosso projeto. Será o **br.com.fiap.tds**, para agrupar as classes do projeto do curso de TDS.

Para isso, clique com o botão direito do mouse no diretório **src** e escolha **new – package**.

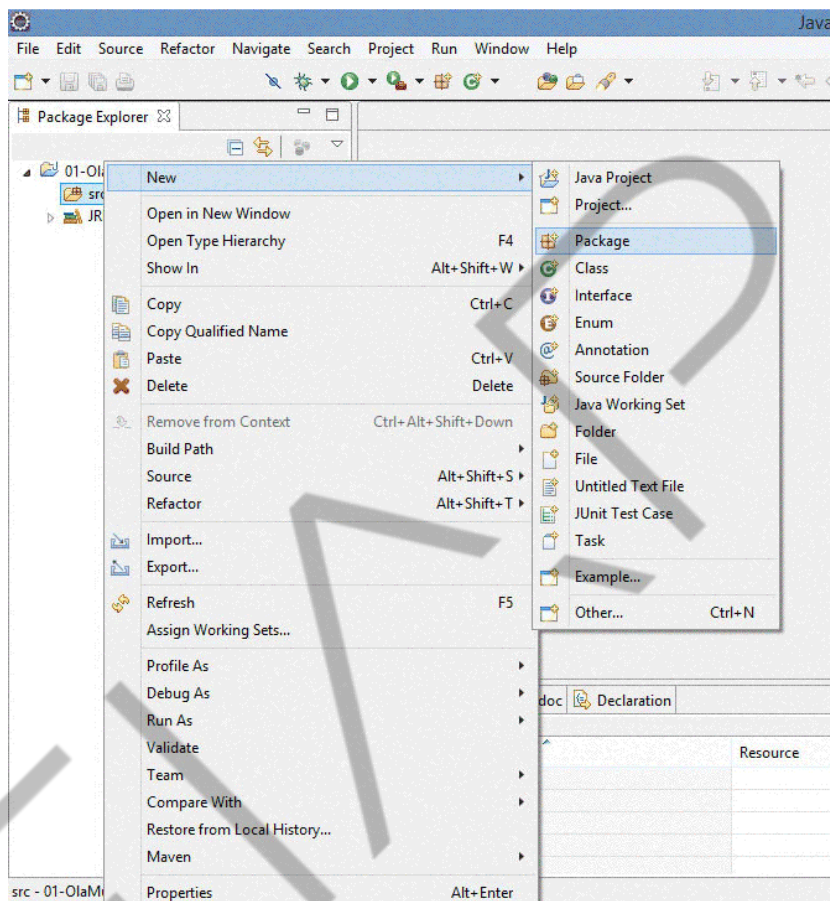


Figura 2.10 – Criando pacotes (parte 1)

Fonte: Elaborado pelo autor (2018)

Agora vamos definir o nome do pacote e finalizar o processo com o botão **Finish**.

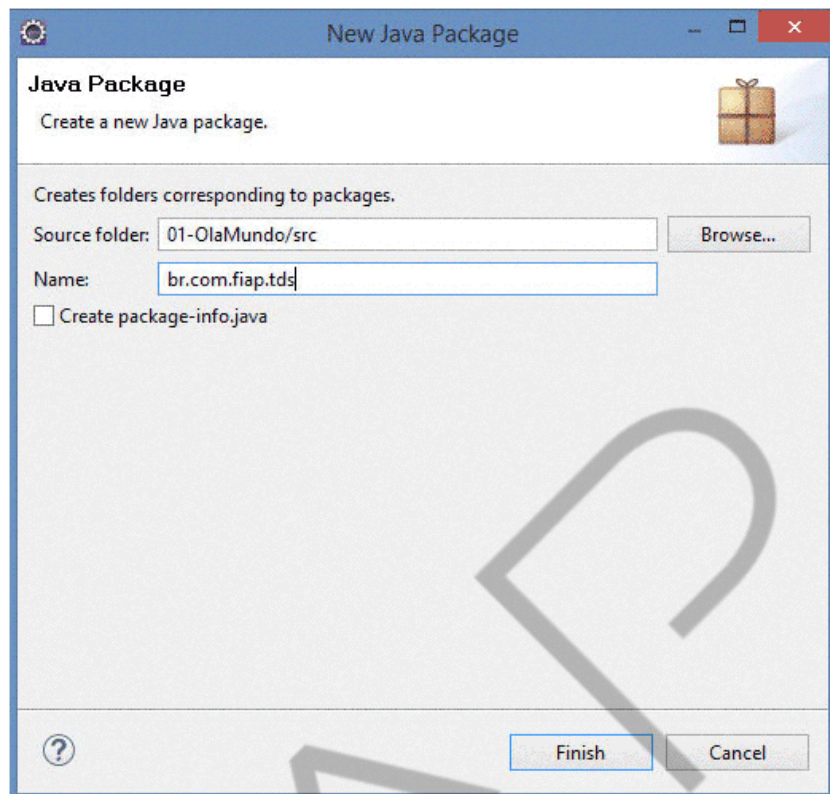


Figura 2.11 – Criando Pacotes (parte 2)

Fonte: Elaborado pelo autor (2018)

O resultado pode ser visualizado no **Package Explorer**:

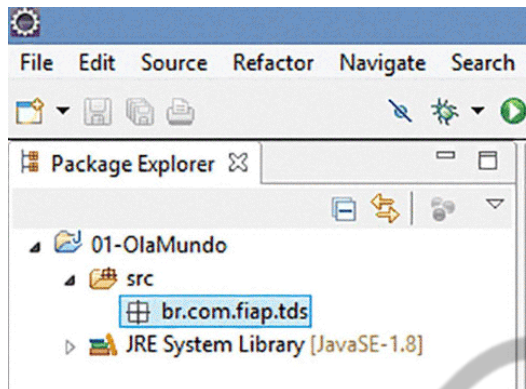


Figura 2.12 – Pacote no Package Explorer.

Fonte: Elaborado pelo autor (2018)

Agora vamos ver a estrutura que foi criada no disco rígido. Para isso, navegue até a pasta do workspace no Windows Explorer.

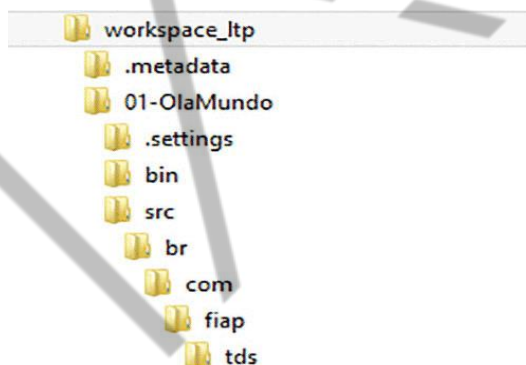


Figura 2.13 – Estrutura de pacotes no Windows Explorer.

Fonte: Elaborado pelo autor (2018)

Podemos visualizar uma pasta que representa o Projeto “**01-OlaMundo**”. Dentro da pasta do projeto, temos a pasta **src** com uma estrutura de diretórios do pacote criado. Assim, podemos perceber que cada pasta do pacote é separada pelo “.” (ponto).

Agora, vamos criar a nossa primeira classe Java. Para isso, clique com o botão direito do mouse no pacote e escolha **New => Class**.

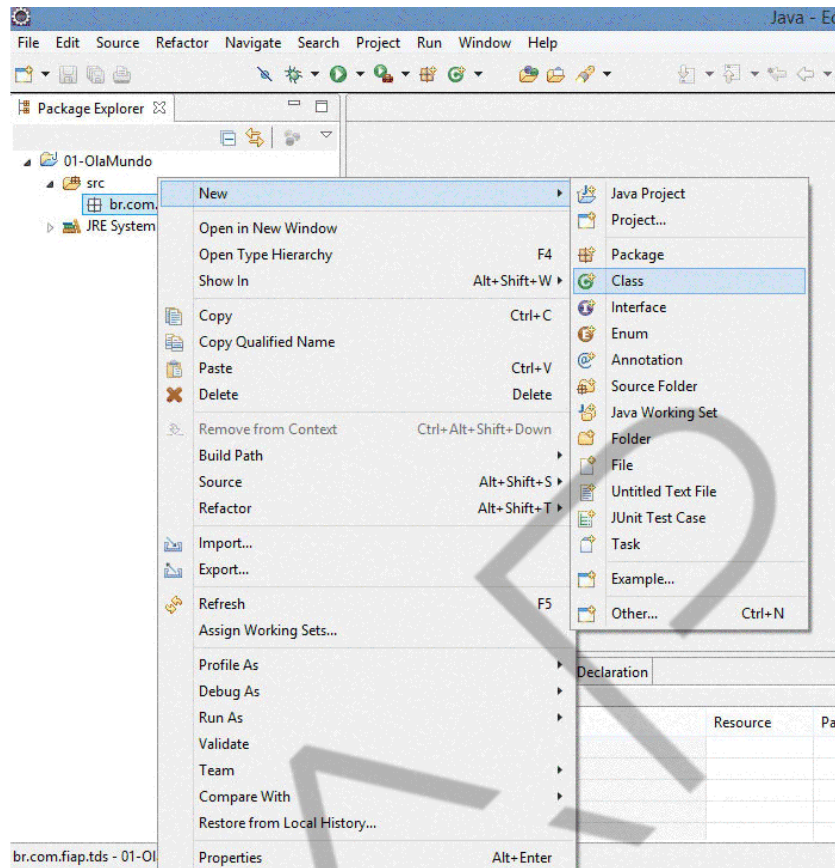


Figura 2.14 – Criação de Classe (parte 1)

Fonte: Elaborado pelo autor (2018)

É hora de definir o nome da classe. Vamos dar o nome de **OlaMundo** e finalizar o processo com o botão **Finish**.

