



**Prof. MSc. Marcos Alexandruk**

**E-mail: [alexandruk@uni9.pro.br](mailto:alexandruk@uni9.pro.br)**

**<https://github.com/alexandruk/poo>**

# **Introdução ao Java**

## **aula 01**

## Objetivo

Esta aula apresenta uma breve introdução ao Java, sua história e sua importância na atualidade.

Apresenta também evolução da linguagem Java.

Você terminará essa aula com o seu primeiro programa em Java funcionando: O clássico Hello World.



## Breve história

**Java** é uma linguagem que se originou em **1991** em um projeto da **Sun Microsystems**.

Começou com **James Gosling** e o **Projeto Green** que visava a criação de tecnologias para que diferentes dispositivos eletrônicos pudessem se comunicar.

Java pode rodar em praticamente qualquer dispositivo, pois não depende da plataforma.

O Projeto Green criou o **Star Seven**, um controle remoto universal com uma tela interativa.

O software deste controle foi desenvolvido com a linguagem de programação **Oak**.

Questões legais obrigaram a equipe a alterar o nome. Portanto, em **maio de 1995**, surgiu o **Java**, uma evolução da linguagem Oak.

A equipe escolheu Java, com base no nome de uma cafeteria que servia um café com origem em Java (uma ilha da Indonésia). Por isso o ícone do Java é uma xícara de café.

Em 2009, a Oracle adquiriu a Sun Microsystems por US\$ 7,4 milhões.



## Java: linguagem multiplataforma

Java é uma das linguagens mais populares do mundo e sua portabilidade é um de seus pontos mais fortes.

Portabilidade é a capacidade de portar um software para diferentes plataformas, com o mínimo de esforço.

A maior parte das linguagens de programação que precisam de um compilador específico para cada plataforma.

Java possui um compilador único, portanto não é uma linguagem que depende de uma plataforma específica para sua execução.



## Java: linguagem multiplataforma

Os binários compilados da linguagem são executados na **JVM (Java Virtual Machine)**, por isso o mesmo código Java pode ser executado em praticamente qualquer dispositivo com uma JVM.

Existem JVMs para quase todas as plataformas, desde eletrodomésticos até computadores quânticos.

Java é uma linguagem proprietária da Oracle, mas de uso livre e gratuito.

Alguns afirmam que Java é uma linguagem lenta, porque não é executada diretamente no Sistema Operacional (as instruções do Sistema Operacional são geradas pela JVM), mas a tecnologia Java evoluiu tanto que o processo de execução é quase transparente.



## Versões do Java

O versionamento da linguagem Java segue um padrão bastante interessante.

O Java está sempre na versão 1.

Desde 1995 o Java está na versão 1 e as evoluções da linguagem são tratadas como “subversões”. Para referência, usa-se normalmente a subversão do Java, por exemplo, a versão 1.8, é referenciada como versão 8 do Java.

Em 2013, a Oracle criou um novo esquema de versionamento do Java para identificar uma versão do tipo CPU (Critical Patch Update) e uma versão do tipo LFR (Limited Feature Release).

- **CPU - Critical Patch Update:** Atualizações criadas com correções de segurança importantes.
- **LFR - Limited Feature Release:** Atualizações com correções de performance, funcionalidades e recursos.



## Versões do Java

É importante manter o Java atualizado.

Algumas atualizações retiram ou substituem métodos prontos em uso.

A grande vantagem de usar uma IDE (Integrated Development Environment ou Ambiente de Desenvolvimento Integrado) para desenvolvimento Java é que ela lhe avisará se as funções em uso já estão obsoletas (já foram retiradas) ou se ficarão obsoletas na próxima versão, sugerindo outra forma de chamar algum método.





## Ranking das linguagens de programação (IEEE)

Posição	2015	2020
1	Java	Python
2	C	Java
3	C++	C
4	Python	C++
5	C#	JavaScript
6	R	C#
7	PHP	Go
8	JavaScript	R
9	Ruby	Ruby
10	Matlab	Dart



## **JRE - Java Runtime Environment (ambiente de execução Java)**

A JRE contém as bibliotecas que são responsáveis pela execução das aplicações na JVM. A JRE é composta pela JVM, bibliotecas e componentes necessários para essa execução.

## **JVM - Java Virtual Machine (máquina virtual Java)**

A JVM é responsável por executar instruções compiladas em Java. A JVM é um ambiente de computação virtualizado (abstrato) que executa e gerencia os processos Java, a alocação de memória e recursos de CPU etc. É a JVM quem abstrai a execução do código compilado para o equipamento onde será executada a aplicação.

## **JDK - Java Development Kit (kit de desenvolvimento Java)**

Kit de Desenvolvimento Java que será instalado para desenvolver aplicações em Java. O JDK possui todas as ferramentas necessárias para compilação de código Java para a JRE. O JDK possui bibliotecas Java para diferentes tipos de desenvolvimento.



## **JSE - Java Standard Edition (edição padrão do Java)**

É a edição essencial do Java e já contém a JDK, JVM e JRE. Trabalharemos com essa edição do Java, que é capaz de criar qualquer aplicação para ser executada em um computador.

## **JEE - Java Enterprise Edition (edição corporativa do Java)**

Edição do Java para desenvolver aplicações corporativas de grande porte e que rodam em rede. Inclui bibliotecas para Web Services, desenvolvimento WEB, JSP (Java Server Pages), JSF (Java Server Faces), JavaBeans etc.

## **JME - Java Micro Edition (edição micro do Java)**

Esta edição do Java acompanha bibliotecas que permitem o desenvolvimento para sistemas embarcados que possuem uma JVM (carros, eletrodomésticos, celulares etc.). Não confundir com desenvolvimento para Android ou iOS (Apple), pois para estes sistemas operacionais existem ferramentas de desenvolvimento específicas de cada fabricante.



## Extensões dos arquivos Java

**.java** - Arquivos de código fonte Java.

**.class** - Arquivos compilados Java (bytecodes).

**.jar** - Pacote de arquivos **.class**. Possui informações de execução do projeto. É possível exportar o projeto compilado para um único arquivo **.jar** e este será o executável.



## IDE - Integrated Development Environment

O Ambiente de Desenvolvimento Integrado (IDE, em inglês) Fornece um enorme conjunto de ferramentas de desenvolvimento que aumentam a produtividade no processo de desenvolvimento. A IDE possui ferramentas de edição de código, implementa as bibliotecas, compila, executa, gera executáveis, depura código em tempo real etc.

Existem diversas IDEs disponíveis no mercado, como:

