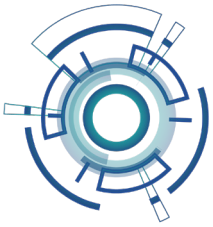


CAPACITAÇÕES

JAVA PARA A PDPJ-Br

JAVA BÁSICO

E-BOOK 1

Ronaldo Pinheiro Gonçalves Junior

Unifor | JUSTIÇA DO TRABALHO Conselho Superior da Justiça do Trabalho | Tribunal Superior Eleitoral | STJ SUPERIOR TRIBUNAL DE JUSTIÇA | JUSTIÇA 40 | PNUD | JUSTIÇA FEDERAL Conselho da Justiça Federal | CNJ CONSELHO NACIONAL DE JUSTIÇA

1 CONCEITOS DE PROGRAMAÇÃO, FUNDAMENTOS DA LINGUAGEM DE PROGRAMAÇÃO JAVA E CONFIGURAÇÃO DO AMBIENTE DE DESENVOLVIMENTO

1.1 HISTÓRICO DA LINGUAGEM DE PROGRAMAÇÃO JAVA

A linguagem de programação Java foi responsável pela popularização do paradigma orientado a objetos. Criada na década de 90 pela Sun Microsystems, a linguagem Java possuía uma vantagem que suas predecessoras não tinham: multiplataforma. O uso de uma máquina virtual fazia com que o código fonte compilado pudesse ser usado em diversas plataformas, uma grande vantagem nessa época.

Em 2004, a Sun Microsystems lançou para a linguagem Java, em sua versão 5, importantes conceitos que estudaremos em nosso curso, como *generics* e enumerações. Em 2010, a Oracle adquiriu a linguagem e continuou a desenvolver novas versões, que introduziram novidades à linguagem, como expressões Lambda, na versão Java 8, em 2014. Estudaremos esses conceitos e essas expressões em nosso material.

Atualmente, Java continua sendo considerada uma das linguagens mais populares do mercado global, sendo utilizada para construção de soluções para diversas plataformas.

1.2 ESTRUTURA DE UMA APLICAÇÃO JAVA

Um primeiro possível passo, na criação de uma aplicação Java, é a criação de um pacote. Em geral, utilizamos o pacote “src” (fonte, do inglês *source*), para guardar nosso código fonte. Podemos criar classes Java em um pacote, para iniciar a construção da nossa aplicação Java. Uma classe Java possui atributos e métodos, que podem estar conectados entre si. Ao avançarmos na estruturação da aplicação, mais pacotes podem ser criados para organizar nossos arquivos Java. Classes relacionadas estarão agrupadas em um mesmo pacote.

Conforme estudado anteriormente, vimos que a linguagem Java segue o paradigma orientado a objetos. Logo, uma aplicação Java faz uso de conceitos de programação orientada a objetos em sua estrutura, como herança, polimorfismo etc. Vale ressaltar a importância da Máquina Virtual Java (JVM), na estrutura de uma aplicação Java, que permite que um código seja escrito uma vez e executado em diferentes plataformas. Finalmente, uma aplicação Java também conta com um sistema automático de gerenciamento de memória, concluindo assim os principais componentes da sua estrutura. Devido ao fato de que estes componentes estão presentes na grande maioria das soluções Java, podemos dizer que sua estrutura é bem definida e iremos entender agora o que é necessário para iniciar o desenvolvimento, utilizando essa linguagem.

1.3 AMBIENTES INTEGRADOS DE DESENVOLVIMENTO (IDE – INTEGRATED DEVELOPMENT ENVIRONMENTS)

A produção de código fonte não está limitada apenas à escrita. Inclui, também, muitas outras tarefas que veremos em breve, como: teste, depuração, projeto e assim por diante. O local de trabalho de um programador conta com ferramentas que auxiliam na execução dessas tarefas, em um ambiente integrado de desenvolvimento, ou IDE (do inglês, *Integrated Development Environment*). A seguir, podemos encontrar uma boa definição para este ambiente (OLIVEIRA, 2019, p. 26):

> IDE, do inglês *integrated development environment*, ou ambiente de desenvolvimento integrado, é um *software* voltado para o desenvolvimento de programas ou aplicativos. O desenvolvedor encontra nos IDEs todas as funções necessárias para o desenvolvimento, desde programas de computador a aplicativos mobile, assim como recursos que possibilitam a criação de códigos com uma incidência menor de erros.

Como em um *software* de edição de documentos de texto, um IDE possui um espaço para criação de código fonte e opções para facilitar a tarefa de construção de programas. Todavia, um IDE não está limitado apenas a um editor de texto. Em breve, utilizaremos dois ambientes integrados de desenvolvimento populares: o Eclipse e o Visual Studio Code. Antes de aprendermos a instalar e configurar cada um desses IDE, vamos aprender alguns componentes que comumente encontramos nesses ambientes e depois estudar cada IDE individualmente.

- **Assistente de marcação de texto**

Ao escrever linhas de código, este componente de um IDE auxilia um programador, por meio de destaques e marcações textuais – por exemplo, destaca palavras que foram escritas de forma errada, indica início e fim de estruturas de código, sinaliza quais linhas estão selecionadas, entre outras marcações visuais diretamente aplicadas no texto (ou código).

- **Preenchedor automático de código**

Estruturas de código de uma linguagem apresentam características que podem ser automaticamente completadas. Um exemplo disso seria, para a linguagem Java, digitar um "(" implica no uso de um ")", pois não podemos ter um parêntese abrindo, sem que haja um parêntese fechando, em seguida. Ao digitar o primeiro símbolo, o IDE irá entender e adicionar o segundo automaticamente. Muitos outros trechos de código podem e são gerados dessa forma.