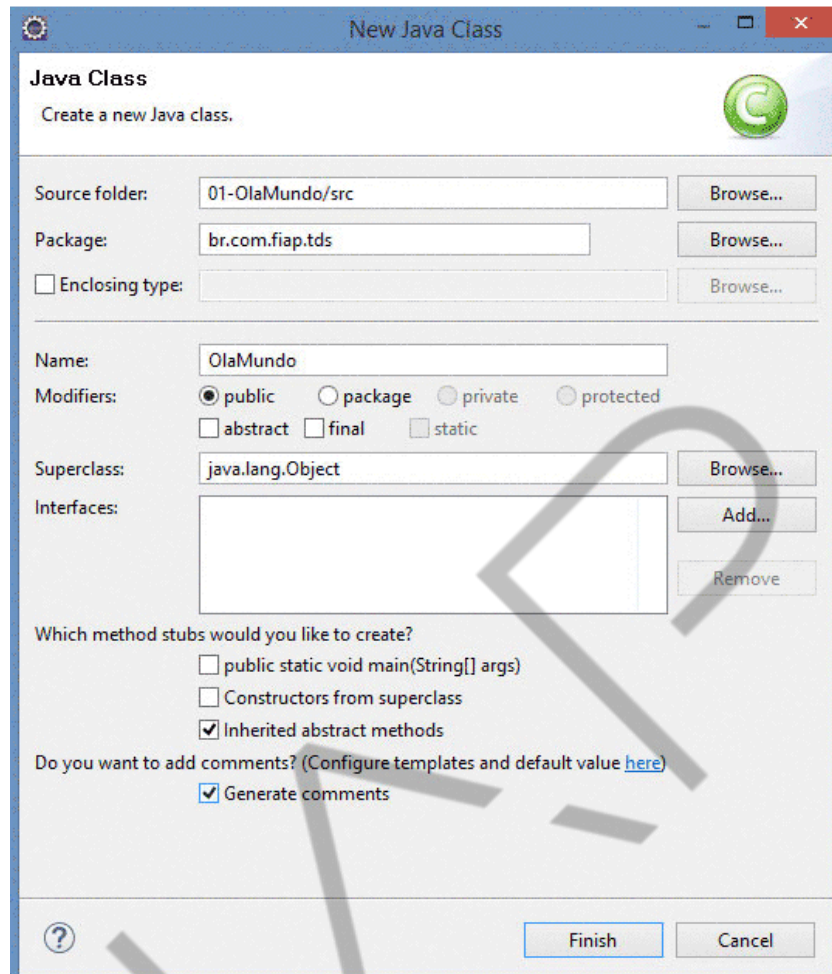


Figura 2.15 – Criação de classe (parte 2).

Fonte: Elaborado pelo autor (2018)

O resultado pode ser visualizado na área central do eclipse, onde está localizado o editor de código fonte. A classe possui uma primeira instrução, que define o pacote da classe: **package br.com.fiap.tds**

A definição de pacote sempre é feita na primeira linha do arquivo Java.

Após, temos a definição da classe.

```
public class OlaMundo {  
  
}
```

A princípio, não será necessário entender o significado da palavra **public**. Veremos isso com detalhes no decorrer do curso.

Após a palavra-chave **class**, vem o nome da classe. Atente-se que o nome do arquivo gerado é o mesmo da classe com a extensão .java. Isso é importante, pois é obrigatório que o nome do arquivo seja o mesmo da classe pública. Os nomes devem iniciar com uma letra, depois, podem conter quaisquer combinações entre letras e dígitos.

O padrão para atribuir nomes a classes é utilizar substantivos que iniciam com uma letra maiúscula. Se o nome tiver mais de uma palavra, cada palavra deve iniciar com letra maiúscula, por exemplo: PrimeiroExemplo, OlaMundo. Essa notação de letras maiúsculas no meio de uma palavra é chamada de “notação camelo” ou “CamelCase”.

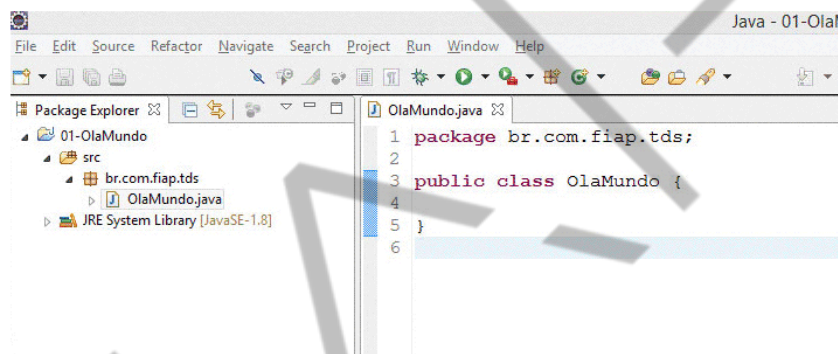


Figura 2.16 – Arquivo gerado e a o código da classe.

Fonte: Elaborado pelo autor (2018)

Observe as chaves { } no código-fonte. No Java, as chaves delimitam os blocos em um programa. Neste caso, as chaves delimitam o início e o fim da classe OlaMundo.

Agora que sabemos o que são as workspaces, projetos, pacotes e classes, podemos ter uma visão lógica da IDE eclipse:

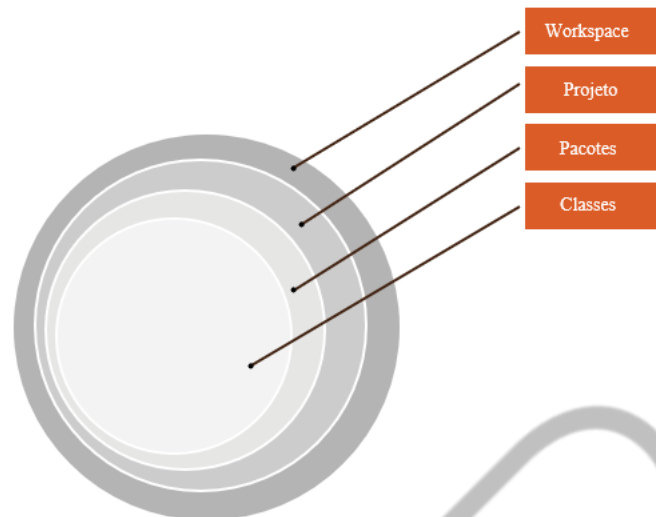


Figura 2.17 – Visão lógica da IDE eclipse.

Fonte: Elaborado pelo autor (2018)

Dentro de uma workspace, definimos os projetos. Projetos podem ter vários pacotes, e estes, por sua vez, podem conter várias classes.

A visão detalhada pode ser visualizada na figura abaixo:

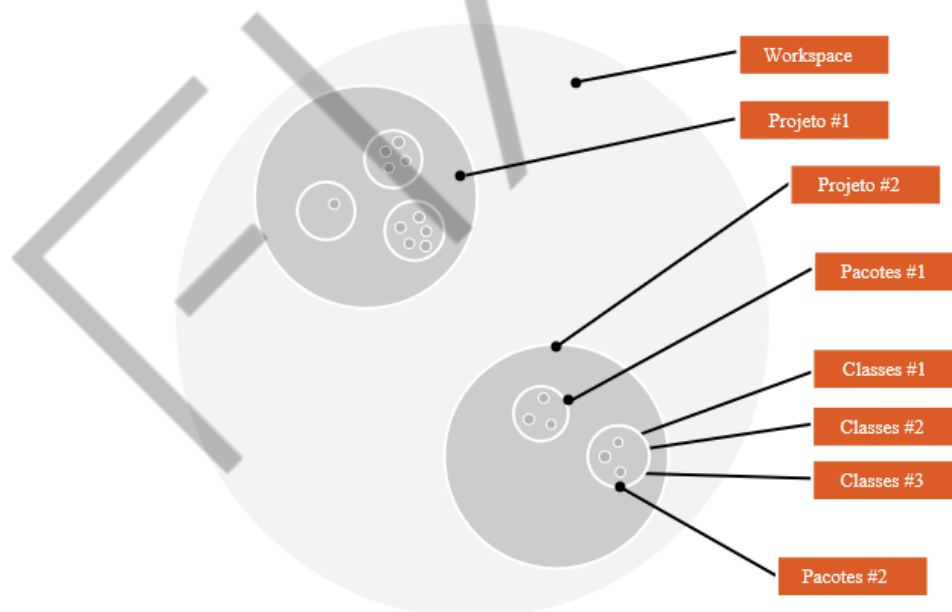
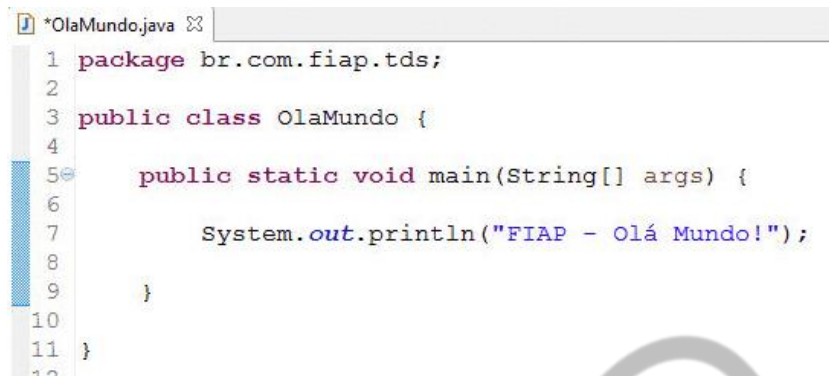


Figura 2.18 – Visão detalhada da IDE eclipse.

