

02

## A plataforma Java

### Transcrição

Antes de mais nada, vamos ver um pouco do que é o Java, o qual te trouxe até aqui: há cerca de vinte anos, quando a linguagem Java nasceu, ela chamava a atenção por conta das seguintes características:

- Orientado a Objeto (O.O.)
- Muitas bibliotecas
- Parece com C++ (hoje em dia isso pode até ser uma desvantagem)
- Roda em vários sistemas operacionais

Você pode estar pensando "poxa, mas a linguagem que uso no dia a dia, atualmente, já possui estas características!". É verdade. É por isto que queremos focar na **plataforma Java**, e não especificamente na linguagem em si, algo que ficará mais claro no decorrer do curso, e até mesmo nesta aula!

A plataforma Java traz:

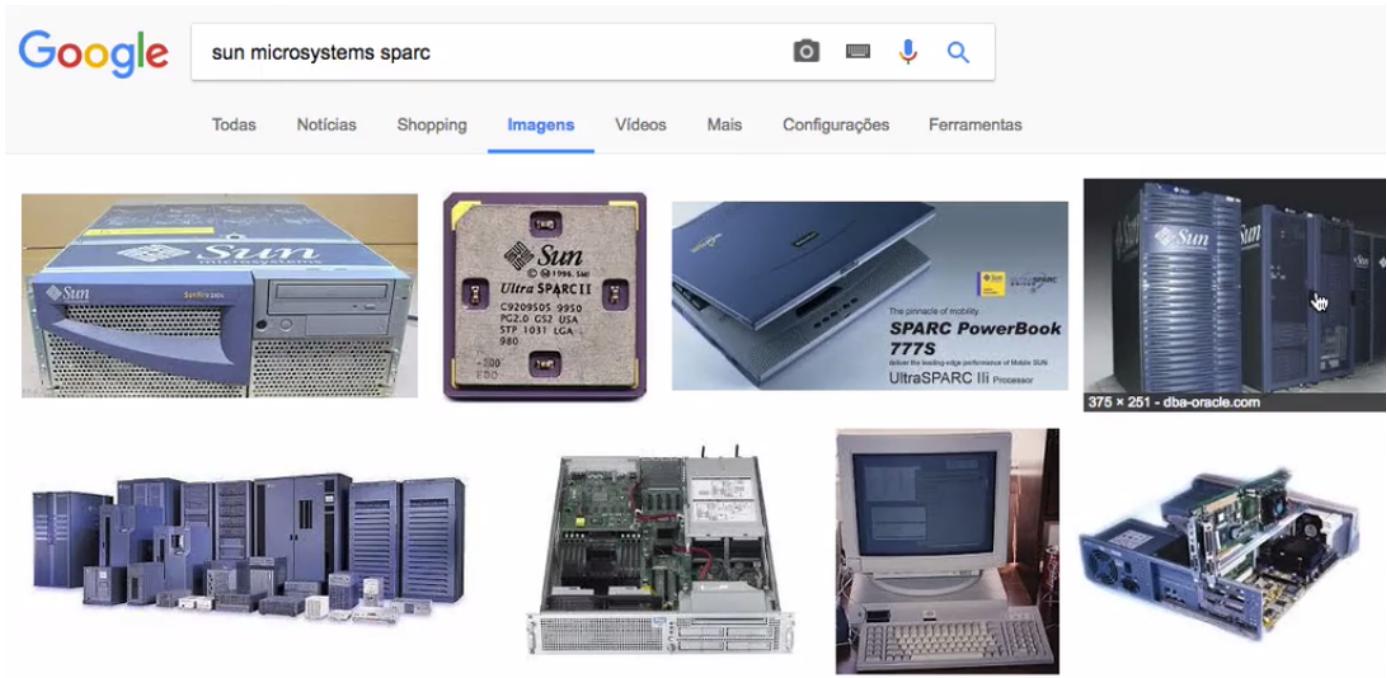
- Portabilidade
- Fácil acesso e desenvolvimento
- Segurança
- Onipresença

Você pode dar uma olhada no [site oficial \(<https://java.com>\)](https://java.com), porém ele ajuda mais o usuário do Java, do que aqueles que irão compilar e escrever programas.

Falando sobre a história da linguagem: James Gosling é considerado um dos "gênios da computação", sendo considerado o "pai do Java", apesar da linguagem ter sido criada por um grupo, normalmente considerado de quatro pessoas.