- **Netbeans**
- Eclipse
- BlueJ
- JCreator
- Visual Studio Code

## Java JDK e Netbeans

## Softwares

### Java JDK 14

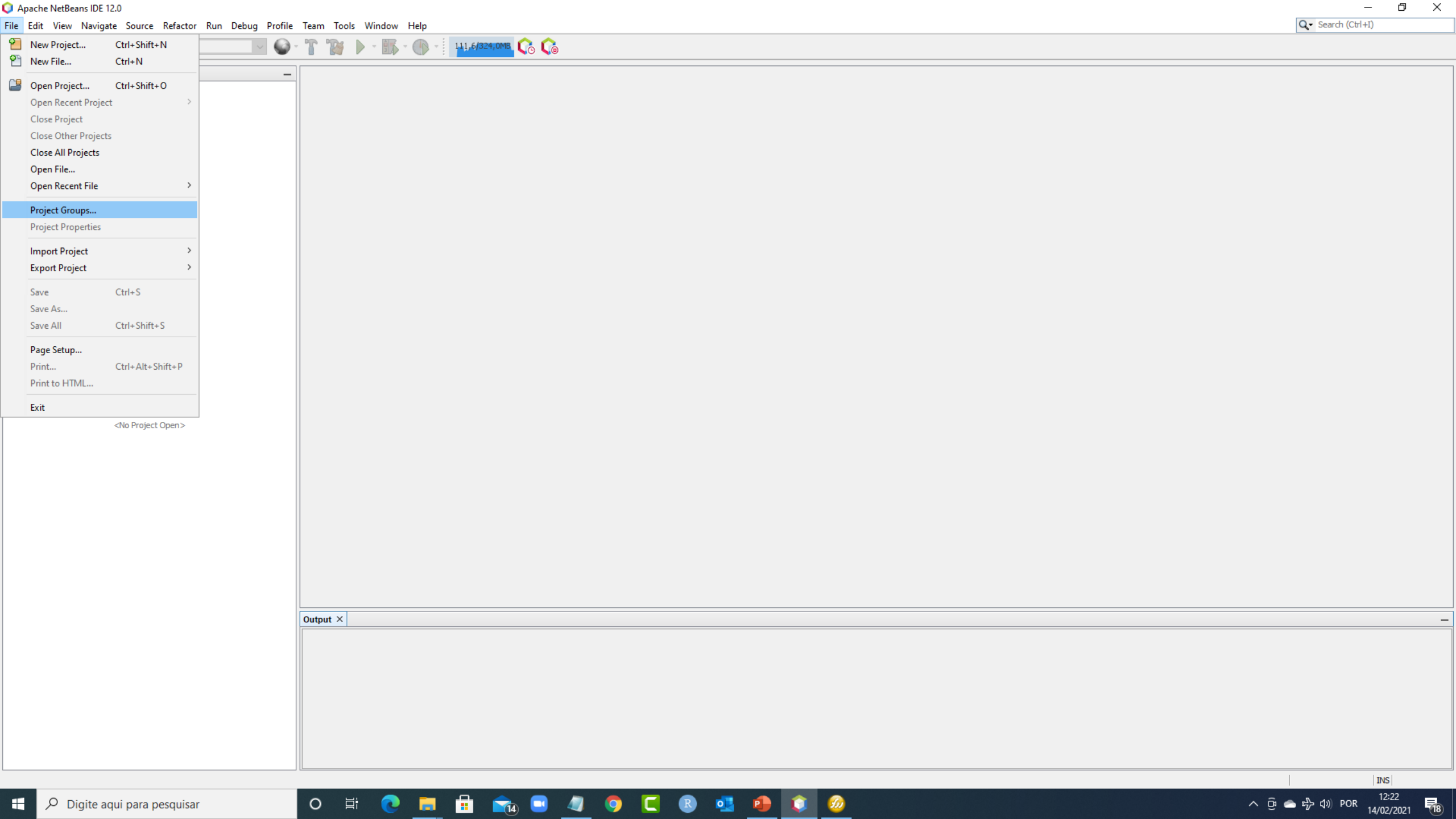
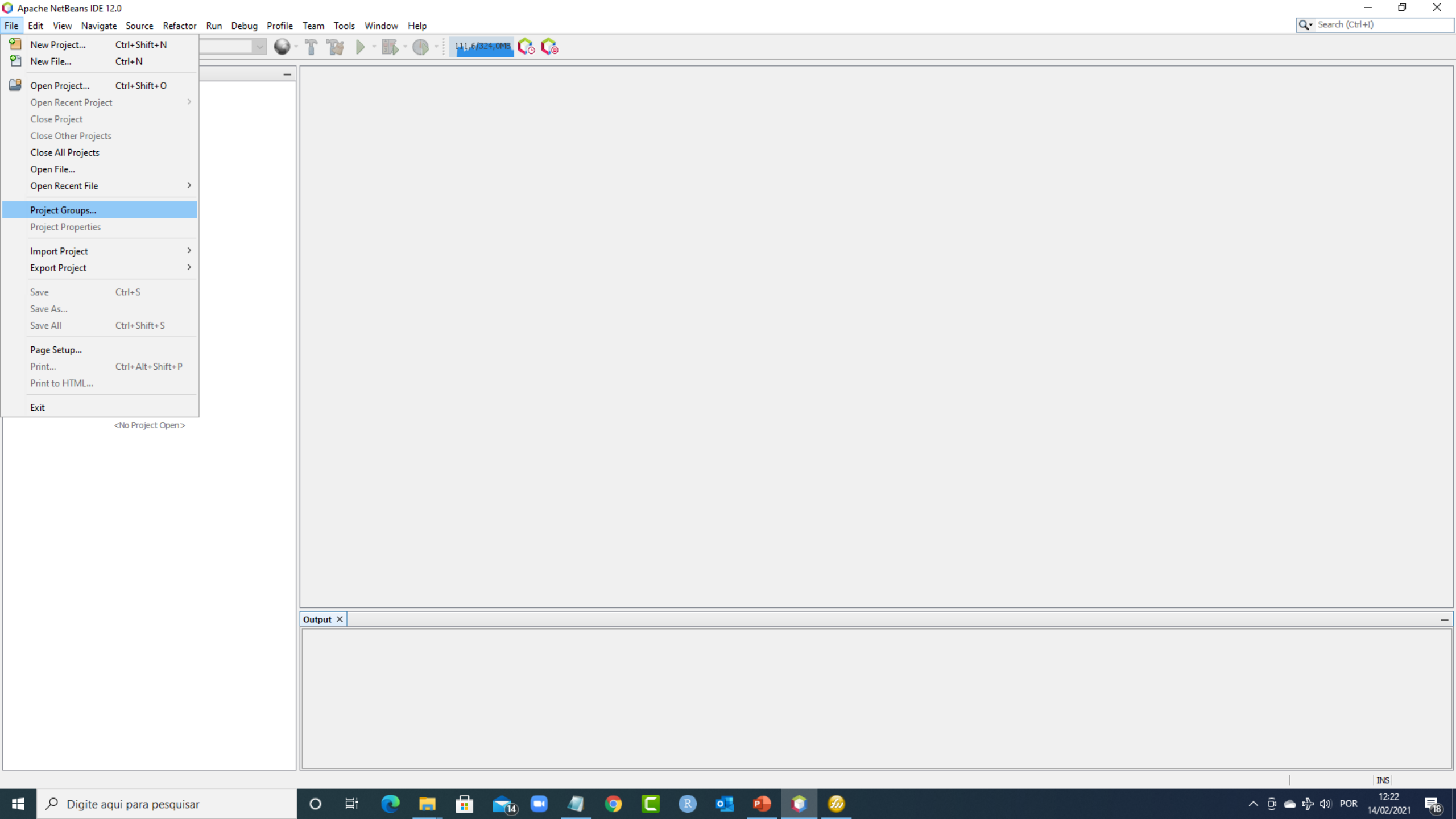
<https://www.oracle.com/br/java/technologies/javase/jdk14-archive-downloads.html>

### Netbeans 12.0

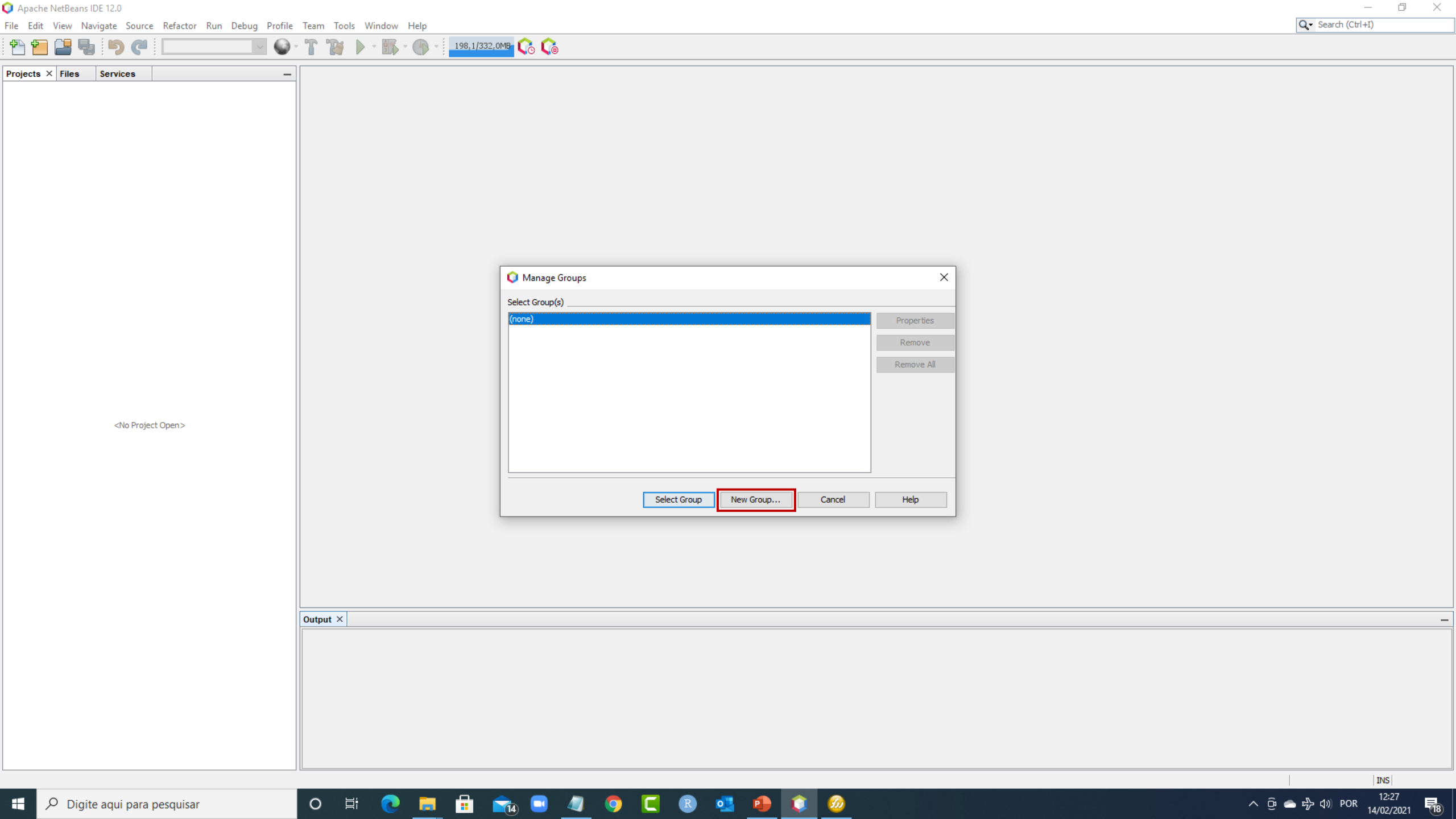
<https://netbeans.apache.org/download/nb120/nb120.html>

Necessário criar uma conta (gratuita) na Oracle.