Fonte: Elaborado pelo autor (2018)

Já criamos a primeira classe Java, agora vamos adicionar um código para executar o programa e exibir informações.

Implemente a classe conforme a classe abaixo:



```
1 package br.com.fiap.tds;
2
3 public class OlaMundo {
4
5     public static void main(String[] args) {
6
7         System.out.println("FIAP - Olá Mundo!");
8
9     }
10
11 }
```

Figura 2.19 – Primeira classe Java

Fonte: Elaborado pelo autor (2018)

Na linha 5, o código **public static void main(String[] args)** define o método **main**. As chaves de abertura e fechamento estão delimitando o início e o fim do método.

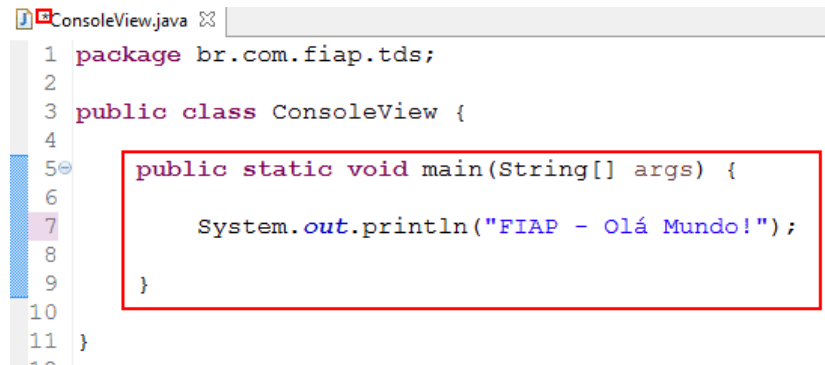
Por enquanto, não se preocupe com o método **main**. Vamos explorar os métodos em breve. O que você precisa saber no momento é que para executar uma classe compilada, a máquina virtual do Java sempre inicia a execução pelo método **main**. Portanto, é necessário ter um método **main** na classe que você deseja executar. Mas podemos criar várias classes e chamá-las a partir do método **main**.

A instrução **System.out.println()**, que está dentro do método **main**, exibe na console a informação que está entre os parênteses, neste caso **“FIAP – Olá Mundo”**. Toda instrução deve terminar com ponto e vírgula (;).

Outro detalhe é o asterisco antes do nome da classe na aba do editor. Esse asterisco indica que a classe não foi salva. Dessa forma, utilize o atalho **c**.

CTRL + S ou o botão **Save**, localizado na barra de ferramentas.

É sempre importante salvar os arquivos, pois nesse momento o Eclipse manda compilar a classe Java.



```
1 package br.com.fiap.tds;
2
3 public class ConsoleView {
4
5     public static void main(String[] args) {
6
7         System.out.println("FIAP - Olá Mundo!");
8
9     }
10
11 }
```

Figura 2.20 – Projeto sem salvar

Fonte: Elaborado pelo autor (2018)

Vamos executar o programa. Para isso, utilize o atalho **F11** ou o botão “play” localizado na barra de ferramentas:

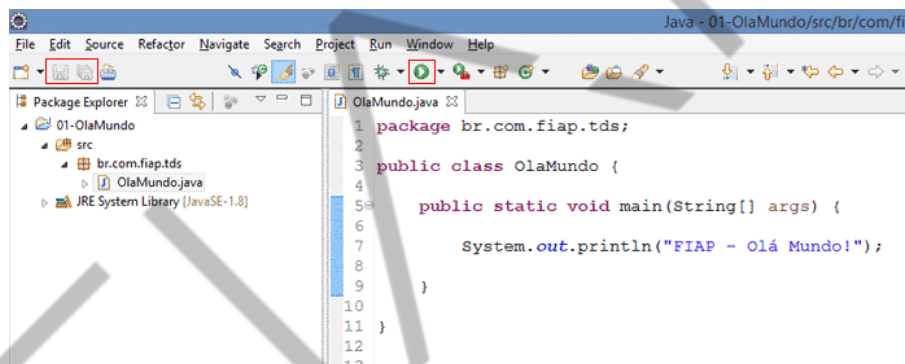


Figura 2.21 – Botão para Salvar e Executar o programa Java.

Fonte: Elaborado pelo autor (2018)

Existe mais uma opção para executar a classe Java. Basta clicar com o botão direito do mouse em cima da classe e escolher as opções: **Run As => Java Application**.

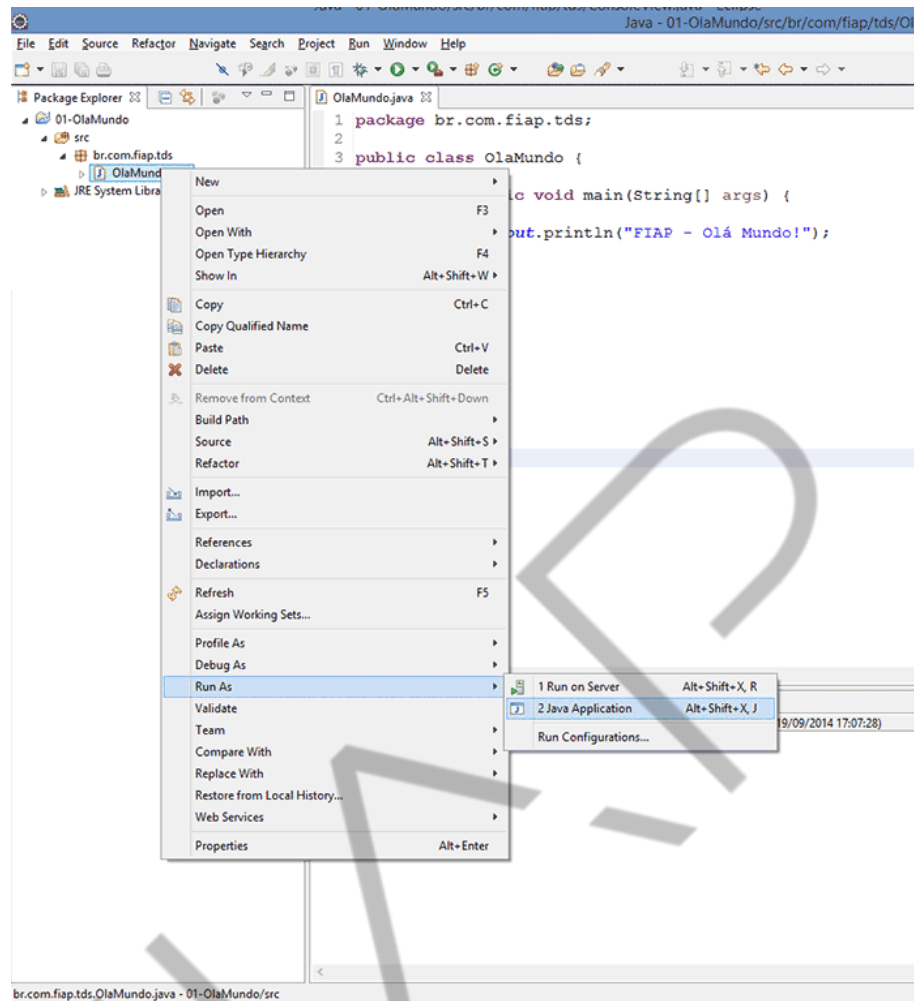


Figura 2.22 – Executando a classe Java.