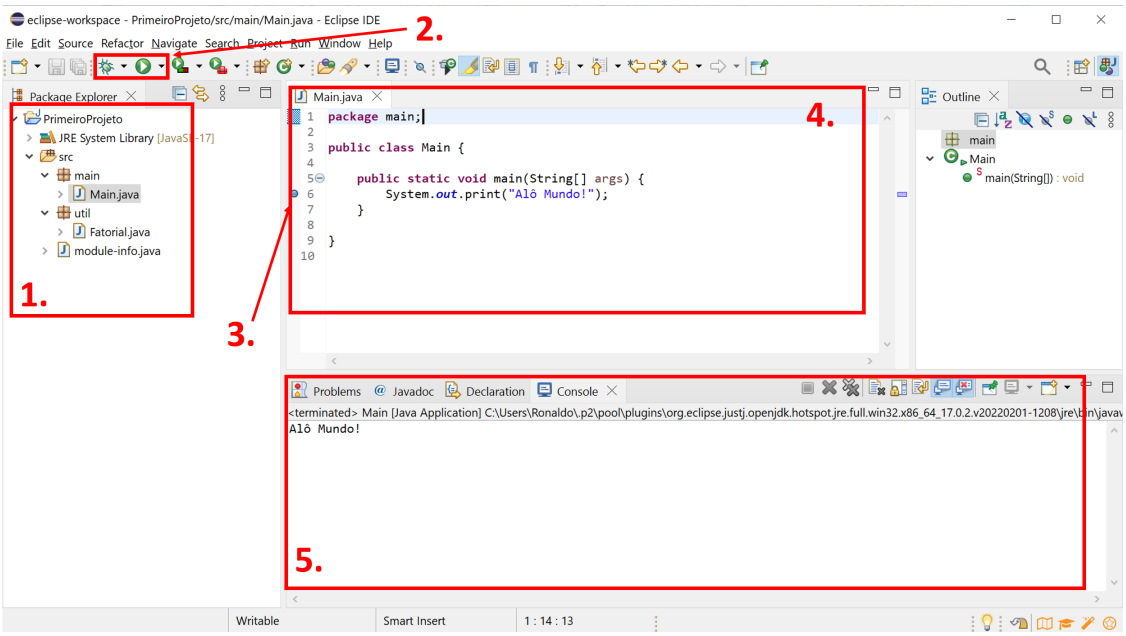
- **Refatorador de código**

Com este componente, podemos realizar tarefas de modificação de código complexas, de maneira mais fácil. Por exemplo, considere um programa que utiliza um rótulo em vários locais de seu código fonte. Caso você precise alterar esse rótulo, em vez de visitar todos os

vários locais onde ele está sendo usado e ajustar manualmente cada um, é possível alterar apenas um, e todos os outros serão ajustados automaticamente. Um IDE tende a contar com múltiplas técnicas de refatoração.
Muitos outros recursos estão presentes em um IDE e a maioria das opções tem pontos em comum. Dois principais critérios utilizados para a escolha de um IDE são: plataformas de desenvolvimento e linguagens a serem utilizadas. Digamos que você deseja trabalhar em uma plataforma mobile. Neste caso, uma boa opção de ambiente de desenvolvimento seria o Android Studio. Todavia, como iremos abordar o desenvolvimento com a linguagem Java, optaremos por iniciar nossos estudos no Eclipse IDE.

1.3.1 ECLIPSE IDE

Criado em 2001 pela IBM, o Eclipse é um ambiente integrado de desenvolvimento bem estabelecido. O IDE ficou conhecido pelo grande suporte à linguagem Java e é amplamente utilizado até hoje. Para conhecer os principais utilitários do Eclipse, utilizaremos um projeto denominado “PrimeiroProjeto”. Note que, no caderno de atividades, você encontrará o passo a passo para instalação, configuração e criação de um projeto no Eclipse.



Elaborada pelo autor, 2023.

1.3.1.1 EXPLORADOR DE PACOTES

O explorador de pacotes – ou, em inglês, *Package Explorer* – está representado pela área "1", na figura. É aqui que fica a estrutura da aplicação Java. Como vimos anteriormente, o pacote src contém o código fonte. Neste exemplo em específico, temos dois pacotes dentro de src: *main*, que contém as classes principais de execução do nosso programa; e *util*, que contém as classes de utilidade. É nesta área que criamos, alteramos e removemos pacotes, classes Java e assim por diante.

1.3.1.2 DEBUG E EXECUÇÃO

O símbolo de "play", na área "2", realiza a execução de um programa. Isto é, nosso código compilado é executado, resolvendo um problema ou realizando uma computação. O símbolo de inseto, logo ao lado, representa uma execução no modo de depuração ou "Debug", o modo de remoção de *bugs*. Um *bug* é um erro de programação, em geral criado de forma não intencional. Ao executar um programa nesse modo, temos a intenção de investigar, identificar e remover erros do nosso programa.

1.3.1.3 BREAKPOINT

Por vezes, para estudar um código em mais detalhes, precisamos executá-lo e interrompê-lo brevemente, para analisar o estado do programa. Fazemos o uso de *breakpoints*, ou pontos de parada, para indicar em quais linhas a execução do programa deve ser interrompida. Em breve, iremos aprender como utilizar *breakpoints* para realizar uma execução em modo de depuração.

1.3.1.4 EDITOR DE TEXTO

A área "4" é reservada para a escrita de código. Note que, na figura, temos um exemplo de código, com algumas palavras destacadas em roxo e outras em azul. A maior parte das opções de construção de código está presente nessa área e, subsequentemente, é a área mais utilizada durante o desenvolvimento de código fonte. Em breve, iremos estudar as estruturas da linguagem Java e entender o que está escrito nesta figura.