Em 1992, o James Gosling trabalhava em uma empresa atualmente inexistente chamada Sun Microsystems (sendo que Sun é acrônimo para *Stanford University Network*), uma dessas *startups* da década de 60, 70, para lidar mais com hardware, que é o que estava dando mais dinheiro.

Eles possuíam um microcomputador, o **Sun Microsystems SPARC**, que hoje em dia já não aparecem em lugar algum, grandes servidores denominados "micro":



Sendo a Sun uma empresa mais focada em hardware, naquela época, a IBM e a Microsoft começaram a crescer vendendo softwares. Os softwares que a Sun utilizava no sistema deles, o UNIX (o tal de Solaris), eram disponibilizados gratuitamente.

Um dia, esses executivos, dentre os quais o próprio James Gosling, se perguntaram como poderiam lucrar com softwares, já que eles o disponibilizavam de graça, e fizeram um retiro de um mês para tentarem chegar a uma conclusão.

A ideia que eles tiveram envolvia um problema de eletrônicos da década de 90: havia muitos deles sendo criados naquela época, como o VHS que, para quem não sabe, é o videocassete. Era a época de surgimento de TVs, videogames, liquidificadores e geladeira.

Cada um deles possui seu código fonte, necessitando de uma linguagem própria para funcionar, e escrever o código para cada um, reescrevendo-o quando tivessem que passar por uma troca de chip, por exemplo, não fazia muito sentido! A linguagem utilizada neles, ***Assembly***, que hoje em dia é raramente usada, precisava ser reescrito várias vezes, imagine o trabalho!

O James Gosling e sua equipe pensaram em escrever um único código que gerasse um "executável" - entre aspas porque após a compilação ele estará em um formato não exatamente comprehensível pelo aparelho em si, mas por um intermediário, no caso, um processador ou uma placa de hardware, para que, aí sim, passe o código aos aparelhos.

Trata-se de algo que realmente simula um computador bem simples e traduz esta linguagem "executável" de acordo com o aparelho em questão. Isto é, esta "máquina de mentira" traduzirá tudo, como se fosse um sistema operacional.

É por isto que surgiu o nome **máquina virtual**, pois veio da ***virtual machine***!

A ideia deles foi, então, criar uma placa pequena, um hardware, que é uma máquina real e compõe todo liquidificador, computador, videocassete, e por aí vai. Desta forma, as pessoas poderão escrever em apenas uma linguagem, que na época se chamava ***Oak*** e depois se tornou Java.

Isso pareceu muito bom, mas acabou fracassando de maneira retumbante, pois era muito caro produzir chips distintos para cada aparelho, cada qual adaptado a uma determinada linguagem.

Então, em 1995, com o *boom* da Web e o surgimento de mais navegadores, como Mosaic, Netscape e posteriormente Internet Explorer, a ideia de máquina virtual foi visualizada como um problema interessante pelo Gosling.

Assim como na atualidade, existia uma variedade relevante de navegadores e sistemas operacionais. E, para escrever um código para Windows, utilizava-se a linguagem no Microsoft Visual Basic, que por sua vez era compilado por um executável (um EXE, no caso do Windows).

Isto é, ele só funciona neste sistema operacional, com determinadas DLLs na máquina, e assim por diante. O executável e o código fonte ficavam atrelados a uma plataforma específica, um conjunto de sistema operacional, hardware e outros detalhes.

Para tentar resolver este problema, que geraria um código e um executável diferentes para cada sistema operacional existente, o Gosling desengavetou a ideia da máquina de verdade, do chip, que eles haviam criado anteriormente.

Com um código fonte único, teríamos um intermediário que soubesse traduzir ou instruir o sistema operacional acerca dos comandos a serem enviados e recebidos. Este meio de campo seria realizado pela **Máquina Virtual Java (JVM)**, que não é meramente um interpretador por conta de alguns detalhes internos que vão além da interpretação.

O código, então, seria a linguagem Java, e o código "executável", quando compilado, não geraria um .exe (pois este seria lido apenas pelo Windows), e sim um formato

chamado **bytecode Java**, de extensão `.class`, lido pela Máquina Virtual Java, que passaria a informação aos sistemas operacionais.