Para criar a conta deverá informar um e-mail válido e escolher uma senha.









Projects x Files Services

&lt;No Project Open&gt;

**Create New Group**

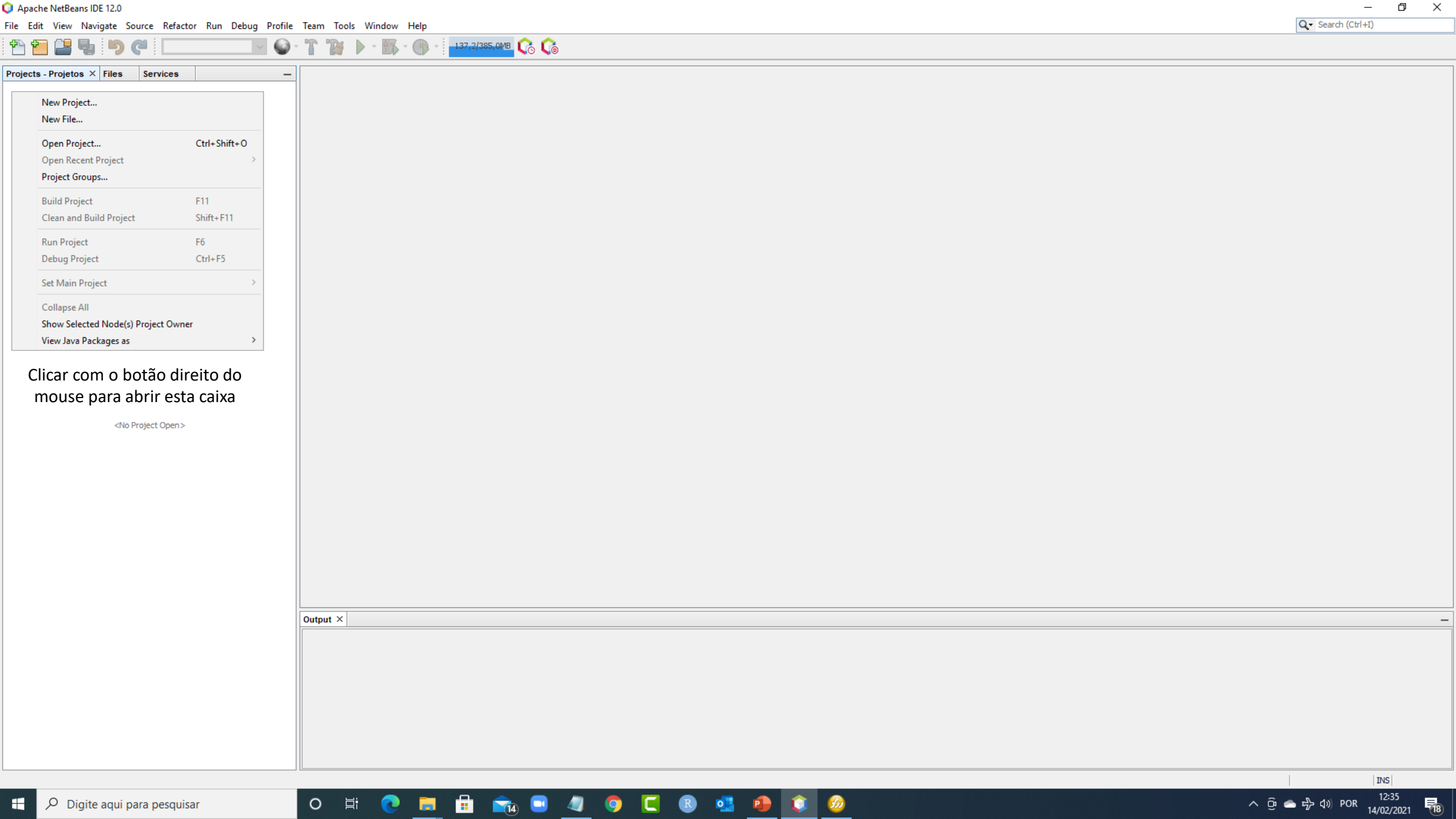
Name:

☐ **Free Group**  
Contains any projects you like. Can be updated manually or automatically.  
☒ Use Currently Open Projects  
☒ Automatically Save Project List

☐ **Project and All Required Projects**  
Contains a master project and all projects it requires, recursively.  
Master Project:

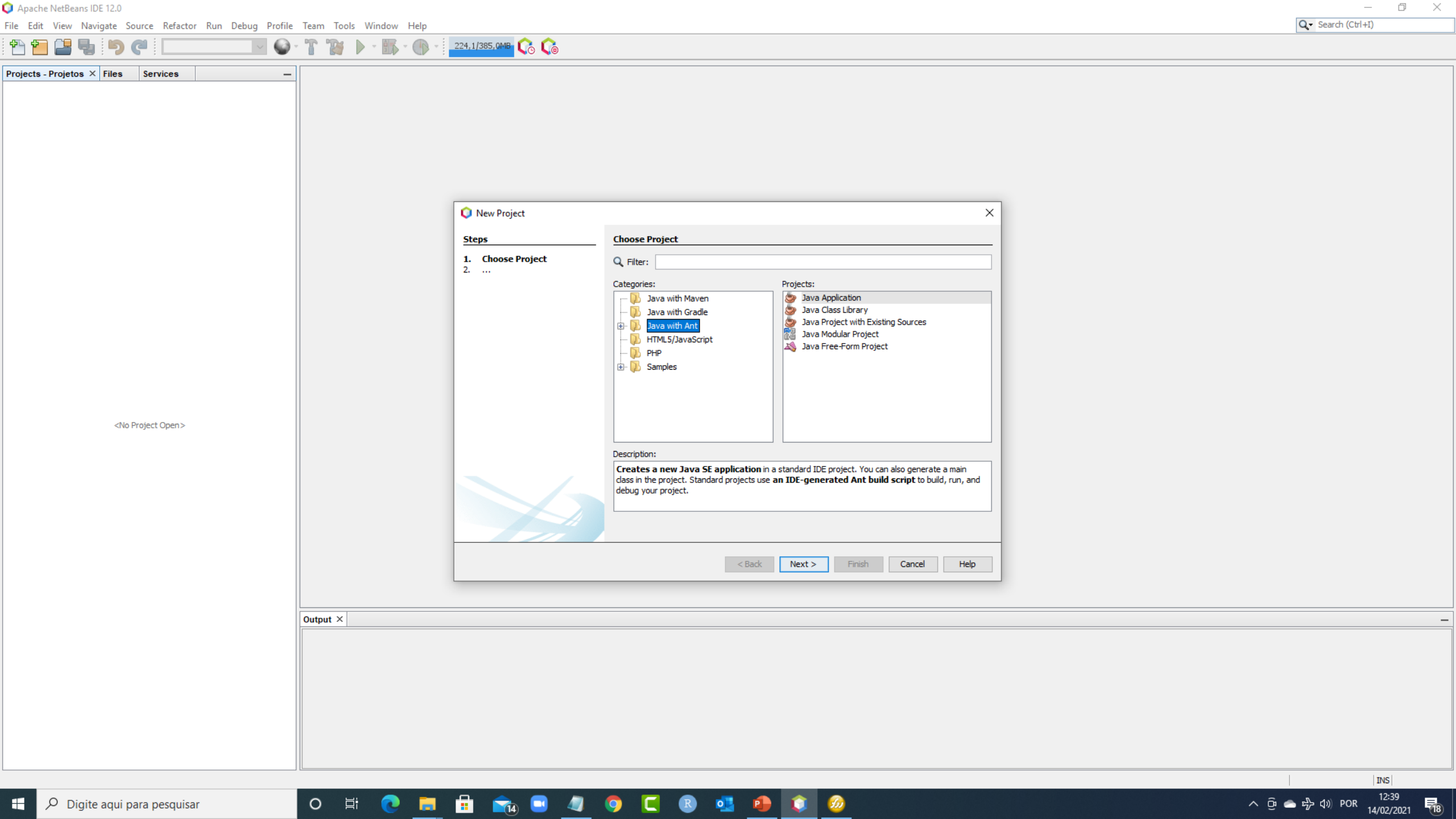
☒ **Folder of Projects**  
Contains any projects found beneath a given folder on disk.  
Folder:

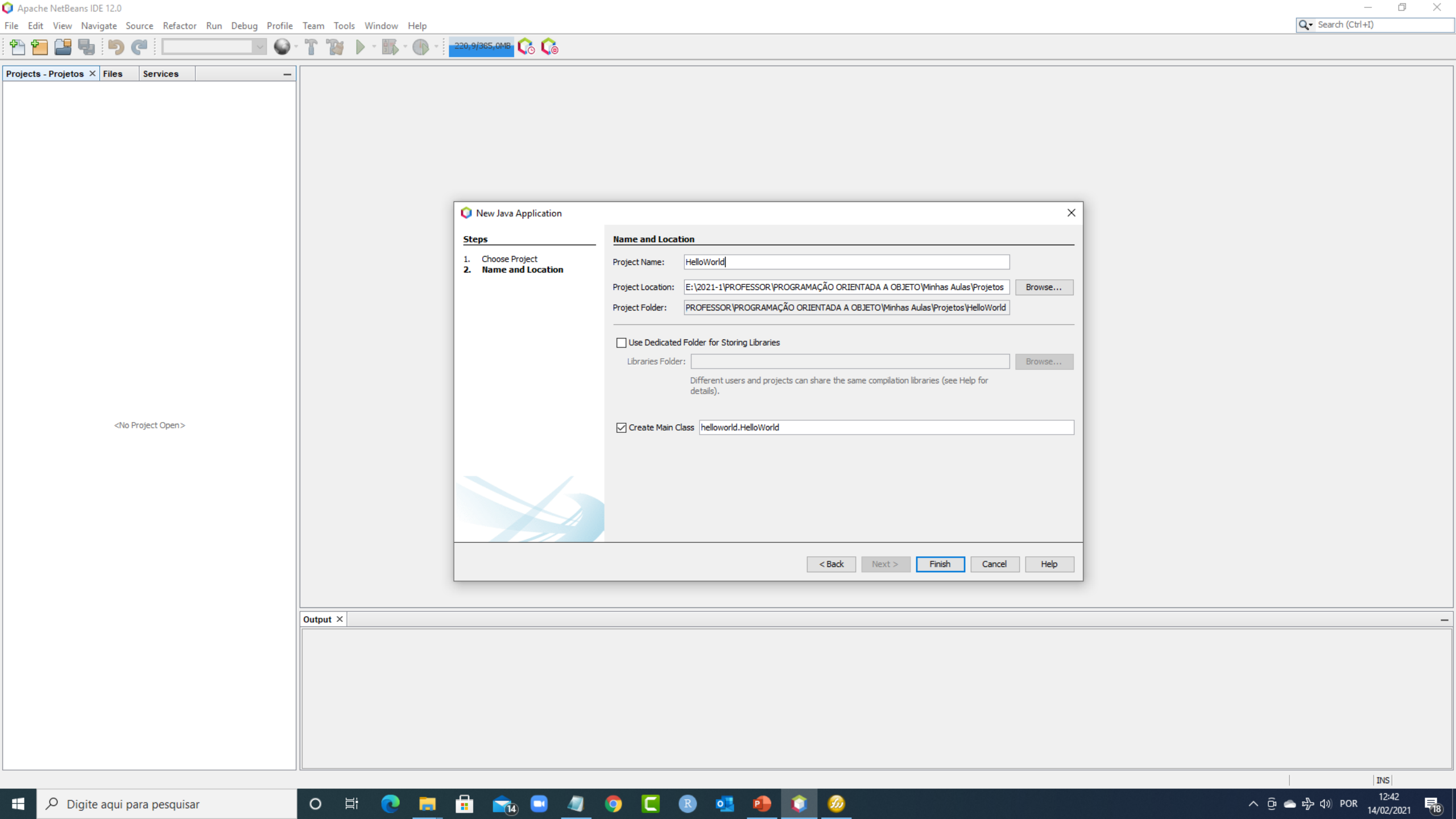
Output x

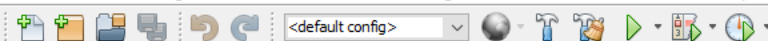


Clicar com o botão direito do mouse para abrir esta caixa

<No Project Open>







Projects - Projetos | Files | Services

Source Packages

- helloworld
  - HelloWorld.java

Libraries

HelloWorld.java - Navigator

Members

<empty>

HelloWorld

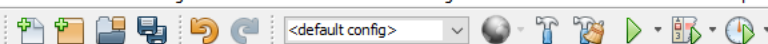
- HelloWorld()
- main(String[] args)

Toolbar icons for development and testing.

Source | History

```
1  /**
2   * To change this license header, choose License Headers in Project Properties.
3   * To change this template file, choose Tools | Templates
4   * and open the template in the editor.
5   */
6   package helloworld;
7
8   /**
9   *
10  * @author malex
11  */
12  public class HelloWorld {
13
14      /**
15       * @param args the command line arguments
16       */
17      public static void main(String[] args) {
18          // TODO code application logic here
19      }
20
21  }
22
```

Output



Projects - Projetos | Files | Services

HelloWorld

- Source Packages
  - helloworld
    - HelloWorld.java
- Test Packages
- Libraries
- Test Libraries

HelloWorld.java - Navigator

Members

<empty>

HelloWorld

- HelloWorld()
- main(String[] args)

Toolbar icons for file operations, navigation, and development tools.

HelloWorld.java

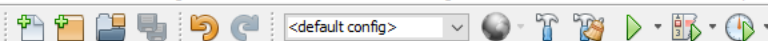
Source | History

```
1  /*
2  * To change this license header, choose License Headers in Project Properties.
3  * To change this template file, choose Tools | Templates
4  * and open the template in the editor.
5  */
6  package helloworld;
7
8  /**
9   *
10  * @author malex
11  */
12  public class HelloWorld {
13
14      /**
15       * @param args the command line arguments
16       */
17      public static void main(String[] args) {
18          System.out.println("hello world");
19      }
20
21  }
22
```

Output







Projects - Projetos | Files | Services

Source Packages  
helloworld  
HelloWorld.java

Test Packages  
Libraries  
Test Libraries

HelloWorld.java - Navigator

Members

Members: <empty>

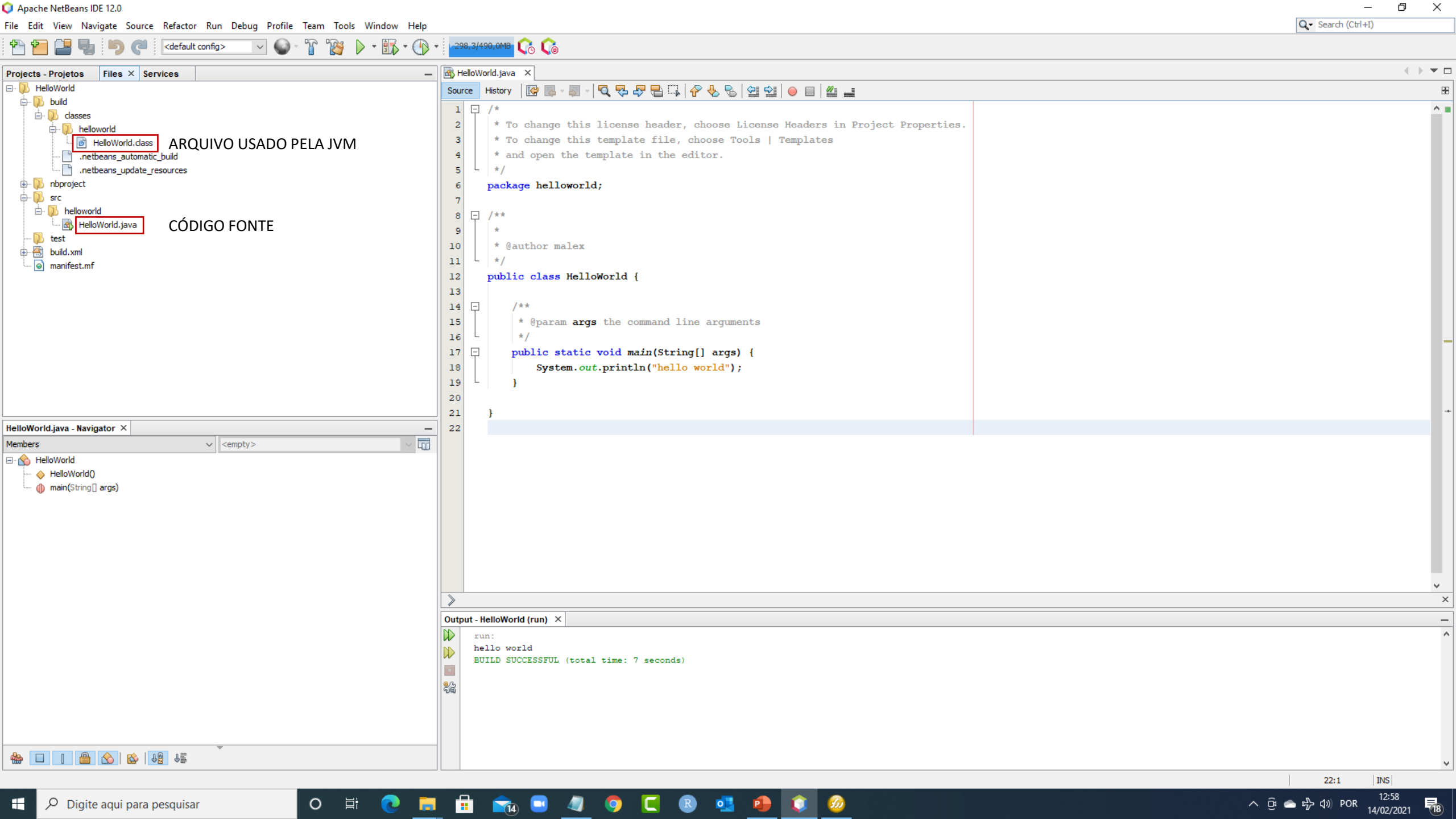
Members:  
HelloWorld  
HelloWorld()  
main(String[] args)