Fonte: Elaborado pelo autor (2018)

O resultado é apresentado na figura abaixo:

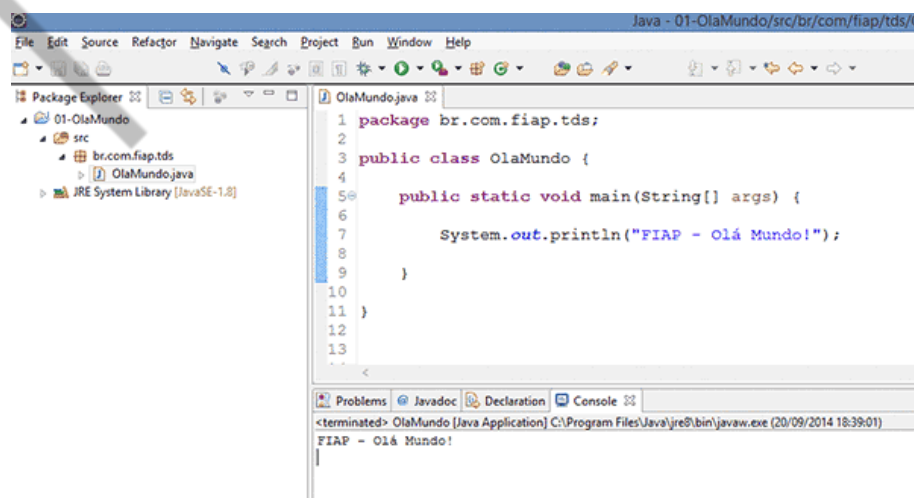


Figura 2.23 – Resultado da execução do programa Java.

Fonte: Elaborado pelo autor (2018)

A informação é exibida no Console do Eclipse.

Pronto! Desenvolvemos o primeiro programa em Java. Agora vamos estudar os fundamentos básicos da programação Java.

2.2 Tipos de dados e variáveis

Para armazenar informações no programa, Java precisamos de variáveis. As variáveis são compostas pelo seu nome e o tipo de informação que irá armazenar. Para declará-la, primeiro é definido o tipo e depois o nome da variável, como por exemplo:

```
int idade;
```

```
double preco;
```

É possível também declarar mais de uma variável do mesmo tipo de uma só vez:

```
double preco, salario;
```

Dessa forma, sabemos que o Java é uma linguagem fortemente tipada, pois cada variável precisa ter um tipo declarado. Existem oito tipos primitivos para armazenamento de informações. Tipos primitivos não são objetos, eles são partes internas da linguagem Java, o que os tornam mais eficientes. Uma variável também pode armazenar um objeto, veremos isso mais adiante.

Os nomes das variáveis podem começar com uma letra, um caractere de sublinhado (`_`) ou `$`. Depois do primeiro caractere, os nomes das variáveis podem conter qualquer combinação de letras ou números.

Por padrão, o nome da variável deve começar com um caractere em minúsculo e se for composto por mais de uma palavra, a próxima palavra deve começar com caractere em maiúscula.

Lembre-se de que a linguagem Java é case sensitiva, ou seja, as letras maiúsculas e minúsculas são diferentes. Assim, uma variável **preco** é diferente de **Preco**.

Na linguagem Java existem palavras que não podemos utilizar para nomear as variáveis, classes ou métodos. São as palavras **reservadas**, que possuem significados dentro da programação. A lista completa é apresentada abaixo:

| | | | | |
|---------------------|----------|------------|------------|--------------|
| abstract | continue | for | new | switch |
| assert*** | default | goto* | package | synchronized |
| boolean | do | if | private | this |
| break | double | implements | protected | throw |
| byte | else | import | public | throws |
| case | enum**** | instanceof | return | transient |
| catch | extends | int | short | try |
| char | final | interface | static | void |
| class | finally | long | strictfp** | volatile |
| const* | float | native | super | while |
| PALAVRAS RESERVADAS | | | | |

Figura 2.24 – Palavras reservadas

Fonte: Elaborado pelo autor (2018)

Voltando aos tipos de variáveis, como já adiantamos, existem oito tipos primitivos. Quatro são para armazenar **tipos números inteiros**, positivo ou negativo, sem a parte fracionária.

Abaixo apresentamos os nomes dos quatro tipos primitivos para números inteiros, o tamanho que ocupa na memória e o intervalo de valores que cada tipo consegue armazenar.

| Tabela 3-1 Tipos de número inteiro do Java | | |
|--|----------------------------|--|
| Tipo | Requisito de armazenamento | Intervalo (inclusive) |
| int | 4 bytes | -2.147.483.648 a 2.147.483.647 (um pouco acima de 2 bilhões) |
| short | 2 bytes | -32.768 a 32.767 |
| long | 8 bytes | -9.223.372.036.854.775.808 a 9.223.372.036.854.775.807 |
| byte | 1 byte | -128 a 127 |
| TIPOS PRIMITIVOS INTEIROS | | |

Figura 2.25 – Tipos primitivos inteiros

Fonte: Elaborado pelo autor (2018)

A diferença entre os tipos é o tamanho do número que consegue armazenar, e consequentemente, a quantidade de memória necessária para isso. Na maioria das situações, o tipo **int** é o mais utilizado. Porém, quando for necessário armazenar um número muito grande, precisamos recorrer ao tipo **long**.

Valores de **ponto flutuante** são os números que contêm parte fracionária, ou seja, os números decimais. Para eles existem dois tipos, como apresentados na tabela abaixo:

| Tabela 3-2 Tipos de ponto flutuante | | |
|-------------------------------------|----------------------------|---|
| Tipo | Requisito de armazenamento | Intervalo |
| float | 4 bytes | aprox. +/- 3.40282347E+38F (6–7 dígitos decimais significativos) |
| double | 8 bytes | aprox. +/- 1.797693134862311570E+308 (15 dígitos decimais significativos) |
| TIPOS PRIMITIVOS DE PONTO FLUTUANTE | | |

Figura 2.26 – Tipos primitivos de ponto flutuante

Fonte: Elaborado pelo autor (2018)

O tipo **double** é duas vezes mais preciso que o tipo **float**. Na maioria dos casos é utilizado o tipo **double**.

O tipo **char** é utilizado para armazenar caracteres individuais, como letras, algarismos, sinais de pontuação, entre outros.

O último tipo é o **boolean**, que possui somente dois valores, verdadeiro (**true**) ou falso (**false**). No Java não é possível converter números inteiros em valores booleanos.

2.2.1 Atribuindo valores às variáveis

Depois de declarar uma variável, podemos atribuir um valor a ela com o operador de atribuição, que é definido pelo sinal de igual (=).

Exemplos:

int idade = 10;

double preco = 10.0;

char sexo = 'M';

boolean maiorIdade = false;

Vamos praticar! Crie uma nova classe com o método main, adicione algumas variáveis, atribua valores e imprima o seu conteúdo. Utilize como guia o exemplo abaixo:

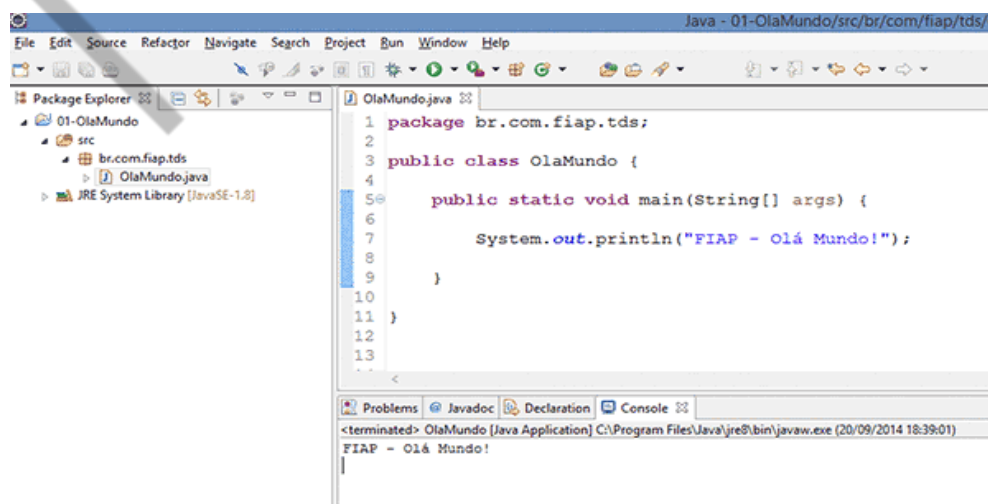


Figura 2.27 – Exemplo tipos primitivos.

Fonte: Elaborado pelo autor (2018)

Teste todos os tipos de dados primitivos.