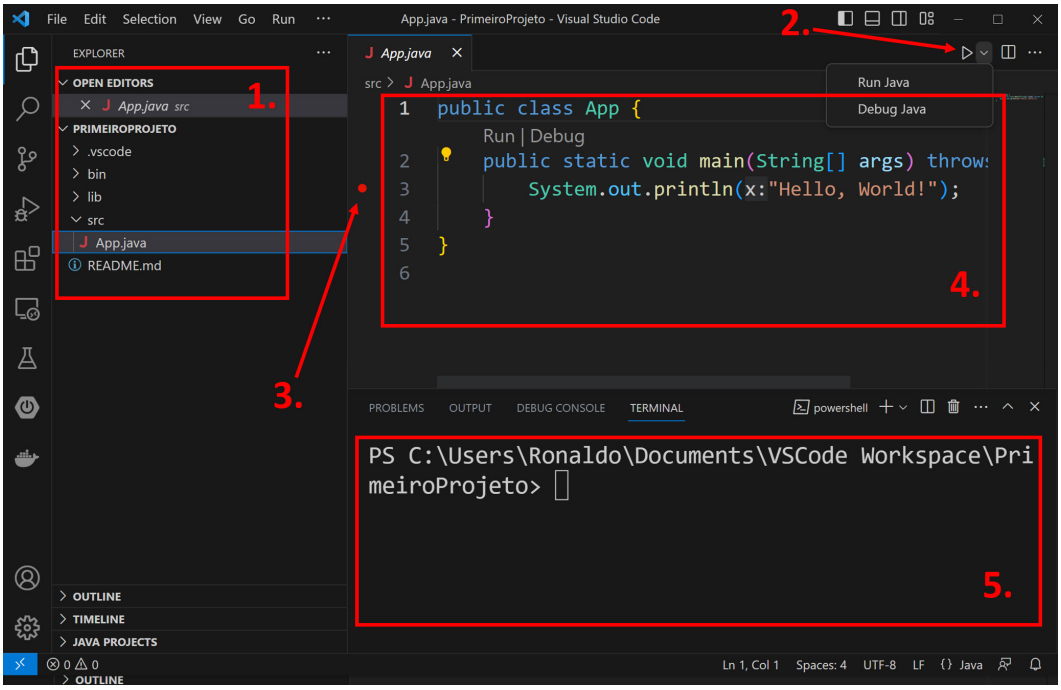
1.3.1.5 CONSOLE

Esta última área destacada representa o espaço de entrada e saída do nosso programa. Em outras palavras, podemos fazer uso desse utilitário para enviar informações para nosso programa em execução e receber outras informações como resposta. No exemplo apresentado, veja que o programa imprime "Alô Mundo!".

Além de ser usado para entrada e saída, podemos dizer que o console foi projetado para que programadores pudessem visualizar vários detalhes de uma execução, como mensagens de tarefas bem-sucedidas, erros de execução e assim por diante.

1.3.2 VISUAL STUDIO CODE

O Visual Studio Code, também conhecido como VSCode, foi criado em 2015 pela Microsoft e é disponibilizado como um editor de texto especializado em código fonte. Entretanto, muitos preferem considerar o Visual Studio Code como um IDE, pois possui a maioria dos utilitários encontrados nestes ambientes de desenvolvimento.
Este ambiente integrado se tornou muito popular entre desenvolvedores, por ser leve e extensível. Diferente do Eclipse, que trabalha com *plugins*, o VSCode trabalha com extensões e possui suporte a muitas linguagens, incluindo o Java.
No caderno de atividades, você encontrará o passo a passo para instalação, configuração e criação de um projeto no Eclipse. Por ora, veja como é o ambiente de desenvolvimento para um projeto semelhante.



Elaborada pelo autor, 2023.

Note que as áreas destacadas para o Eclipse IDE também estão presentes no Visual Studio Code: 1. Explorador de pacotes; 2. Debug e execução; 3. Breakpoints; 4. Editor de texto; e 5. Console. Vamos estudar as diferenças entre estes IDE, em nossas práticas de desenvolvimento, mas vale ressaltar que ambos possuem utilitários semelhantes, para auxiliar programadores em seu cotidiano. Agora que sabemos sobre as ferramentas que nos auxiliam na produção de

código, vamos entender os últimos passos necessários para, então, iniciar nosso aprendizado de estruturas da linguagem Java.

Como vimos anteriormente, a Máquina Virtual Java (JVM) é um componente essencial da

1.4 MÁQUINA VIRTUAL DO JAVA (JVM – JAVA VIRTUAL MACHINE)

estrutura de uma aplicação Java. Através da JVM, um código fonte pode ser escrito, apenas uma vez, e então executado em mais de uma plataforma. Essa característica é chamada "*write once, run everywhere*" (em português, "escreva uma vez, execute em qualquer lugar"). Isso garante portabilidade e segurança.

A JVM também é conhecida por empregar uma técnica eficiente de compilação, chamada *Just-in-Time* (JIT). De maneira tradicional, compilar código fonte na linguagem Java produz código em *bytecode* Java, um formato intermediário, mais próximo do formato da máquina. Ao usar um compilador JIT, o código *bytecode* é traduzido diretamente para código de máquina, resultando em uma melhora em desempenho.

Outra grande vantagem que a JVM traz com seu uso é um subcomponente chamado *Garbage Collector* (em português, "coletor de lixo"). Este componente é responsável por detectar objetos que não estão sendo utilizados e limpar a memória da máquina de forma automatizada. Isso é importante, pois, em algumas linguagens, o programador deve se preocupar em gerenciar a memória. Ou seja, com a JVM, fica mais fácil programar sem cometer erros, como se esquecer de limpar um espaço de memória. Vamos, então, entender como instalar o Java, para utilizar esses componentes.

Um dado primitivo é um valor simples que podemos encontrar no mundo real, como a idade

2 TIPOS DE DADOS PRIMITIVOS

de uma pessoa. Um tipo é uma definição, quanto ao domínio do dado. Por exemplo, uma idade pode ser do tipo inteiro. Ou seja, é um número pertencente ao conjunto de números inteiros positivos.

Exemplos concretos desse tipo primitivo: "10" anos, "18" anos, "68" anos etc. Não é comum, por exemplo, uma idade de "20,2" anos ou "-15,5" anos ou "Z" anos.

Na linguagem Java, temos uma tipagem estática. Isso significa que é necessário definir previamente um tipo, para representar um dado.

Uma vez definido, este dado só poderá ser desse tipo. Além disso, dizemos que a linguagem

Cada tipo primitivo tem uma palavra reservada na linguagem Java, conforme a lista a seguir:

"byte": ocupa pouco espaço e é utilizado para representar dados com valores inteiros de -128 a 127.

"short": também representa valores inteiros, mas dentro de um limite maior, que compreende valores de -32.768 a 32.767.

"int": tipo mais comum de representação para valores inteiros, com um limite entre -2.147.483.648 e 2.147.483.647.
"long": tipo inteiro, utilizado para números maiores que o "int" comporta, indo de -9.223.372.036.854.775.808 a 9.223.372.036.854.775.807. É comum usar um "L" maiúsculo ou minúsculo após o valor, para deixar explícito o tipo long.
"float": representa números com ponto flutuante, fracionários. É necessário seguir com "F" maiúsculo ou minúsculo.
"double": tipo mais comum de representação para valores de ponto flutuante. Semelhante ao float, mas com dupla precisão.
"char": representa caracteres únicos.
"boolean": o tipo booleano pode representar "verdadeiro" ou "falso", apenas.