Source | History

```
1  /*
2  * To change this license header, choose License Headers in Project Properties.
3  * To change this template file, choose Tools | Templates
4  * and open the template in the editor.
5  */
6  package helloworld;
7
8  /**
9   *
10  * @author malex
11  */
12  public class HelloWorld {
13
14      /**
15       * @param args the command line arguments
16       */
17      public static void main(String[] args) {
18          System.out.println("hello world");
19      }
20
21  }
```

Output - HelloWorld (run)

```
run:
hello world
BUILD SUCCESSFUL (total time: 7 seconds)
```



# Programação Orientada a Objeto

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
```

Comentários. Começam com `/*` terminam com `*/`

```
package helloworld;
```

Nome do pacote. Usar minúsculas. Exemplo: `br.com.domínio.helloworld`

```
/**
 *
 * @author malex
 */
```

Comentários para Java doc. Começam com `/**` terminam com `*/`

```
public class HelloWorld {
```

Nome da classe. Começar com maiúsculas. Notação CamelCase: `NomeDaClasse`

```
/**
 * @param args the command line arguments
 */
```

```
public static void main(String[] args) {
    System.out.println("hello world");
}
```

**public:** Qualificador que indica que o método é visível a outras classes  
**static:** Qualificador que indica que o método pertence (é estático) a classe que o criou  
**void:** Indica que não há nenhum valor a ser retornado  
**main:** Método principal. Indica o ponto inicial da execução da classe.  
**String[] args:** Vetor de strings. Responsável por receber valores que serão processados internamente à classe.

**println:** Método da classe `System`.  
Imprime uma linha e posiciona o cursor na linha seguinte.

## Referências

Deitei P. e Deitel H., 2010, Java : Como programar, 8ª Ed., Pearson Pretice Hall

Teruel, E. C., 2015, Programação Orientada a Objetos com Java - sem mistérios - 1ª Ed., Editora Uninove



**Prof. MSc. Marcos Alexandruk**

**E-mail: [alexandruk@uni9.pro.br](mailto:alexandruk@uni9.pro.br)**

**<https://github.com/alexandruk/poo>**

# **Programação Orientada a Objetos com Java**

## **aula 02**

## Objetivo

Apresentar os conceitos de orientação a objetos, preparar o ambiente para programação Java SE (Standard Edition) e iniciar a programação em Java com orientação a objetos.



## Orientação a Objetos

Classes dão origem aos objetos, que só existem no programa em execução.

A classe é a "receita de bolo" do objeto.

Uma classe pode gerar vários objetos com características e comportamentos diferentes.

Exemplo: Cada ser humano possui suas próprias características físicas: cor dos olhos, cor dos cabelos, altura, peso etc. No entanto, podemos dizer que cada ser humano pertence a uma classe chamada Pessoa.





## Orientação a Objetos

Dentro da classe **Pessoa** são definidas as **características** e os **comportamentos**.

Chamamos as características, no mundo orientado a objetos, de **atributos** e os comportamentos de **métodos**.

A classe **Pessoa** pode ter, por exemplo, os seguintes atributos e seus tipos (em java, atributos também podem ser chamados de variáveis):

- nome (texto)
- cor dos olhos (texto)
- cor do cabelo (texto)
- sexo (caractere - uma letra)
- altura (número real)
- peso (número real)



## Orientação a Objetos

Objeto1: **Pessoa1:**

nome = "Alfie"

corDosOlhos = "Preto"

corDoCabelo = "Azul"

Sexo = "M"

altura = 1.80

peso = 100

Objeto2: **Pessoa2:**

nome = "Josefina"

corDosOlhos = "Amarelo"

corDoCabelo = "Magenta"

Sexo = "F"

altura = 1.70

peso = 80

Cada uma dessas pessoas criadas é um objeto da classe **Pessoa**.

No mundo da orientação a objetos, dizemos que **cada objeto é uma instância da classe que o cria.**

## Classe

## Instâncias

### Ambiente de Desenvolvimento

#### Classe: Pessoa

nome: Texto  
corDosOlhos: Texto  
corDoCabelo: Texto  
sexo: Caractere  
altura: Real  
peso: Real

### Programa em Execução

#### Objeto: Pessoa1

nome: Alfie  
corDosOlhos: Preto  
corDoCabelo: Azul  
sexo: M  
altura: 1.80  
peso: 100

#### Objeto: Pessoa1

nome: Josefina  
corDosOlhos: Amarelo  
corDoCabelo: Magenta  
sexo: F  
altura: 1.70  
peso: 80



## Orientação a Objetos

*Essa é a grande "sacada" da orientação a objetos: Uma classe que pode criar vários objetos, como uma “fábrica de objetos”, mas a coisa não para aí: Um objeto pode se relacionar com outro(s) ou ser criado por outro(s).*

*Podemos ter, por exemplo, uma classe chamada Endereço e poderemos associar a pessoa a um endereço. Poderemos fazer, ainda, com que um endereço consiga criar um bairro ainda não cadastrado. Tudo é possível no maravilhoso mundo orientado a objetos. Perceba que a classe Endereço, não precisa se relacionar apenas com uma pessoa, pode relacionar-se com uma empresa e se for preciso dar manutenção no endereço, apenas uma classe é afetada. Uma das grandes vantagens da orientação a objetos é essa: Casa coisa deve ficar em seu devido lugar.*

- Thiago Graziani Traue



## Orientação a Objetos

*Classes são "moldes" que programamos e que geram os objetos em tempo de execução, ou seja, quando o programa está rodando.*

*Esses objetos são chamados de instâncias das classes. O que define as características dos objetos são seus atributos e seus métodos.*

*Programar de forma estruturada é mais fácil. Pois orientação a objetos é um paradigma totalmente novo; uma nova forma de pensar.*

*Contudo, imagine que você está desenvolvendo uma aplicação para uma grande corporação, onde é preciso manipular dados financeiros, de clientes, funcionários, etc. Separar as coisas em seus devidos lugares lhe facilitará muito na hora de realizar alterações ou dar manutenção em seu código.*

- Thiago Graziani Traue



## Classes, atributos e métodos

Este é um dos tópicos mais importantes desta disciplina, pois o entendimento destes conceitos é fundamental para programar corretamente em Java, utilizando orientação a objetos.

Para programarmos em Java, é muito importante seguirmos as boas práticas de programação que definem a estrutura de uma classe, que deve ser composta, nessa ordem, por:

1. Importações de outras classes e pacotes
2. Definição do pacote o qual a classe pertence
3. Definição da classe
4. Atributos locais
5. Métodos

Os **atributos** são as variáveis locais que pertencem a classe e são por ela utilizados. Os atributos podem ser de vários tipos: texto, caractere, número inteiro, número real etc.

Os **métodos** são, os comportamentos das classes, ou seja, são "funções" que podem ser chamadas para executar alguma tarefa.



## Métodos

Um método, nada mais é que a execução de alguma tarefa quando necessário.

Para essa execução, os métodos podem receber informações (dados) externas e podem retornar valores para quem o chamou. Sim, os métodos precisam ser chamados.

Se um método retorna um valor, dizemos que o método é "tipado", ou seja, ele conterá uma informação de algum tipo que pode ser um texto, um número etc.

Se o método não retorna nada para quem o chamou, então ele é "vazio", ou, como dizemos em Java, "**void**".

Veja um exemplo de método "**vazio**" que apenas imprime, em console, o nome do autor:

```
public void imprimeNomeAutor() {  
    System.out.println("Fulano de Tal");  
}
```

Veja um exemplo de um método "**tipado**", que recebe dois valores e retorna a soma deles:

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

Note que no método "**soma**" dois valores são necessários para sua execução. Neste caso, eles são passados via "parâmetros" dos métodos, ou seja, quem invocar este método deverá passar, também, os valores que devem ser somados.



## Classes

Uma classe Java pode ser instanciada por outra, ou seja, quando seu programa estiver sendo executado, um objeto poderá se relacionar a outro, através de suas instruções.

Em Java, para uma classe referenciar outra, utiliza-se a palavra reservada "new". Ao utilizar a instanciação de uma classe, o método construtor da classe é chamado automaticamente.

Por exemplo, imagine a classe "**Pessoa**" criando um endereço. Sua estrutura será assim:

```
public class Pessoa(){  
    Endereco end = new Endereco();  
    //Criou-se, aqui, um atributo "end", do TIPO Endereco.
```

A variável "**end**" é do TIPO **Endereco** e, portanto, assume todas as características e comportamentos que a classe "**Endereco**" pode ter.

Podemos, também, invocar um construtor com passagem de parâmetros, como boas práticas de programação, contudo, neste caso, devemos implementar o construtor na classe.



Let's coding





```
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        String texto;
        texto = "O São Paulo é tricampeão mundial de futebol";
        System.out.println(texto);
        leitor.close();
    }
}
```



## Classe Scanner

```
import java.util.Scanner;

Scanner sc = new Scanner(System.in);

float numF = sc.nextFloat();
int num1 = sc.nextInt();
byte byte1 = sc.nextByte();
long lg1 = sc.nextLong();
boolean b1 = sc.nextBoolean();
double num2 = sc.nextDouble();
String nome = sc.nextLine();
```