Muitas vezes, é necessário converter um tipo de dado em outro tipo. A figura abaixo apresenta as conversões possíveis:

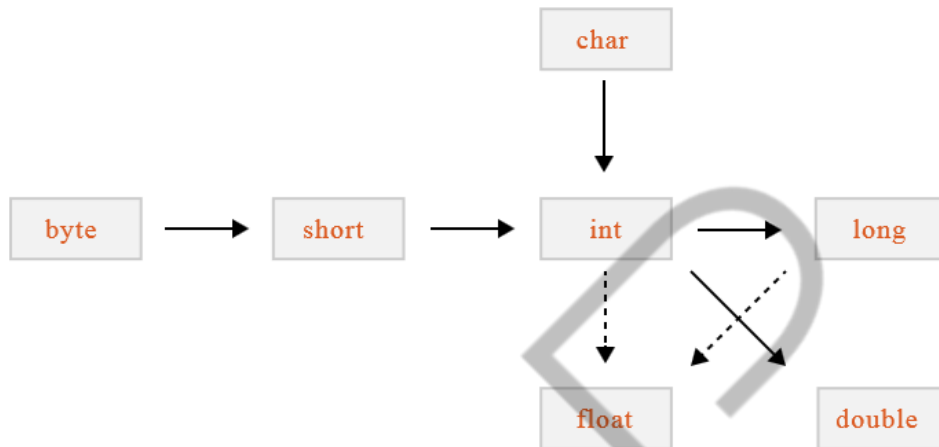


Figura 2.28 – Conversões de tipos de dados

Fonte: Elaborado pelo autor (2018)

As setas sólidas indicam as conversões de tipos onde não se perdem informações, pois o tipo muda de um tamanho menor para um maior. As três setas pontilhadas indicam conversões que podem perder informações. Por exemplo, um número inteiro grande pode ter mais números que um tipo float pode armazenar.

Essas são as conversões que podem ser realizadas automaticamente.

```
Conversoes.java
1 package br.com.fiap.tds;
2
3 public class Conversoes {
4
5     public static void main(String[] args) {
6         int x = 10;
7         double d = x;
8         long l = x;
9         float f = x;
10    }
11 }
```

Figura 2.29 – Conversões entre tipos primitivos

Fonte: Elaborado pelo autor (2018)

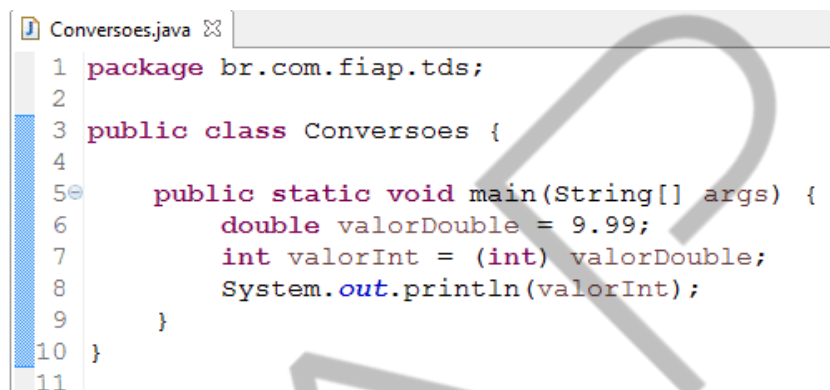
Por outro lado, existem momentos em que não é possível realizar as conversões automáticas. Por exemplo, transformar um double em int. Essa conversão

é viável, porém há risco em perder informações. Esse tipo de conversão é possível por meio de `cast`. A sintaxe é adicionar o tipo que queremos converter entre parênteses:

```
double x = 10;
```

```
int y = (int) x;
```

Exemplo:



```
1 package br.com.fiap.tds;
2
3 public class Conversoes {
4
5     public static void main(String[] args) {
6         double valorDouble = 9.99;
7         int valorInt = (int) valorDouble;
8         System.out.println(valorInt);
9     }
10 }
11
```

Figura 2.30 – Conversão utilizando cast

Fonte: Elaborado pelo autor (2018)

O exemplo acima converte um tipo `double` em tipo `int`, mas neste caso, a variável `x` recebe somente a parte inteira do número, perdendo assim informações.

2.3 Operadores

Operadores são símbolos especiais utilizados para operações matemáticas, instruções de atribuições e comparações lógicas.

Os operadores aritméticos comuns são utilizados para realizar somas, subtrações, multiplicação e divisão. Existe ainda um operador que retorna o resto da divisão de dois números, chamado módulo.

Abaixo apresentamos os cinco operadores aritméticos:

| Operador Java | Operador | Expressão Algébrica | Expressão Java |
|---------------|----------|----------------------|----------------|
| Adição | + | $f + 7$ | $f + 7$ |
| Subtração | - | $p - c$ | $p - c$ |
| Multiplicação | * | Bm | $b * m$ |
| Divisão | / | x/y ou $x \cdot y$ | x/y |
| Resto | % | $r \bmod s$ | $r \% s$ |

Figura 2.31 – Operadores aritméticos

Fonte: Elaborado pelo autor (2018)

Observe que o símbolo asterisco (*) representa uma multiplicação, o sinal de porcentagem (%) retorna o resto da divisão entre dois números.

A divisão entre inteiros produz números inteiros. A parte fracionária é descartada, sem nenhum arredondamento. Por exemplo, o resultado de $7/2$ é 3. O operador módulo (%) fornece o resto da divisão, assim $7\%2$ produz o resultado 1.

Assim como na matemática, no Java podemos realizar várias operações aritméticas de uma vez. As regras de precedência de operadores aplicados na matemática também se aplicam no Java:

- As operações de multiplicação, divisão e módulo são realizadas primeiro. Eles possuem o mesmo nível de precedência, dessa forma, se a expressão matemática possuir dois operadores desses, elas serão aplicadas da esquerda para a direita.
- As operações de adição e subtração são aplicadas em seguida. Elas possuem o mesmo nível de precedência.

Podemos utilizar os parênteses para agrupar as operações matemáticas, da mesma forma que são utilizados em expressões algébricas. Por exemplo, para $x + y$ e depois multiplicado por z escrevemos:

$(x + y) * z;$

Dessa forma, a expressão $x + y$ é realizada primeiro.

2.3.1 Operadores de atribuição

Como já vimos, atribuir valores a uma variável utilizamos o símbolo de igual (=).

A atribuição ocorre sempre após o processamento das expressões do lado direito da atribuição. Por exemplo:

```
int x = 10;
```

```
x = x + 15;
```

Qual valor final de x? Primeiro a variável x foi inicializada com o valor 10. Depois a expressão x + 15 é processada primeiro, retornando o valor 25. Após isso, é realizada a atribuição desse valor na própria variável x.

Para facilitar essa tarefa, que é extremamente comum na programação, existem operadores específicos para esses casos. Assim, o código assim pode ser escrito dessa forma:

```
int x = 10;
```

```
x +=15;
```

Os outros operadores aritméticos básicos: multiplicação, divisão e subtração também possuem esses operadores de atribuição com a operação matemática.

Abaixo apresentamos os operadores de atribuição e as expressões às quais são equivalentes.

| Expressão | Significado |
|-----------|-------------|
| x += y | x = + y |
| x -= y | x = x - y |
| x *= y | x = x * y |
| x /= y | x = x/y |

Figura 2.32 – Operadores de atribuição

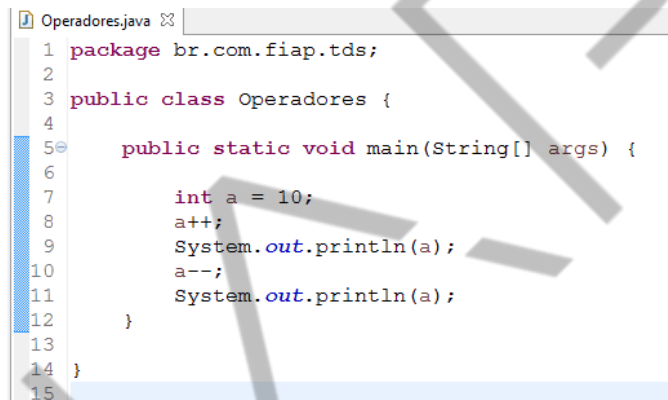
Fonte: Elaborado pelo autor (2018)

2.3.2 Operadores de incremento e decremento

Para aumentar ou diminuir o valor de uma variável em uma unidade podemos utilizar os operadores de incremento e decremento.

Por exemplo, para adicionar 1 à variável `x`, utilizamos `x++`; É o mesmo resultado da expressão `x = x + 1`; Para subtrair 1 de `x`, podemos escrever `x--`; que produz o mesmo resultado de `x = x - 1`;

Assim, quais valores serão impressos na execução do código abaixo?