Java tem uma tipagem forte, pois não existe conversão dinâmica em tempo de execução. Em outras palavras, uma vez definido o tipo de um dado, este sempre será do tipo definido. Essa característica diminui a flexibilidade de programação, mas ajuda a evitar erros de programação e é mais fácil de ser compreendida, ao investigarmos o código.
Ao escrever um tipo de um dado em uma linha do programa Java, podemos seguir definindo

2.1 VARIÁVEIS

um rótulo para identificar o dado. Por exemplo:

```
byte diaPeticao;
```

No exemplo citado, criamos um rótulo chamado diaPeticao e o tipo a ser usado é o byte. Em outras palavras, acabamos de alocar um espaço em memória, para guardar um valor concreto do tipo byte, em momento de execução. O valor alocado em memória pode ser alterado com o símbolo de atribuição ("="), na linha de declaração ou nas linhas seguintes do programa, mas o tipo e o rótulo se mantêm sempre os mesmos. Por exemplo:

```
byte diaPeticao = 13;
```

Note que, agora, diaPeticao está associado ao valor "13". Neste exemplo, podemos supor que o dia da petição de um caso judicial ocorreu no dia 13 do mês. Caso o dia da petição seja modificado para dia 27, podemos escrever a seguinte linha de código Java:

```
diaPeticao = 27;
```

Nesse caso, chamamos esses elementos de variáveis, pois o tipo e o rótulo são os mesmos, mas os valores em memória variam. Veja os seguintes exemplos de declaração de variáveis, com valores para cada tipo primitivo de dados que estudamos na seção anterior:

```
byte diaPeticao = 13;
short anoJulgamento = 2015;
int numeroDeProcessos = 45150;
long numeroProcesso = 20151234567890000 L;
float custoProcessual = 123.45 f;
double valorDaSentenca = 255000.70;
char decisao = 'A';
boolean recursoAceito = false;
```

2.2 RESTRIÇÕES E CONVENÇÕES PARA NOMES

Quando falamos de programação, o primeiro pensamento que vem à mente é o de escrever código. Todavia, é importante seguir convenções para nomes de variáveis, pois a tarefa de leitura de código também faz parte do dia a dia de um programador. Dentre as variáveis "*int i = 37*" e "*int idade = 37*", note que a segunda é mais fácil de ser compreendida. Na linguagem Java, não podemos utilizar palavras reservadas no nome das variáveis, como por exemplo "*int int = 37*". Segundo Martin (2020, p. 175), "*Software de excelente qualidade* significa código com um design simples e testado. É um código que não temos receio de modificar e que possibilita que os negócios respondam de modo rápido. É um código flexível e robusto".
Podemos considerar, então, que uma boa forma de trazer qualidade ao código de um programa é fazendo boas escolhas, como a adoção de convenções para a escolha de nomes de variáveis, a fim de melhorar a legibilidade do código. Uma forma de fazer isso, segundo Mayer (2015) é: "Um nome deve poder ser lido na sua língua nativa, ou seja, nada de abreviaturas que criarão palavras inexistentes e sem nexo, não é necessário estabelecer abreviaturas de tipagem junto ao nome definido, pois o código deve ser simplificado, que não permitirão que seja encontrada com facilidade dentro do código, causando problemas nos momentos da manutenção dos programas".
Começamos a estudar as estruturas presentes na linguagem de programação Java, com a preocupação de ter um código que facilite a manutenção. Ou seja, escrevemos código que, no futuro, poderá ser utilizado por outros programadores. Estudaremos todas as estruturas da linguagem Java em breve, mas, por ora, podemos adotar um conjunto simples de recomendações para melhorar significativamente nosso código (LAMOUNIER, 2021):

- Variáveis: utilizar nomes significativos e posicioná-las próximas ao local principal de uso.
- Classes: utilizar substantivos simples e evitar verbos.
- Métodos: utilizar verbos, minimizando o número de caracteres, tanto quanto possível.

Por fim, uma prática comum em Java é o uso de Cammel Case, onde um rótulo começa com letra minúscula para uma palavra. Caso haja mais de uma palavra no rótulo, a primeira letra de todas as palavras concatenadas é escrita maiúscula e o restante, em letras minúsculas. Vamos recuperar um exemplo anterior que faz uso do Cammel Case no Java:

```
double valorDaSentenca = 255000.70;
```

Note que as letras "D" e "S" são maiúsculas, pois estão concatenadas. Isso melhora a legibilidade da variável.

2.3 COMENTÁRIOS

O uso de comentários em um programa pode auxiliar na compreensão de suas linhas de código, especialmente para trechos complexos. Um comentário não é executado pela máquina. Quando um programa é compilado, o comentário não é traduzido em comandos executáveis. Em outras palavras, um comentário é apenas uma informação textual, para que programadores possam ler. No Java, utilizamos duas barras ("//") para comentar toda a informação textual que segue as barras. Por exemplo:

```
char decisao = 'A'; // 'A' absolvição, 'C' condenação, 'I' indeferimento
```

Note que a declaração da variável char decisão ocorre normalmente, e o valor 'A' é alocado nesta variável. O comentário vem em seguida, com o objetivo de descrever o que acontece nessa linha. Neste caso, sobre a variável decisão e o valor 'A', que significa absolvição. Além disso, podemos ver que 'C' significa condenação e 'I' indeferimento.
É possível comentar várias linhas de código, utilizando "//" em todas as linhas a serem comentadas, conforme exemplificado. Entretanto, existe ainda uma forma de comentar blocos de código por completo, utilizando um símbolo ("/*"), para indicar o início do bloco a ser comentado e outro para indicar onde o bloco termina ("*/"). No exemplo a seguir, todas as três variáveis estão comentadas e não seriam executadas:

```
/*
long numeroProcesso = 20151234567890000L;
float custoProcessual = 123.45f;
double valorDaSentenca = 255000.70;
*/
```