```
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        String texto;
        Scanner leitor = new Scanner(System.in);
        System.out.print("Digite o texto: ");
        texto = leitor.nextLine();
        System.out.println("Você digitou: " + texto);
        leitor.close();
    }
}
```



## Calculadora

Vamos implementar uma simples **calculadora de números inteiros**, com **dois atributos locais** (para armazenamento de valores a serem operados) e **quatro métodos com retorno**: Soma, Subtração, Multiplicação e Divisão.

Serão necessárias **duas classes**, pois vamos programar já utilizando o conceito de orientação a objetos com passagem de valores para métodos.

A classe "**Main**" ("Principal") que será responsável por criar a instância da classe responsável pelas operações matemáticas. Nessa classe, faremos a leitura dos valores via teclado.

A classe "**Matematica**" que será responsável por realização das operações aritméticas. Lembre-se do conceito de orientação a objetos: Cada coisa deve ficar em seu respectivo lugar, por isso, quem executa operações aritméticas, é a classe responsável por isso que, neste caso, chama-se "Matematica".

NOTA: Podemos invocar um construtor com passagem de parâmetros. Contudo, como boa prática de programação, neste caso, precisamos implementar o construtor na classe. A classe "Matematica" mostra o construtor implementado manualmente e a passagem de parâmetros para ele pela classe "Principal".

# Programação Orientada a Objeto



```
public class Matematica {  
    //Atributos locais  
    int a, b;  
    //Construtor da classe, que recebe dois valores (x e y) e atribui  
    //aos valores locais (a e b);  
    public Matematica (int x, int y){  
        a = x;  
        b = y;  
    }  
    //Metodo de soma  
    public int soma(){  
        return a + b;  
    }  
    //Metodo de subtracao  
    public int subtrai(){  
        return a - b;  
    }  
    //Metodo de multiplicacao  
    public int multiplica(){  
        return a * b;  
    }  
    //Metodo de divisao  
    public int divide(){  
        return a / b;  
    }  
}
```

# Programação Orientada a Objeto



```
import java.util.Scanner;
public class Main {
    public static void main(String args[]) {
        //Define o leitor do teclado
        Scanner leitor = new Scanner(System.in);
        //Define a variáveis locais
        int x, y;
        //Le:
        System.out.print("Informe o 1º valor: ");
        x = leitor.nextInt();
        System.out.print("Informe o 2º valor: ");
        y = leitor.nextInt();
        //Cria a instancia da classe Matematica utilizando o construtor
        Matematica mat = new Matematica(x, y);
        // Imprime o resultado das operações através
        // de chamadas aos métodos da classe Matemática
        System.out.println("-----"); //Apenas para organizar a saída
        System.out.println("O valor da soma é: " + mat.soma());
        System.out.println("O valor da subtração é: " + mat.subtrai());
        System.out.println("O valor da multiplicação é: " + mat.multiplica());
        System.out.println("O valor da divisão é: " + mat.divide());
        System.out.println("-----"); //Apenas para organizar a saída
    }
}
```



**Prof. MSc. Marcos Alexandruk**

**E-mail: [alexandruk@uni9.pro.br](mailto:alexandruk@uni9.pro.br)**

**<https://github.com/alexandruk/poo>**



# **Tipos de dados em Java**

## **aula 03**



## Objetivo

Apresentar os principais tipos de dados e exemplos de uso.



## O que são tipos de dados

Cada Classe em Java representa algo que virará um ou mais objetos, quando o programa estiver em execução.

O objeto tem duas partes principais: características (atributos) e comportamentos (métodos).

Para se definir os atributos de uma classe usamos as variáveis, que podem ser de vários tipos: texto, número inteiro, número real, verdadeiro ou falso, caractere etc.

Existem dois tipos de dados em Java: primitivo e referência.

## Tipos primitivos



	Tipo primitivo	Espaço em bytes usado na memória	Exemplo
	char	2	char sexo='M';
inteiros	byte	1	byte idade=55;
	short	2	short x=3456;
	int	4	int y = 678934;
	long	8	long cod=1756453;
reais	float	4	float pi=3.1415F;
	double	8	double valor=34.56;
	boolean	1	Boolean casado=true;



## Tipos primitivos

Cada tipo usa uma quantidade diferente de bytes em memória para armazenamento das informações. Hoje em dia, com computadores poderosos, pode não ser tão preocupante usar uma variável do tipo **long** para se armazenar um inteiro curto (idade, por exemplo), mas lembre-se que se você estiver programando para dispositivos móveis ou legados, a quantidade de memória, muitas vezes, pode ser limitada. Isso quer dizer que é muito importante usar o tipo certo para armazenar a informação desejada, independente de onde será executado seu programa.



## Tipos primitivos

**char:** Armazena um único caractere. A atribuição de valor deve ser feita sempre com aspas simples;

**byte:** Armazena valores inteiros, com um byte alocado em memória. Pode armazenar valores de -128 até 127 (o zero conta);

**short:** Armazena valores inteiros, utilizando 2 bytes de memória. Pode armazenar valores entre -32,768 e 32,767;

**int:** Armazena valores inteiros, utilizando 4 bytes. Pode armazenar valores inteiros de -2147483648 até 2147483647.

**long:** Armazena valores inteiros, utilizando 8 bytes. Pode-se armazenar valores de -9223372036854775808 até 9223372036854775807.

**float:** Armazena valores de ponto flutuante, ou seja, valores reais, que necessitam de casas decimais. Utiliza-se o ponto para separação do decimal. Utiliza 4 bytes de memória.

**double:** Armazena valores reais, isto é, com ponto flutuante. Utiliza 8 bytes de memória;

**boolean:** Armazena informações que podem ser apenas de dois tipos: verdadeiro (true) ou falso (false), por isso utiliza apenas 1 byte de espaço em memória.



## Tipos de referência

**String:** Este tipo de dados é um dos mais usados para se armazenar texto. Esse tipo é implementado nativamente pelo Java, utilizando a classe **String.java**, como uma cadeia de caracteres e, internamente, é manipulado dessa forma. Deve-se utilizar aspas duplas para atribuição de valores a este tipo. Exemplo:

```
String nome = new String("Fulado de Tal");
```

**Date:** É utilizado para armazenamento de datas. É implementado pelo Java através da classe **Date.java**. Exemplo:

```
Date dtNascimento = new Date("10/10/2020").
```

### IMPORTANTE:

Note que todos os tipos primitivos são declarados com todas as letras minúsculas e os tipos de referência começam com uma letra maiúscula. Porque? Porque String e Date são classes Java.



## Convertendo tipos (casting)

Em Java é muito comum a necessidade de converter um tipo em outro, e isso vale tanto para tipos primitivos quanto tipos de referência.

Isso, em programação, chama-se "casting".

Para fazer isso, será exemplificada a conversão de um valor float para int:

```
float X = 10;  
int Y = (int) X;
```

Contudos, se o valor for um número real, a conversão irá pegar somente a parte inteira do valor, ou seja:

```
float X = 20.5F;  
int Y = (int)X; //neste caso o Y valerá 20
```

No exemplo acima, foi preciso usar a letra F junto ao número real, para "dizer" ao Java que isso é um número real.





## Principais tipos de operações

Como qualquer linguagem de programação, em Java é possível realizar uma série de operações com os valores armazenados nas variáveis.

Os principais tipos de operações que podem ser realizadas sobre os dados são por meio de expressões:

- Aritméticas
- Relacionais
- Lógicas



## Expressões aritméticas

Utilizadas para cálculos matemáticos convencionais (soma, subtração, divisão e multiplicação):

### Soma:

```
int a = 10;  
int b = 20;  
int c = a + b;
```

### Subtração:

```
double salario = 2456.76;  
double desconto = 200.57;  
double salLiquido = salario - desconto;
```

### Multiplicação:

```
double quantidade = 20;  
double valor = 109.6;  
double total = quantidade * valor;
```

### Divisão:

```
double salario = 1950.75;  
double percentualAumento = 10.0;  
double aumento = salario * percentual / 100;
```



## Forma reduzida de escrever uma operação

Em Java há uma forma simples de escrever operações, facilitando muito na hora de digitar. Podemos dizer que é uma forma reduzida de se escrever uma operação.

Por exemplo, se for preciso incrementar 1 em uma variável chamada `a`, normalmente escrevemos assim:

```
//...  
a = a + 1;  
//...
```

A forma reduzida dessa mesma expressão é:

```
//...  
a++;  
//...
```

E se precisarmos incrementar essa mesma variável de 2, podemos escrever da seguinte forma:

```
//...  
a += 2;  
//...
```



## Expressões relacionais

As expressões relacionas envolvem a comparação de dois valores. Em Java podemos comparar duas variáveis e obter verdadeiro (true) ou falso (false) como resultado dessa comparação.

Os operadores relacionais são:

- Igual: ==
- Menor: <
- Menor ou igual: <=
- Maior: >
- Maior ou igual: >=
- Diferente: !=

### Exemplo:

```
//...
int a, b, c;
a = 2;
b = 5;
c = 5;
//...

a == b // false
b == c // true
b < a  // false
b <= c // true
a > b  // false
b >= c // true
```



## Expressões lógicas

Expressões lógicas são aquelas que envolvem a comparação de dois valores boolean e resultam em um terceiro valor boolean.

Em Java podemos utilizar o "Não", o "ou" e o "e", conforme ilustrado pelo cenário abaixo.

- Não (not) : !
- E (and): &&
- Ou (or): ||

### Exemplo:

```
//...
boolean a, b;
a = true;
b = false;
//...