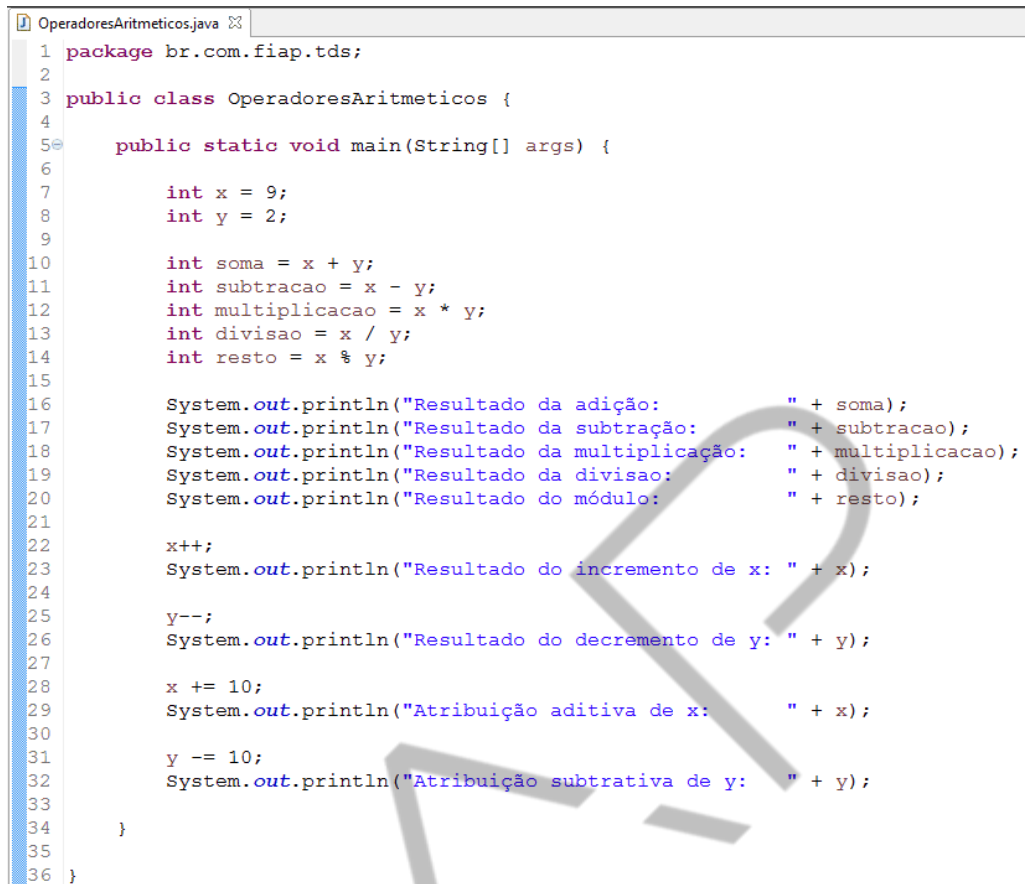
```
Operadores.java
1 package br.com.fiap.tds;
2
3 public class Operadores {
4
5     public static void main(String[] args) {
6
7         int a = 10;
8         a++;
9         System.out.println(a);
10        a--;
11        System.out.println(a);
12    }
13
14 }
15
```

Figura 2.33 – Exemplo de utilização dos operadores de incremento e decremento

Fonte: Elaborado pelo autor (2018)

A resposta é 11 e 10. Pois primeiro acrescentamos 1 à variável `a` e imprimimos. Depois decrementamos em 1 e imprimimos.

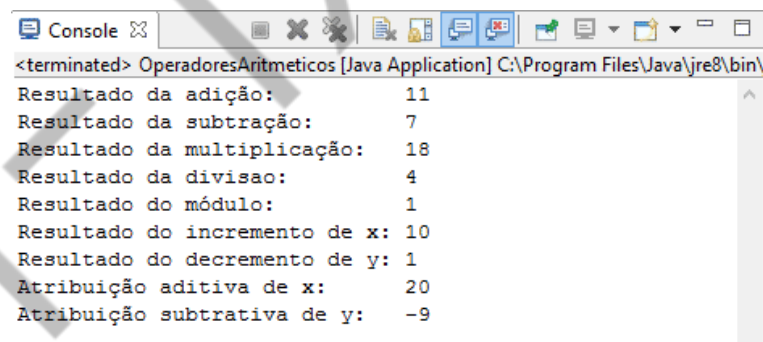
A classe abaixo apresenta um resumo de todos os operadores visto até o momento:



```
1 package br.com.fiap.tds;
2
3 public class OperadoresAritmeticos {
4
5     public static void main(String[] args) {
6
7         int x = 9;
8         int y = 2;
9
10        int soma = x + y;
11        int subtracao = x - y;
12        int multiplicacao = x * y;
13        int divisao = x / y;
14        int resto = x % y;
15
16        System.out.println("Resultado da adição:      " + soma);
17        System.out.println("Resultado da subtração:    " + subtracao);
18        System.out.println("Resultado da multiplicação:  " + multiplicacao);
19        System.out.println("Resultado da divisao:      " + divisao);
20        System.out.println("Resultado do módulo:      " + resto);
21
22        x++;
23        System.out.println("Resultado do incremento de x: " + x);
24
25        y--;
26        System.out.println("Resultado do decremento de y: " + y);
27
28        x += 10;
29        System.out.println("Atribuição aditiva de x:      " + x);
30
31        y -= 10;
32        System.out.println("Atribuição subtrativa de y:   " + y);
33
34    }
35
36 }
```

Figura 2.34 – Operadores aritméticos e de atribuição.

Fonte: Elaborado pelo autor (2018)



```
<terminated> OperadoresAritmeticos [Java Application] C:\Program Files\Java\jre8\bin\
Resultado da adição:      11
Resultado da subtração:    7
Resultado da multiplicação: 18
Resultado da divisao:      4
Resultado do módulo:      1
Resultado do incremento de x: 10
Resultado do decremento de y: 1
Atribuição aditiva de x:  20
Atribuição subtrativa de y: -9
```

Figura 2.35 – Resultado da execução

Fonte: Elaborado pelo autor (2018)

Agora é a sua vez! Crie uma nova classe e realize testes com os operadores aritméticos e de atribuição, conforme o exemplo anterior.

2.3.3 Operadores de igualdade e relacionais

Para realizar comparações entre variáveis, variáveis e valores ou outros tipos de informações, são utilizados operadores para formar expressões que retornam um valor booleano verdadeiro (true) ou falso (false). Abaixo apresentamos os operadores de comparação do Java:

| Operador de igualdade ou relacional algébrico padrão | Operador de igualdade ou relacional java | Exemplo de condição em Java | Significado da condição em Java |
|--|--|-----------------------------|---------------------------------|
| Operadores de igualdade | | | |
| = | == | x == y | x é igual a y |
| ? | != | x != y | x é diferente de y |
| Operadores relacionais | | | |
| | | y | x é maior que y |
| < | < | x < y | x é menor que y |
| >_ | >= | x >= y | x é maior que ou igual a y |
| <_ | <= | x <= y | x é menor que ou igual a y |
| OPERADORES DE IGUALDADE E RELACIONAIS | | | |

Figura 2.36 – Tabela Operadores de igualdade e relacionais

Fonte: Elaborado pelo autor (2018)

Exemplo:

```
int idade = 10;
```

```
boolean maioridade = idade > 18;
```

No exemplo acima, a variável idade é inicializada com o valor 10. Depois é criada uma variável booleana chamada maioridade, que recebe o resultado da expressão **idade > 18**, ou seja, o resultado do teste da idade ser maior que 18. Neste caso **false**.

2.3.4 Operadores lógicos

Operadores lógicos são utilizados para formar expressões de comparação mais complexas, que possuem mais de um termo para comparação. Essas expressões resultam em valores booleanos.

Os operadores utilizados para combinar as comparações são **AND**, **OR**, **XOR** e **NOT**.

O operador lógico **AND (e)** é representado pelo símbolo **&&**. Quando duas expressões booleanas utilizam o operador **&&**, o resultado final é verdadeiro (true) somente quando as duas expressões forem verdadeiras. Exemplo:

```
boolean precisaVotar = idade > 18 && idade < 70;
```

A expressão combina duas comparações: a primeira verifica se a variável *idade* é maior que 18 (*idade > 18*) e a segunda, testa se a *idade* é menor que 70 (*idade < 70*). Se as duas expressões forem verdadeiras, a variável **precisaVotar** recebe verdadeiro (true), em qualquer outra situação o valor será falso (false).

Para a combinação **OR**, o operador **||** é utilizado. Essas expressões combinadas retornam como verdadeiro caso uma das expressões seja verdadeira.

Observe o exemplo abaixo:

```
boolean teste = x < 10 || x > 50;
```

A expressão acima também combina duas comparações. A diferença é que ela foi combinada com o operador **OR (ou)**, dessa forma, a variável *teste* receberá o valor verdadeiro se qualquer uma das duas expressões forem verdadeiras. Ela receberá o valor falso somente se as duas expressões forem falsas.

O operador **XOR** é representado pelo símbolo **^**. A combinação resulta em um valor verdadeiro (true) somente se as duas expressões tiverem valores opostos. Se ambos forem verdadeiros, ou ambos serem falsos, o resultado da combinação será false.

O operador **NOT** utiliza o operador lógico!

Ela reverte o valor da expressão booleana. Por exemplo, `idade > 18` retorna um valor verdadeiro se a idade for maior que 18. A expressão `!(idade > 18)` retornará um valor false, caso a idade seja maior que 18:

```
!(idade > 18)
```

Se a idade for 20, a expressão retorna true: `!(true) ->` invertendo o valor `(!)` -> o resultado é false.

2.4 Fluxo de controle e escopo de bloco

A maioria das linguagens possui um fluxo de controle como as instruções de seleção e loops. Primeiramente vamos estudar as instruções condicionais (seleção).

Antes de entrar na estrutura de seleção, vale relembrar os blocos.