2.4 OPERADORES

Além dos símbolos de comentário, vários outros são utilizados com frequência, durante a construção de um programa. Um exemplo seria o operador de atribuição, que atribui um valor a uma variável. O símbolo que utilizamos para esse operador é o "=", que já usamos algumas vezes nos exemplos anteriores.

```
int numeroDeProcessos = 15; // Atribuimos o valor 15 para numeroDeProcessos
```

Outros operadores muito utilizados são os operadores aritméticos, que são representados em Java através dos mesmos símbolos matemáticos: "+" para adição; "-" para subtração; "*" para multiplicação; e "/" para divisão. Outros operadores matemáticos podem ser usados, como "%", para calcular o módulo do resto de uma divisão. Nas linhas que seguem, dividimos o valor da sentença por 2 e depois subtraímos um valor específico:

```
double valorDaSentenca = 255000.70 / 2; // o resultado será 127500.35
valorDaSentenca = valorDaSentenca - 500; // o resultado será 127000.35
```

Utilizar dois símbolos de adição antes ou depois de uma variável representa um incremento – aumento do valor da variável em 1. De maneira semelhante, utilizar dois símbolos de subtração, antes ou depois de uma variável, representa um decremento – diminuição do valor da variável em 1. Posicionar um incremento antes da variável significa realizar a soma antes de avaliar a expressão, enquanto depois significa realizar a soma depois de avaliar a expressão. Isso só será importante se fizermos a operação dentro de um espaço de checagem, que veremos em breve.

```
numeroDeProcessos++; // aumenta o valor de numeroDeProcessos em 1
diaPeticao--; // diminui o valor de diaPeticao em 1
```

Alguns operadores podem ser utilizados para realizar comparações entre diferentes valores, algo fundamental na construção de programas. Dentre eles estão:

- ">": verifica se um valor é maior que outro
- ">=": verifica se um valor é maior ou igual a outro
- "<": verifica se um valor é menor que outro
- "<=": verifica se um valor é menor ou igual a outro
- "==": verifica se dois valores são iguais
- "!=": verifica se dois valores são diferentes

Os resultados de uma comparação sempre retornarão *true* (verdadeiro) ou *false* (falso), isto é, um tipo primitivo booleano. Observe os exemplos a seguir, levando em consideração que a variável numeroDeProcessos possui um valor igual a 15:

```
boolean numeroDeProcessos == 10; // resulta em false
boolean numeroDeProcessos > 10; // resulta em true
boolean numeroDeProcessos < 10; // resulta em false
boolean numeroDeProcessos != 10; // resulta em true
```

Finalmente, temos três principais operadores lógicos. O primeiro é o E lógico (AND), que utiliza o símbolo "&&". Quando temos um operador AND, ambos os valores devem ser verdadeiros, para que o resultado seja verdadeiro. Nos demais casos, o resultado é falso. O segundo é o OU lógico (OR), que utiliza o símbolo ||. Quando temos um operador OR, basta um valor ser verdadeiro, que o resultado é verdadeiro. No caso de todos os valores serem falsos, o resultado é falso. O último é a negação lógica (NOT), que utiliza o símbolo "!". Nesse caso, quando o valor é verdadeiro, o resultado é falso e, quando o valor é falso, o resultado é verdadeiro. Veremos alguns exemplos de uso desses três comparadores na próxima seção: expressões.

2.5 EXPRESSÕES

O simples uso de uma operação representa uma expressão, como uma comparação, uma adição e assim por diante. Consideramos também expressões as combinações simples ou complexas de operações. Ou seja, o resultado de uma operação pode ser utilizado em outra operação, em uma mesma linha de código. Em casos mais complexos, precisamos fazer uso de parênteses, para definir qual operação ocorre primeiro ou para auxiliar na legibilidade.

```
float custoProcessual = 15 f;
double valorDaSentenca = 450.99;
boolean recursoAceito = true;
double expressao1 = custoProcessual + valorDaSentenca;// resulta em 465.99
boolean expressao2 = recursoAceito && (valorDaSentenca < 500);
boolean expressao3 = !recursoAceito || (valorDaSentenca < 500);
```

Para o exemplo apresentado, a primeira expressão é representada por uma simples operação de adição. Na segunda expressão, primeiro avaliamos o valor da variável recursoAceito, que é verdadeiro. Como temos um operador "&&", precisamos avaliar se o valorDaSentenca é menor que 500. Como ambos são verdadeiros, o resultado é verdadeiro. Na terceira expressão, temos a negação da variável booleana recursoAceito. Como recursoAceito é verdadeiro, a negação de recursoAceito é falso. Em outras palavras, queremos saber se o recurso não foi aceito. Precisamos avaliar o restante da expressão, valorDaSentenca menor que 500, para saber o resultado de "||". Falso ou verdadeiro avalia para verdadeiro, resultado da execução dessa expressão.

Note que operadores aritméticos serão avaliados com a mesma precedência definida na matemática. Multiplicação acontece antes de soma, por exemplo. Na execução de uma expressão complexa com parênteses, a operação dos parênteses mais internos é executada primeiro e assim por diante, até que a última operação seja executada. Ou seja, parênteses precedem a avaliação normal. Veja os exemplos a seguir:

```
double valorDaSentenca = 500 + 50 * 2; // resultado é 600
double valorDaSentenca = (500 + 50) * 2; // resultado é 1100
```

2.6 SEQUÊNCIAS DE CARACTERES (JAVA STRING)

Os tipos primitivos de dados que aprendemos podem nos ajudar a representar em código muitos dados encontrados no mundo real, como idade, gênero, saldo etc. Por outro lado, ainda precisamos de uma forma para representar informações textuais, como nome, endereço e assim por diante. A linguagem Java possui algumas classes que nos ajudam em determinadas necessidades de programação. Uma delas é a classe String, que pode ser utilizada para armazenar informações textuais. Veja um exemplo de criação de uma String “Maria”, que é atribuída a uma variável do tipo String chamada “nome”:

```
String nome = “Maria”;
```

Podemos dizer que a String “Maria” é imutável, pois, uma vez definida, ela não pode mudar. No entanto, da mesma forma que podemos usar operadores em tipos primitivos, podemos efetuar algumas operações sobre uma String. Por exemplo, podemos utilizar um método chamado *length* (em português, “comprimento”), para descobrir o número de caracteres em uma String, concatenando “.length()”, ao final da String. Iremos estudar classes e métodos muito em breve. Por ora, vamos observar esta operação, sendo realizada na variável do exemplo anterior:

```
int tamanhoNome = nome.length();// resulta em 5, pois “Maria” tem 5 letras
```