!a // false
!b // true
a && b // false
a && a // true
a || b // true
b || b // false
```



## Exemplo: Calculo do IMC (Índice de Massa Corporal)

```
import java.util.Scanner;
public class CalculadoraSimples {
    public static void main(String[] args) {
        //Declaracao do scanner:
        Scanner leitor = new Scanner(System.in);
        //Declaracao das variaveis que serao utilizadas:
        float peso, altura, imc;
        //Informacao de instrucoes para o usuário:
        System.out.print("Informe o PESO: ");
        //le e armazena o valor do peso:
        peso = leitor.nextFloat(); //Le um valor de ponto flutuante
        //le e armazena o valor da altura:
        System.out.print("Informe a ALTURA: ");
        altura = leitor.nextFloat(); //Le um valor de ponto flutuante
        //calcula:
        imc = peso / (altura * altura);
        //Imprime o resultado
        System.out.println("\n\tO IMC desta pessoa é " + imc + "\n\n");
    }
}
```



**Prof. MSc. Marcos Alexandruk**

**E-mail: [alexandruk@uni9.pro.br](mailto:alexandruk@uni9.pro.br)**

**<https://github.com/alexandruk/poo>**

## Estruturas de Controle de Fluxo e Laços em Java

### aula 04





## Objetivo

Usar os diferentes tipos de controles de fluxo e laços que podem ser descritos em Java.

## Controle de Fluxo



## Controle de fluxo

Em Java (e outras linguagens de programação) usamos estruturas de controle de fluxo (ou estruturas de seleção) para testes lógicos.

Nas estruturas de seleção um determinado trecho de código é executado caso o teste lógico seja verdadeiro e outro caso o resultado seja falso.

Estamos falando do importante "if...else" (se, senão, em inglês).

Porém, conforme veremos, há mais de uma forma de trabalharmos com estruturas de seleção em Java.



## if...else

O **if...else** é a forma mais simples de fazermos o controle de fluxo de execução do programa, pois apenas desviamos a execução para determinada linha quando obtemos verdadeiro no teste ou para linha do else quando obtemos falso.

A estrutura do **if...else**, em Java, é:

```
//...
if (<condição lógica>) {
    // trecho de código que será executado se o resultado for verdadeiro
}
else {
    //trecho de código que será executado se o resultado for falso
}
//...
```



## if...else if...else

Podemos, também, usar um ou mais **else if** caso outras condições também não sejam atendidas, da seguinte forma:

```
//...
if (expressao 1) {
    //procedimento caso o resultado da expressao seja true
} else if (expressao 2) {
    //procedimento caso o resultado da primeira expressao tenha sido false
    //e da expressao atual tenha sido true.
} else if (expressao 3){
    //procedimento caso o resultado da primeira e segunda expressoes tenham
    //sido false e da expressao atual tenha sido true.
} else {
    //procedimento caso o resultado de todas as expressoes anteriores tenha
    //sido false
}
//...
```



## Exemplo:

Para exemplificar, vamos programar um controle de fluxo que calcula a média a partir de três notas, contudo pega apenas as duas maiores para o cálculo:

[arquivo: aula04\_01.txt]

Note que neste exemplo, foi criado um **método estático** para o cálculo da média.

Métodos estáticos não precisam ter a instância da classe em memória para serem invocados (chamados), ou seja, podem ser chamados diretamente pelo seu nome.



## Comparação de Strings

Para comparar valores do tipo String, os operadores aritméticos convencionais não funcionam, pois trabalham apenas com valores numéricos e, como vimos anteriormente, Strings são cadeias de caracteres.

Para comparar valores String utilizamos o método **equals** ou **equalsIgnoreCase** (quando não queremos que a caixa do texto, isto é, maiúsculas e minúsculas, seja levada em consideração).

Veja um exemplo a seguir:

[arquivo: aula04\_02.txt]



## switch...case...default

Temos, em Java, uma outra forma para fazer controle de fluxo: switch...case...default

Neste caso, usamos uma única variável para realizar comparações encadeadas.

```
//...
switch(variavel) {
    case valor:
        //operacao a ser realizada se a variavel contiver o valor especificado no case
        break;
    case valor2:
        // operacao a ser realizada se a variavel contiver o valor especificado no case 2
        break;
    case valor3:
        //operacao a ser realizada se a variavel contiver o valor especificado no case 3
        break;
    default:
        //operação a ser realizada se a variavel não contiver o valor especificado e
        //nenhum dos cases acima
}
//...
```





## switch...case...default

Vamos ver um exemplo de uso, bem simples, onde o usuário informa um valor e o programa verifica se o valor informado está sendo tratado (é previsto no código) ou não:

[arquivo: aula04\_03.txt]



## Exercício

Criar um programa que verifica dois números inteiros e informa:

**Os dois números são pares**

**Apenas um dos números é par**

**Nenhum dos números é par**

[arquivo: aula04\_04.txt]

## Laços (estruturas de repetição)



## for

Os laços do tipo for são equivalentes "para" em algoritmos, ou seja, "para i de um valor até outro, repita".

Os laços for permitem apenas comparações com valores numéricos para parada. Em Java, a sintaxe de laços for é:

```
//...  
for(<condição de início>;<condição de parada>;<condição de incremento/decremento>  
//... Trecho de código que será executado N vezes  
{  
//...
```



## for

Para exemplificar, vamos fazer um programa que imprime todos os números pares entre 0 e 10. Neste caso, o laço ficará assim:

```
//...  
for (int i = 0; i <= 10; i++){  
    //É o mesmo que: Para i de 0 a até 10 (inclusive)  
    if (i % 2 == 0) // Verifica se se o resto de sua divisão por 2 é 0  
        System.out.println(i);  
}  
//...
```

Note que no laço exemplificado acima, a variável de controle (i) foi declarada no próprio laço e foi incrementada ao final do laço.

**Resumindo:** O laço **for** funciona assim: A variável de controle a é inicializada com o valor inicial declarado na condição de início, o laço é executado. Depois disso a condição de parada é verificada. Se ela for falsa, então a variável de controle sofre o incremento ou decremento.



## for

### Exercício 04.01:

Elaborar um programa que imprima o resultado da tabuada do 1 ao 10, conforme abaixo.

Dica: Use for encadeado.

1x1=1

1x2=2

...

10x9=90

10x10=100



**for**

## **Exercício 04.02:**

Elabore um programa que:

- 1º Solicite a quantidade de alunos na turma
- 2º Solicite a inserção da idade de cada aluno da turma
- 3º Calcule e apresente a soma e a idade média dos alunos da turma



## while

O laço do tipo while fornece ao programador uma liberdade maior de condições de incremento ou decremento pois ela pode ocorrer em qualquer momento dentro do laço. Outra diferença é que laços do tipo while permitem a comparação de valores não numéricos como condição de parada.

Diferentemente dos laços for, os laços while precisam que a variável de controle seja inicializada antes. Sintaxe dos laços while:

```
//...  
while (<condição de parada>){  
    //Trecho de código que será executado N vezes  
    <incremento ou decremento da variável de controle>  
}  
//...
```





## while

Para exemplificar, vamos criar um pequeno programa que imprime os 10 primeiros valores ímpares. O código ficará assim:

```
//...  
int i = 0;  
while(i <= 10){  
    if (i % 2 != 0) //Verifica se o resto da divisão por 2 é diferente de zero.  
        System.out.println(i); //Imprime a variável de controle  
    i++; //Incrementa a variável de controle  
}  
//...
```



## do...while

A grande diferença entre este tipo de laço está no momento em que ele verifica a condição de parada. Em laços do tipo do...while a condição de parada é verificada ao final de cada execução do laço, ou seja, o laço é executado e depois é verificada a condição. Isso quer dizer que o laço do...while sempre será executado ao menos uma vez.

Sintaxe dos laços do...while:

```
do{  
    //...Trecho de código que será executado N vezes  
    <incremento ou decremento da variável de controle>  
} while(<condição de parada>)
```



## do...while

Assim como o laço while, este também precisa que a variável de controle seja inicializada antes. Vamos ver um exemplo de um pequeno trecho de código que é capaz de imprimir os 10 primeiros números em ordem inversa (do maior para o menor):