Vimos que as chaves `{ }` delimitam o início e o fim de classes e métodos. Neste caso, eles delimitam os blocos de código que pertencem à classe e ao método. Os blocos podem ser aninhados dentro de outro bloco, como, por exemplo, a classe que possui um método.

A instrução condicional no Java é realizada pela palavra reservada **if** e tem a seguinte sintaxe:

if (condição) instrução.

A condição deve ser colocada entre os parênteses e podem conter várias expressões combinadas com os operadores lógicos.

Quando a condição for verdadeira, o que estiver dentro do bloco **if** será executado. Para determinar o bloco de código que pertence ao **if** as chaves são utilizadas:

```
if (condição) {  
    instrução 1;  
    instrução 2;  
}
```

Exemplo:

```
if(x > 10){  
  
    System.out.println("X é maior que 10");  
  
    System.out.println(" ***** FIM ***** ");  
  
}
```

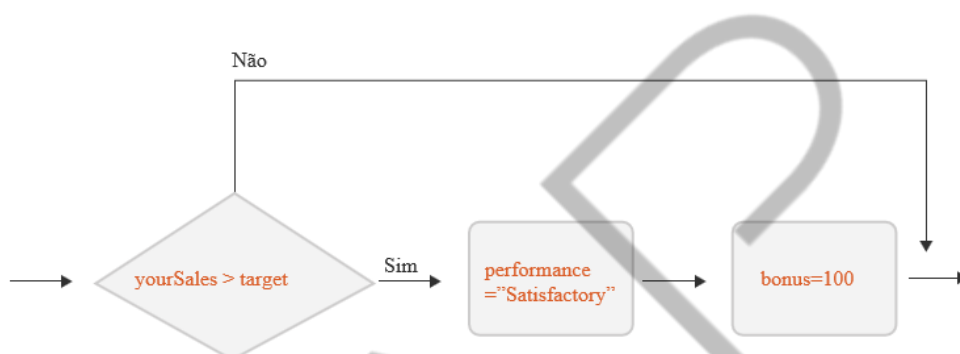


Figura 2.37 – Fluxo para a instrução if

Fonte: Elaborado pelo autor (2018)

Podemos utilizar também o comando else (que é opcional) junto do if. Assim, se a condição do if for false, o bloco de código do else será executado.

```
if (condição) {  
    Instrução;  
}else{  
    Instrução;  
}
```

Exemplo:

```
if (x > 10){  
  
    System.out.println("X é maior que 10");  
  
} else {  
  
    System.out.println("X é menor ou igual a 10");  
  
}
```

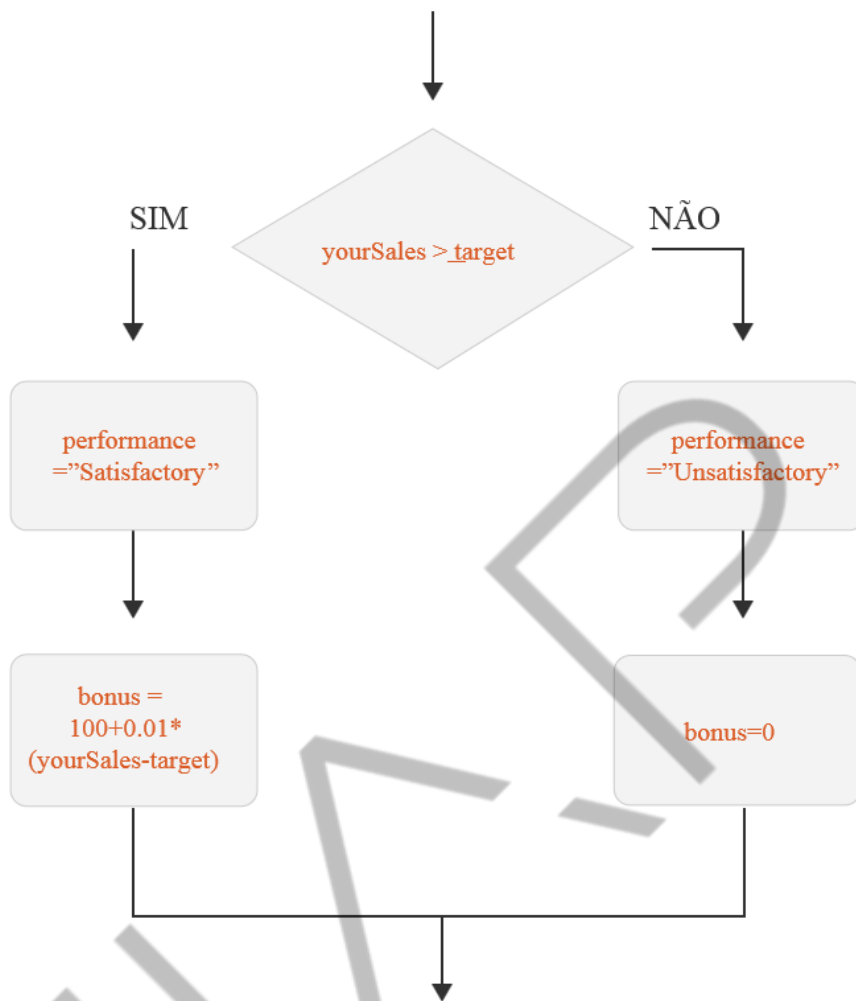


Figura 2.38 – Fluxo para a instrução if – else
Fonte: Elaborado pelo autor (2018)

É possível utilizar uma instrução de seleção dentro de outra instrução de seleção. Como por exemplo:

```
if(x > 10){  
    System.out.println("X é maior que 10");  
} else if(x == 10) {  
    System.out.println("X é igual a 10");  
} else {  
    System.out.println("X é menor que 10");  
}
```


Dessa forma, um if fica dentro de outro if (neste caso do else). Sempre um else pertence ao if que estiver mais próximo dele. No exemplo, primeiro é verificado se o x é maior que 10, senão for, verifica-se o x é igual a 10, se não for imprime o valor “X é menor que 10”.

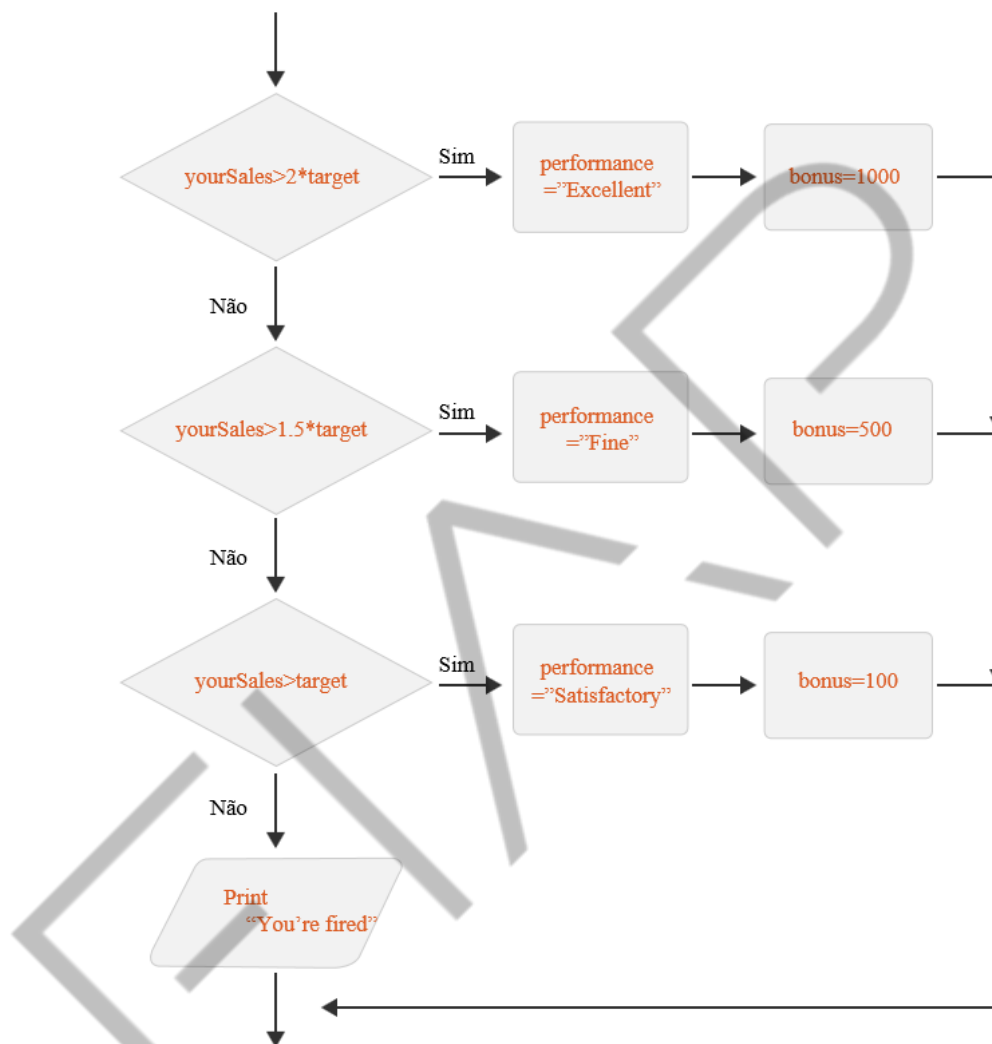


Figura 2.39 – Fluxo para if – else aninhados

Fonte: Elaborado pelo autor (2018)

2.5 Entrada e saída de dados

Já vimos como podemos exibir informações para o usuário com a instrução **System.out.println()**; existe também a instrução **System.out.print()**; sendo que a diferença entre eles é a quebra de linha. O **println** pula uma linha no final, enquanto que o **print** somente imprime a informação e continua na mesma linha.