Outra operação comumente aplicada em Strings é a concatenação através do símbolo “+”. Ao concatenar duas ou mais Strings, uma nova String é criada como resultado da concatenação, mantendo as informações das Strings originais:

```
String nomeCompleto = nome + “Navarro”; // resulta em “Maria Navarro”
```

Algumas operações avançadas também podem ser utilizadas, como o método *contains* – que retorna verdadeiro, se uma String está dentro de outra; o método split – que divide uma String em múltiplas Strings; e o método *equals* – que pode ser utilizado para checar se duas Strings são idênticas.

```
String logradouro = "Rua Arco Íris da Felicidade, 500";
boolean contemRua = logradouro.contains("Rua"); // resulta em verdadeiro
boolean saoIguais = logradouro.equals("Rua Felicidade, 500"); // resulta em falso
```

2.7 DADOS E OPERAÇÕES MATEMÁTICAS (JAVA MATH)

Em adição à classe String, outra classe da linguagem Java que nos auxilia em determinadas tarefas de programação é a classe Math. Essa classe possui alguns métodos matemáticos avançados que podemos utilizar para realizar, com mais facilidade, algumas operações matemáticas, como: o método abs, que calcula o valor absoluto de um número; os métodos *min* e *max*, que retornam o menor e maior valor entre dois números, respectivamente; e os métodos *floor* e *ceil*, que calculam o piso e o teto de um número, respectivamente.

```
double saldo = -15.15;
double saldoAbsoluto = Math.abs(saldo); // resulta em 15.15
double saldoAbsoluto = Math.ceil(saldoAbsoluto); // resulta em 16
double menor = Math.min(saldo, saldoAbsoluto); // resulta em -15.15
double maior = Math.max(saldo, saldoAbsoluto); // resulta em 16
```

A classe Math também conta com o método "random", para gerar números aleatórios e métodos para operações trigonométricas, logarítmicas, exponenciais e outras operações aritméticas básicas, como calcular a raiz quadrada de um número.

2.8 DADOS E OPERAÇÕES LÓGICAS (JAVA BOOLEAN)

Uma última classe que estudaremos neste momento é a classe Boolean. De forma semelhante ao tipo de dado primitivo boolean, uma variável da classe Boolean, chamada de classe *Wrapper* do tipo boolean, pode ser utilizada para guardar os valores booleanos verdadeiro ou falso. Todavia, por meio da classe Boolean, podemos representar o valor *null* (em português, "nulo"). Esse valor não é verdadeiro nem falso, mas sim um terceiro valor. Utilizamos esse valor quando queremos representar que não é possível saber se algo é verdadeiro ou falso, ou quando ainda não sabemos se algo é verdadeiro ou falso, mas podemos descobrir posteriormente.
Outra diferença no uso da classe Boolean e do tipo primitivo boolean é que a classe Boolean possui métodos adicionais, como uso de Strings e comparações. Vale mencionar, entretanto, que o tipo primitivo é mais eficiente, pois utiliza menos memória.

2.9 TYPE CASTING

A linguagem Java conta com um conceito de programação chamado *type casting*, em que um tipo de dado é convertido para outro tipo de dado. Esse conceito é usado quando precisamos passar a utilizar um tipo diferente de dado e pode acontecer de forma automática ou explícita. Por exemplo, imagine que precisamos mudar a variável "idade", que era do tipo inteiro, para um tipo double, a fim de permitir casas decimais.

```
int idade = 34;
double idadeNova = idade;
```

No exemplo apresentado, note que o valor inteiro "34" é atribuído à variável idade, do tipo inteiro. Em seguida, colocamos o valor da variável idade na variável do tipo double. Ocorre aí um *type casting* implícito, onde um tipo int passa a ser double. Isso pode ser feito explicitamente, colocando o novo tipo entre parênteses e antes da variável:

```
double idade = 34.5;
int idadeNova = (int) idade;
```

Veremos que *casting* pode ser feito também para classes, como a classe String. Estudaremos classes em maior profundidade, além de outros conceitos de programação orientada a objetos básicos, como métodos e atributos, mas antes disso vamos estudar alguns comandos básicos: os comandos condicionais e os comandos de iteração.

3 COMANDOS CONDICIONAIS

O processo de construção de um programa consiste em escrever linhas de código que buscam atender um determinado passo a passo. No exemplo de registro de casos judiciais, podemos encontrar condições que alteram a forma como registramos um caso. Por exemplo, o processo de petição pode ser diferente para pessoas físicas e pessoas jurídicas. Ao construir um programa, será necessário fazer uso de comandos condicionais, para tratar essas situações.

3.1 COMANDO IF...ELSE

A forma mais simples de checar o valor de uma expressão lógica, para decidir que caminho seguir em um programa, é o comando if...else (ou se...então, em português). A estrutura desse comando conta com uma checagem no início e dois blocos de execução delimitados por chaves ("{}"), o bloco *if* e o bloco *else*. Caso o resultado da checagem seja verdadeiro, o primeiro bloco é executado. Se não, o segundo bloco é executado. Em outras palavras, executar as linhas de código do primeiro bloco significa que as linhas do segundo bloco não serão executadas.

Executar as linhas do segundo bloco significa que as linhas do primeiro não foram executadas. Note, no exemplo a seguir, que estamos checando o valor da variável "pessoaFisica", que tem valor *false*.

```
boolean pessoaFisica = false;
if(pessoaFisica) {
    // Primeiro bloco de linhas de código, para pessoas físicas
} else {
    // Segundo bloco de linhas de código, para pessoas jurídicas
}
```

A execução do programa acontece linha por linha, de cima para baixo, até alcançar a linha do *if*. Quando a execução do programa chega à linha do *if*, checa-se o valor da variável "pessoaFisica". Caso o resultado fosse verdadeiro, o primeiro bloco do *if* iria executar. Nesse caso, sabemos que o valor é falso, então as linhas do segundo bloco são executadas. Vale ressaltar que as linhas do primeiro bloco não são executadas. Veja outro exemplo desse comando:

```
int idade = 17;
float custoProcessual = 75.50f;
if(idade > 60) {
    custoProcessual = custoProcessual + 25;
} else {
    custoProcessual = custoProcessual + 50;
}
```