```
//...
int i = 10;
do {
    System.out.println(i); //Imprime a própria variável de controle
    i--; //Decrementa a variável de controle
}while (i != 0); //Condição de parada
//...
```



## Quando usar cada tipo de laço?

"Não se preocupe, muitos programadores ficam confusos na hora de escolher qual laço usar, ou seja, essa é uma pergunta muito comum!

E, na verdade, não existe uma resposta exata para essa pergunta, pois cada situação é única. Há situações em que qualquer um dos três laços podem ser usados e dependerá, apenas, do "gosto" do programador. Contudo, haverá situações em que um determinado tipo de laço será necessário.

Isso quer dizer, então, que a resposta para essa pergunta é: Depende da situação.

Analise a situação cuidadosamente. Veja em que momento a variável de controle deve ser inicializada e a que momento ele deve ser incrementada."

- THIAGO GRAZIANI TRAUE



**Prof. MSc. Marcos Alexandruk**

**E-mail: [alexandruk@uni9.pro.br](mailto:alexandruk@uni9.pro.br)**

**<https://github.com/alexandruk/poo>**

## Arrays e Coleções de Dados

### aula 05



## Objetivo

Implementar as principais formas estruturas de dados em Java, com aplicação das boas práticas de programação.



## Arrays

Um array, em Java, é uma estrutura de dados que armazena uma série de objetos em sequência, todos do mesmo tipo, em posições distintas da memória.

Array em Java é o mesmo que "vetor" em estruturas de dados. Um array é a implementação manual dessa estrutura de dados.

Em Java, para declarar um array utiliza-se um par de colchetes: [ ].

Veja um exemplo, abaixo de declaração de um array de inteiros com 10 posições, ou seja, uma estrutura de dados que armazena até 10 valores inteiros:

```
//...  
int x[] = new int[10];  
//...
```

É muito importante ressaltar que em cada posição exemplo acima pode-se armazenar um número inteiro e **a numeração das posições inicia-se em 0 (zero!)**.

Isso quer dizer que se um array possui tamanho 10, as posições de endereçamento possível são de 0 a 9.





## Arrays

Exemplo de array de três posições do tipo String e a implementação do código que atribui valores a cada posição do array:

```
//...
//declaração do array de strings de tamanho 3
String nomes[] = new String[3];
//atribui valor a cada posição do array, de 0 a 2
nomes[0] = "Fulano";
nomes[1] = "Beltrano";
nomes[2] = "Cicrano";
//para acessar cada posição, podemos usar um laço, de 0 a 2
for (int i = 0; i <= 2; i++) {
    System.out.println("Na posição " + i + " do array, temos: " + nomes[i]);
}
//...
```

[aula05\_01.txt]



## Arrays

Graficamente, podemos dizer que o exemplo apresentado cria uma estrutura de dados como vista a seguir:

Posição#	0	1	2
Valor	Fulano	Beltrano	Sicrano

É muito importante tomar muito cuidado na hora de percorrer ou acessar os valores de um array, para não acessar uma posição não existente, ou seja, se um array tem tamanho 6, não podemos acessar a posição 7, pois ela não existe. Isso daria um erro no programa que, se não tratado pode finalizar o programa inesperadamente e fazer com que o usuário perca todo seu trabalho.



## Arrays

Para praticar, implemente um programa que leia um array de 10 inteiros e imprima para o usuário o maior valor inserido dentro do array.

Dica: Use uma variável auxiliar para armazenar o primeiro valor e, se encontrar algum valor maior enquanto percorre o array, o valor dessa variável é substituído pelo valor encontrado. A cada laço será preciso comparar o valor na posição em que está com o valor na variável auxiliar.

[aula05\_02.txt]



## Vetores multidimensionais

Podemos ter quantas dimensões forem necessárias em nosso array, ou seja, ter mais de uma possível posição de endereçamento.

O mais comum é a implementação de arrays de duas posições (bidimensionais), resultando em algo parecido com uma tabela de duas colunas.

Para exemplificar, vamos implementar um array bidimensional que armazena na primeira posição um nome e na segunda posição o sobrenome.

**Nota:** Neste exemplo, utilizaremos um array bidimensional cujo número de linhas e colunas será o mesmo. Podemos, em Java, criar um array com quantas linhas e colunas forem necessárias, de tamanhos diferentes.

[aula05\_03.txt]



## Array tipado com uma classe criada por você

Além de criar arrays de tipos primitivos, como exemplificado até agora, podemos criar um array cujo tipo é implementado por você.

Imagine, por exemplo ter uma lista de pessoas em uma só estrutura de dados, ou seja, uma lista do tipo "Pessoa" e cada pessoa dessa lista ter suas próprias características. Essa é uma das grandes vantagens do mundo orientado a objetos, pois isso é possível em Java.

Para este exemplo, precisamos antes, implementar a classe que usaremos para "tipar" nosso array, neste caso, a classe "Pessoa". Nossa pessoa terá os seguintes atributos:

**Nome** (String)

**Idade** (int)

**E-mail** (String)



## Array tipado com uma classe criada por você

A codificação dessa classe, poderá ser assim:

```
public class Pessoa {  
    String nome;  
    int idade;  
    String email;  
}
```

A seguir, vamos criar uma classe que poderá conter até 3 pessoas com as características mostradas acima e imprimir na console as informações das pessoas:

[aula05\_04.txt]



## Como descobrir o tamanho de um array?

Há uma função que retorna o tamanho total de um array:

```
nome_do_array.length
```

A função `length` retorna o tamanho total de posições endereçáveis. Portanto, se estiver iterando este array, não esqueça de ir de 0 até `nome_do_array.length - 1`.

```
public class Main {  
    public static void main(String[] args) {  
        String nomes[] = new String[3];  
        System.out.println("Tamanho do vetor nomes: " + nomes.length);  
    }  
}
```



## Implementações prontas no Java

Um array é uma implementação manual da estrutura de vetores. Contudo, em Java, podemos utilizar implementações prontas de estruturas de dados de coleções. É o caso das **listas**.

Embora você gaste mais memória utilizando estruturas prontas e não tenha tanto controle das estruturas prontas quanto tem com aquelas implementadas manualmente, existem vantagens de se trabalhar com estruturas de dados prontas em Java, como:

- Capacidade de alocação dinâmica de dados (a lista, por exemplo, pode ter o tamanho que você quiser e este tamanho pode mudar em tempo de execução;
- Métodos prontos de ordenação, retirada e acréscimo de valores em posições da lista de forma rápida;
- Métodos prontos de consulta às listas;

Será apresentada a seguir, a principal implementação de coleção em Java: a **ArrayList**.





## ArrayList

ArrayList é uma coleção do Java implementada pela classe **java.util.ArrayList**, ou seja, sempre que for utiliza-la, é preciso adicionar no início de seu código as instrução de importação dessa classe, assim:

```
import java.util.ArrayList;
```

Usar um ArrayList não é difícil. É muito importante saber que, para usá-lo, é preciso indicar o tipo de objeto que está sendo armazenado nessa estrutura de dados. A sintaxe de sua declaração é assim:

```
ArrayList<ObjetoUsado> nomeDaColecao = new ArrayList<ObjetoUsado>();
```

Por exemplo, se você está criando uma coleção nomes de clientes, ou seja, de valores do tipo String, sua ArrayList ficará assim:

```
ArrayList<String> clientes = new ArrayList<String>();
```



## ArrayList

Quando você cria um objeto que é do tipo ArrayList, você possui a disposição os métodos que a classe implementa. Os principais métodos são:

**add(VALOR):** Serve para acrescentar um item na estrutura. Você deve passar como parâmetro o valor que está sendo inserido, que deve ser do mesmo tipo da lista.

**add (ÍNDICE, VALOR):** Você também pode dizer em qual posição (índice) quer acrescentar o valor, numericamente. A contagem nos ArrayLists também começa no índice 0.

Exemplo: `clientes.add(1, "Fulano") ;`

**size():** Retorna o tamanho da lista.

Exemplo: `int x = clientes.size() ;`

**get(POSICÃO):** Retorna a objeto que está na posição (índice) informada.

Exemplo: `String cliente1 = clientes.get(1) ;`

**remove(VALOR):** Remove o valor informado. O objeto passado deve ser do mesmo tipo da ArrayList.

Exemplo: `clientes.remove("Beltrano") ;`

**clear():** Limpa, completamente, a lista.

Exemplo: `clientes.clear() ;`



## ArrayList

Este exemplo (arquivo aula\_05\_05.txt) apresenta a implementação de uma ArrayList tipada com a classe "Pessoa" (usa a mesma classe "Pessoa" já criada).

O exemplo faz o seguinte:

- Cria uma lista contendo 3 pessoas;
- Imprime os dados de todas elas (uma a uma) usando um laço;
- Procura uma pessoa chamada "Beltrano" para remoção da lista;
- Imprime a lista de nomes que permaneceram na estrutura.

[aula05\_05.txt]



## ArrayList

Note que, neste exemplo, após a remoção do item, o índice que era ocupado pelo objeto que foi removido não existe mais, ou seja, a própria lista cuidou da remoção e realocação dos objetos que permaneceram na lista, automaticamente.

Se estivéssemos trabalhando com vetores, por exemplo, essa remoção seria bem mais complicada, pois seria necessário mover todos os demais itens manualmente, após a remoção.





**Prof. MSc. Marcos Alexandruk**

**E-mail: [alexandruk@uni9.pro.br](mailto:alexandruk@uni9.pro.br)**

**<https://github.com/alexandruk/poo>**

## Gerando e Interceptando Exceções

### aula 06



## Objetivo

Tratamento de erros ou exceções durante a execução do programa, evitando que o programa encerre inesperadamente.