Agora, para ler uma informação inserida pelo usuário, precisamos da ajuda da classe Scanner.

Scanner sc = new Scanner(System.in);

Assim, foi declarada uma variável do tipo Scanner com o nome sc (veremos com detalhes o operador new e construtores nos próximos capítulos).

Um scanner permite a leitura de dados que podem ser provenientes de várias origens, como um arquivo do disco ou informações digitadas pelo usuário. Dessa forma, o valor **System.in** especifica que queremos ler os valores digitados pelo usuário.

Após a declaração, podemos utilizar os métodos da classe Scanner para ler as informações. Existem vários métodos, que são utilizados para ler cada um dos tipos de dados possíveis:

Scanner sc = new Scanner(System.in);

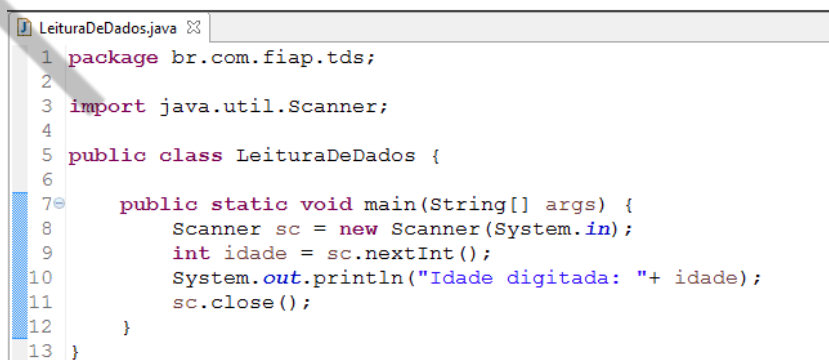
int idade = sc.nextInt();

double preco = sc.nextDouble();

No exemplo acima, o método nextInt() lê um número inteiro. E o método nextDouble() lê um número de ponto flutuante.

A classe Scanner está definida no pacote **java.util**.

Sempre que precisamos utilizar uma classe que estão em pacotes diferentes e não são do pacote básico (java.lang) é necessário utilizar a instrução **import**.



```
LeituraDeDados.java
1 package br.com.fiap.tds;
2
3 import java.util.Scanner;
4
5 public class LeituraDeDados {
6
7     public static void main(String[] args) {
8         Scanner sc = new Scanner(System.in);
9         int idade = sc.nextInt();
10        System.out.println("Idade digitada: " + idade);
11        sc.close();
12    }
13 }
```

Figura 2.40 – Exemplo de leitura de dados

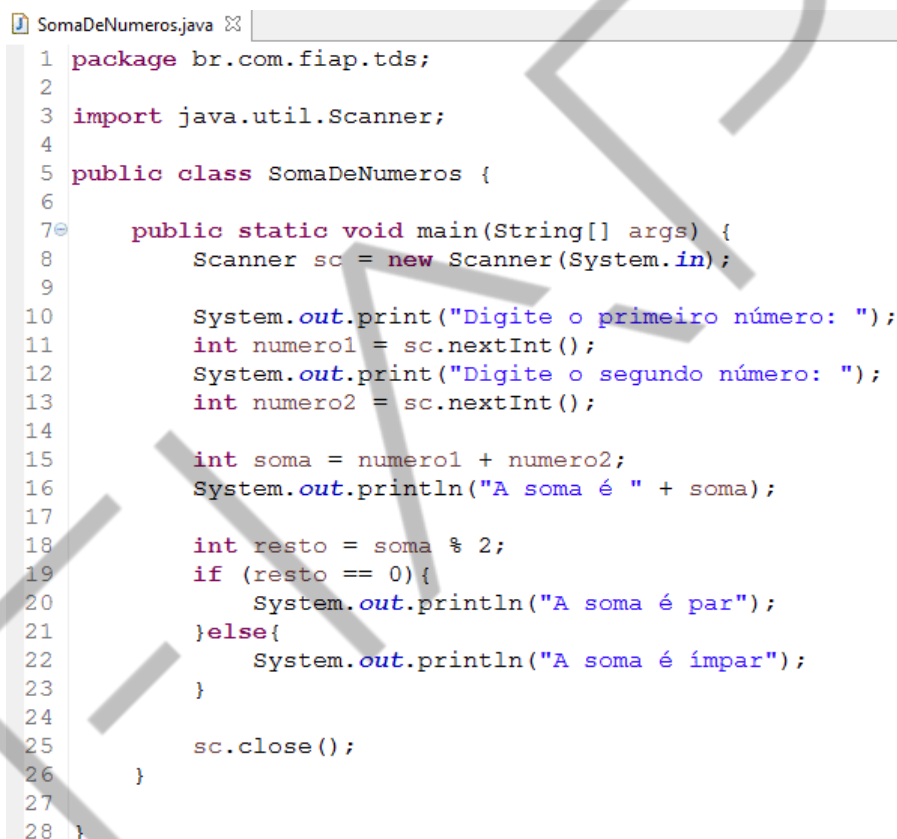
Fonte: Elaborado pelo autor (2018)

Observe a instrução **import** logo após a definição do pacote da classe. O **import** tem o nome do **pacote** mais o **nome da classe** que queremos utilizar na classe **LeituraDeDados**.

Na linha 9 é realizada a leitura de dados. Após, é feita a impressão do valor digitado.

Na linha 11 estamos fechando o scanner, pois não vamos mais utilizá-lo.

No exemplo abaixo, o programa lê dois números e realiza a soma entre eles. Depois verifica se o número é somado é par.

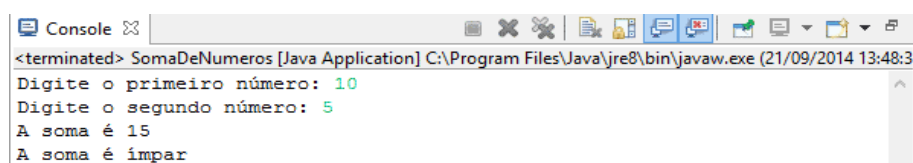


```
1 package br.com.fiap.tds;
2
3 import java.util.Scanner;
4
5 public class SomaDeNumeros {
6
7     public static void main(String[] args) {
8         Scanner sc = new Scanner(System.in);
9
10        System.out.print("Digite o primeiro número: ");
11        int numero1 = sc.nextInt();
12        System.out.print("Digite o segundo número: ");
13        int numero2 = sc.nextInt();
14
15        int soma = numero1 + numero2;
16        System.out.println("A soma é " + soma);
17
18        int resto = soma % 2;
19        if (resto == 0) {
20            System.out.println("A soma é par");
21        } else {
22            System.out.println("A soma é ímpar");
23        }
24
25        sc.close();
26    }
27
28 }
```

Figura 2.41 – Exemplo de leitura de dados.

Fonte: Elaborado pelo autor (2018)

Resultado da execução:



```
<terminated> SomaDeNumeros [Java Application] C:\Program Files\Java\jre8\bin\javaw.exe (21/09/2014 13:48:3
Digite o primeiro número: 10
Digite o segundo número: 5
A soma é 15
A soma é ímpar
```

Figura 2.42 – Exemplo de leitura de dados.

Fonte: Elaborado pelo autor (2018)

Vamos praticar! Crie um programa Java que calcule o IMC e exiba se ele está no peso ideal ou não.

O programa deve receber dois valores: altura e peso. O resultado do IMC é calculado através da expressão: $\text{peso} / (\text{altura} * \text{altura})$;

Caso o imc esteja entre 18.5 e 25, informe que o peso é ideal, caso contrário, informe que está fora do peso normal.

EMENDADO

REFERÊNCIAS

BARNES, David J. **Programação Orientada a Objetos com Java**: Uma introdução Prática Utilizando Blue J. São Paulo: Pearson, 2004.

CADENHEAD, Rogers; LEMAY, Laura. **Aprenda em 21 dias Java 2 Professional Reference**. 5.ed. Rio de Janeiro: Elsevier, 2003.

DEITEL, Paul; DEITEL, Harvey. **Java Como Programar**. 8.ed. São Paulo. Pearson, 2010.

HORSTMANN, Cay; CORNELL, Gary. **Core Java**. Volume I Fundamentos. 8.ed. São Paulo: Pearson, 2009.

SIERRA, Kathy; BATES, Bert. **Use a cabeça! Java**. Rio de Janeiro: Alta Books, 2010.