A variável custoProcessual é inicializada com valor 75.50. Em seguida, checamos se a idade é maior que 60. Ao verificar se 17 > 60, vemos que o resultado é falso e seguimos para execução do bloco *else*, realizando uma adição de 50, na variável custoProcessual. Ao final da execução desse programa, será atribuído o valor 125.50 à variável custoProcessual. Note que, se o valor de idade fosse 67, por exemplo, iríamos executar o bloco do *if*, em vez disso, e a variável custoProcessual teria 100.50, ao final da execução. Comandos *if...else* podem ser aninhados e possuir expressões complexas, em suas condições. Para o exemplo a seguir, ache o resultado atribuído à variável custoProcessual:

```
int diaPeticao = 7;
int anoJulgamento = 2010;
double custoProcessual = 0;
if(anoJulgamento > 2019) {
    custoProcessual = 150;
} else {
    if(diaPeticao > 15) {
        custoProcessual = 135;
    } else {
        custoProcessual = 125;
    }
}
```

O valor final atribuído à variável custoProcessual, depois da execução do código apresentado, é 125. O uso excessivo de comandos *if...else* aninhados pode tornar o código muito grande ou prejudicar a legibilidade. Vamos estudar alguns outros comandos condicionais na linguagem Java.

3.1.1 COMANDO SWITCH...CASE

Outro comando condicional que realiza um controle de fluxo é o *switch...case*. Nesse comando, o primeiro passo é realizar uma checagem de uma expressão lógica. Diferente do *if...else* que tem dois blocos, o *switch...case* possui um número de blocos igual ao número de casos resultantes da checagem. Para cada caso de resultado distinto, definimos um bloco de execução. Veja o exemplo a seguir:

```
int numeroDependentes = 3;
double valorDaSentenca;
switch(numeroDependentes) {
    case 0:
        valorDaSentenca = 0;
        break;
    case 1:
        valorDaSentenca = 1500;
        break;
    case 2:
        valorDaSentenca = 1700;
        break;
    case 3:
        valorDaSentenca = 1900;
        break;
    default:
        valorDaSentenca = 2100;
}
```

Nesse exemplo, estamos verificando a variável numeroDependentes. Caso essa variável tenha valor igual a zero, o "case 0" seria executado. Caso tivesse valor 1, o "case 1" seria executado e assim por diante. Note que numeroDependentes é 3, então o caso 3 é executado e o valorDaSentenca fica com valor igual a 1900. O caso default é executado quando nenhum outro caso é atendido. Um detalhe importante do *switch...case* é que, sem os comandos *break*, ao entrar em um caso, todos os seguintes são executados também, algo que, em geral, pode ser indesejável. Ainda nesta seção falaremos um pouco mais do comando *break*.

3.1.2 EXPRESSÃO CONDICIONAL TERNÁRIA

Podemos dizer que a condicional ternária é uma versão compacta do comando condicional *if...else*. Nesse caso, podemos verificar o valor de uma expressão e definir um retorno de valor, caso seja verdadeiro, e outro valor, caso seja falso, tudo em uma única linha. Fazemos isso por meio de dois símbolos: "?", para fazer a checagem; e ":", para separar o primeiro retorno (verdadeiro) do segundo retorno (falso). Um exemplo de expressão condicional ternária seria:

```
String resultado = (recursoAceito)? "Recurso aceito" : "Recurso não foi aceito";
```

O primeiro passo na execução dessa expressão ternária é avaliar a variável booleana recursoAceito. Caso ela seja verdadeiro, atribuímos a String "Recurso aceito" à variável resultado. Caso contrário, atribuímos a String "Recurso não foi aceito".

3.2 COMANDOS DE ITERAÇÃO

O último tipo de comando básico que precisamos estudar são os comandos de iteração. Como diz o nome, esses comandos realizam iterações, processo onde repetimos a execução de algumas linhas de código de um programa.
Imagine que precisamos visitar todos os processos de um tribunal e gerar um número aleatório. Se o tribunal tem cinco casos, podemos criar cinco conjuntos de linhas iguais, cada um usando o método *random* da classe Math, para gerar um número aleatório. E se o tribunal tivesse 45 mil casos? Criaríamos 45 mil conjuntos de linhas iguais no programa? Aprenderemos agora a alternativa mais apropriada, os comandos de iteração, em que criamos apenas algumas linhas e repetimos sua execução até atingirmos uma condição de parada.

3.2.1 COMANDO WHILE...LOOP

A primeira estrutura de repetição que veremos é o *while*. Com esse comando, criamos um bloco de execução que será repetido até que a condição de parada, representada por uma expressão lógica, seja falso. Precisaremos de uma variável auxiliar "i", para contar quantas vezes repetimos o laço. Apesar de nossas convenções afirmarem que uma variável deve ter nome descritivo e significativo, quando falamos de loops (em português, "laços de repetição"), é comum utilizarmos a letra "i" para a variável auxiliar, pois se refere ao índice atual.

```
numeroDeCasos = 45000;
int i = 1; // índice do caso atual
while(i <= numeroDeCasos)  {
    double numeroAleatorio = Math.random();
    i++;
}
```

Através do comando *while*, iniciamos com uma checagem se *i* é menor ou igual a *numeroDeCasos*. O valor de *i*, inicialmente, é 1, então o resultado é verdadeiro. Quando a checagem resulta em "verdadeiro", executamos o bloco do *while*. Nesse caso, a primeira linha do bloco gera um número aleatório e a segunda incrementa a variável *i*.
Ao chegar ao fim do bloco do *while*, retornamos para a primeira linha do laço de repetição – a linha de checagem. Isso sempre acontecerá, até que a checagem resulte em "falso". Note que, toda vez que executamos o bloco *while*, aumentamos o valor de *i* em 1. Ou seja, depois de 45 mil execuções, o valor de *i* aumentou em 45 mil. Logo, na última execução do *while*, iremos checar se 45001 é menor que 45000, o que resulta em "falso" e o laço de repetição se encerra. Podemos, então, concluir que, para este exemplo, apesar do bloco *while* possuir apenas duas linhas, estas foram executadas 45 mil vezes.

3.2.2 COMANDO FOR...LOOP

Possivelmente o mais utilizado, o laço de repetição *for* tem a mesma intenção do comando *while*, mas possui uma estrutura diferente. Neste comando, é comum fazermos três definições, em uma única linha: a variável auxiliar, a condição de parada e a taxa de crescimento da variável auxiliar. Observe a diferença quando utilizamos o comando *for*, para resolver o mesmo problema no exemplo do comando *while*:

```
numeroDeCasos = 45000;
for(int i = 1; i <= numeroDeCasos; i++)  {
    double numeroAleatorio = Math.random();
}
```

O bloco do laço de repetição *for* tem apenas uma linha, a geração do número aleatório. Note que a definição da variável auxiliar e do incremento acontecem na primeira linha, a mesma linha em que ocorre a checagem. Para este exemplo em específico, a primeira checagem irá conferir se 1 é menor que 45 mil. O resultado é verdadeiro, então executamos o bloco. Após a execução da única linha que temos no bloco, o fluxo de execução retorna para o laço de repetição. Note que houve um incremento no valor de *i*, então a segunda checagem irá conferir se 2 é menor que 45 mil. Isso irá acontecer até que o bloco do laço tenha repetido 45 mil vezes.
Conseguimos realizar a mesma computação que fizemos com o *while*, mas com um número menor de linhas. A maior diferença entre o *for* e o *while* é que a variável auxiliar *i* no *for* só pode ser usada dentro do laço de repetição, enquanto no *while* a variável auxiliar *i* pode ser usada depois que o laço for encerrado. Em geral, podemos dizer que o *for* é o laço de repetição mais popular.

3.2.3 COMANDO DO...WHILE

O comando *do...while* é extremamente semelhante ao comando *while*, mas possui a diferença de sempre executar o bloco do laço de repetição, pelo menos uma vez. Em outras palavras, este comando tem uma estrutura muito semelhante à do comando *while*, mas a checagem só ocorre no final. Consequentemente, podemos dizer que o primeiro laço de repetição sempre será executado. Veja como utilizamos o comando *do...while* neste mesmo cenário:

```
numeroDeCasos = 45000;
int i = 1; // índice do caso atual
do {
    double numeroAleatorio = Math.random();
    i++;
} while(i <= numeroDeCasos);
```

3.2.4 COMANDOS BREAK E CONTINUE

Um importante aspecto dos laços de repetição é que queremos repetir a execução de determinadas linhas de código. Todavia, podem existir restrições específicas ou condições adicionais que exijam um controle adicional de fluxo. Por exemplo, caso o número aleatório do Math.random() seja zero, eu gostaria de encerrar meu laço de repetição prematuramente. Podemos fazer isso através do comando *break*:

```
numeroDeCasos = 45000;
for(int i = 1; i <= numeroDeCasos; i++)  {
    double numeroAleatorio = Math.random();
    if(numeroAleatorio == 0) {
        break;
    }
}
```

O comando *break* interrompe o laço de repetição atual e termina a execução do *loop*, sem que o *loop* volte a checar a condição de parada. Em outras palavras, este comando encerra as atividades do atual comando de iteração. O comando *break* pode ser usado em diferentes *loops* além do *for*, como o *while* e o *do...while*. E até mesmo ser utilizado no comando condicional *switch...case*.

Um último comando que pode nos ajudar a controlar o fluxo de um laço de repetição é o comando *continue*. Este comando faz com que o fluxo de execução prossiga imediatamente para o próximo laço de repetição. Por exemplo, se eu quiser evitar que um número aleatório seja gerado para processos quando o número for par:

```
numeroDeCasos = 45000;
for(int i = 1; i <= numeroDeCasos; i++)  {
if(i % 2 == 0) {
        continue;
    }
    double numeroAleatorio = Math.random();
}
```

Nesse exemplo, toda vez que executamos o bloco do laço de repetição, primeiramente fazemos uma checagem se o resto da divisão da variável auxiliar *i* por 2 é igual a zero. Isto é, estamos verificando se o número é par. Caso o número seja par, o comando *continue* pula para o próximo laço de repetição, evitando, assim, que a linha de geração do número aleatório seja executada.

Na próxima seção, aprenderemos o que é programação orientada a objetos e como resolver problemas mais complexos. Lembre-se de fazer todos os exercícios contidos no caderno de atividades e praticar o conteúdo presente no material. O sucesso no aprendizado está na replicação prática de todos os conceitos abordados no material de estudo.

REFERÊNCIAS

DEITEL, Paul; Deitel, Harvey. **Java**: Como Programar. 10ª edição. Editora Pearson Universidades, 2016.

FERREIRA, Arthur G. **Interface de programação de aplicações (API) e web services**. Editora Saraiva, 2021.

FURGERI, Sérgio. **Java 7** - Ensino Didático. Editora Saraiva, 2012.

LAMOUNIER, Stella Marys D. **Qualidade de software com Clean Code e técnicas de usabilidade**. Editora Saraiva, 2021.

MACHADO, Rodrigo P.; FRANCO, Márcia H I.; BERTAGNOLLI, Silvia de C. **Desenvolvimento de software III**: programação de sistemas web orientada a objetos em java (Tekne), Editora Bookman, 2016.

MARTIN, Robert C. **Desenvolvimento Ágil Limpo**. Alta Books, 2020.

MAYER, Carlos R.; MAZER, Ademir Jr. **Visão Introdutória sobre os conceitos do código limpo**. Revista Científica Semana Acadêmica. Fortaleza, ano MMXV, Nº. 000071, 2015.

OLIVEIRA, Diego Bittencourt de. *et al*. **Desenvolvimento para dispositivos móveis**. SAGAH, 2019.

PRESSMAN, Roger S.; MAXIM, Bruce R. **Engenharia de software**. Editora AMGH, 2021.

ROSEN, Kenneth H. **Matemática Discreta e suas Aplicações**. 8ª edição. McGraw Hill, 2018.

SCHILDT, Herbert. **Java para iniciantes**. Editora Bookman, 2015.

SEBESTA, Robert. **Conceitos de Linguagens de Programação**. Editora Bookman, 2018.

SOMMERVILLE, Ian. **Engenharia de software**. Tradução Maurício de Andrade. 9. ed. São Paulo: Pearson Education do Brasil; Addison-Wesley, 2011.