Um exemplo deste formato entendido pela *virtual machine* (JVM), o *bytecode*, é o seguinte:

```
Compiled from "Onibus.java"
class Teste {

    public static void main(java.lang.String);
        Code:
            0: new           #2  // class Onibus
            3: dup
            4: invokespecial #3 Onibus."<init>":()V
            7: astore_1
            8: aload_1
            9: ldc           #4 // String Jabaquara...
           11: putfield       #5
                // Field Onibus.linha:Ljava/lang/String;
           14: return

}
```

**COPIAR CÓDIGO**

Quem conhece a linguagem de **Assembly** talvez identifique a semelhança, mas este código não parece ser de fácil leitura e compreensão. Para meios de comparação, segue um exemplo de um arquivo `.java`, a ser compilado e traduzido para `.class`, o tal do bytecode:

```
public class Onibus {
    String nome;
    String linha;
}
```

```
class Teste {  
    public static void main(String args) {  
        Onibus o = new Onibus();  
        o.linha = "Jabaquara-Liberdade";  
    }  
}
```

[COPIAR CÓDIGO](#)

Então, em 1995 surgiu o Java, capaz de rodar em vários dispositivos e sistemas operacionais, com foco de criar *applets*, quando ainda tínhamos que instalar o Java para rodá-lo dentro do navegador.

O Java nasceu com um propósito, mas acabou se fortalecendo em *server-side*, pois quando escrevemos uma aplicação, um site web ou sistema grande, não queremos ficar dependendo de diferentes sistemas operacionais, em implantações e *deploys*.

O Java traz liberdade, quebrando nossa dependência em relação às versões de sistema operacional e navegadores. Empresas grandes, como bancos e o governo, não querem ficar engessados - o que é conhecido por *Vendor lock-in*.

As principais características do conceito de Máquina Virtual Java são:

- Multiplataforma
- Gerenciamento de memória
- Segurança
- Sandbox
- Otimizações
- JIT Compiler

Hoje, mais do que na linguagem Java em si, o enfoque está na plataforma, no **ecossistema Java!** A *virtual machine* é interessante para as empresas pois elas não dependem do que se encontra abaixo da sua *stack*, ou pilha de tecnologia, além do acesso a uma grande variedade de bibliotecas, e as linguagens Java que rodam nesta plataforma.

Não é à toa que há programas que lidam com linguagens Ruby, Clojure ou Scala, por exemplo, e geram o bytecode Java. Depois, basta a Máquina Virtual Java, JVM, trabalhar de acordo com o sistema operacional desejado.

 08

## Para saber mais: o nome Bytecode

Já falamos um pouco sobre o Bytecode que é um código de máquina parecido com o Assembly. Talvez você (como eu!) estranhou o nome *Bytecode*, no entanto, tem uma explicação bem simples para tal. Existe um conjunto de comandos que a máquina virtual Java entende. Esses comandos também são chamados de *opcodes (operation code)*, e cada *opcode* possui o tamanho de exatamente 1 Byte! E aí temos um **opcode de 1 Byte** ou, mais simples, **Bytecode**. :)

01

## Versões

### Transcrição

Você deve estar muito ansioso para instalar o Java e o ambiente de programação para compilar e executar seu primeiro programa!

Mas antes de todo este processo de instalação e configuração, gostaria de falar sobre **versões**, uma vez que é comum encontrarmos vários números e versões e ficarmos perdidos sem saber por onde iniciar no Java.

Apesar da última versão lançada ser o `9`, lançada em 2017, a linguagem, surgida em 1995, teve mudanças consideráveis na versão `5`, que saiu em 2004, e na `8`, de 2014. Nelas, apareceram muitos recursos na linguagem, novos comandos, palavras-chave e conceitos.

Estes tais de **Streams**, de **Templates Generics**, serão vistos durante o curso - há até um [curso específico sobre estes novos recursos do Java 8](#) (<https://cursos.alura.com.br/course/java8-lambdas>). Nas versões `9` e `7`, houve mudanças pequenas e pontuais, além de bibliotecas.

Então, não se preocupe, você pode, sim, focar na versão `8`, pois você verá que muitas empresas grandes inclusive ainda não alcançaram esta versão (o que é uma pena).

Aqui, usaremos a versão **Neon** do **Eclipse**, mas existe uma versão mais recente, **Oxygen**, que está sendo trabalhada para dar suporte ao Java `9`. Até o momento, 1

há versão oficial do *Eclipse* que dê suporte para a última versão disponível do Java.

Todos os conceitos focados neste curso, que envolvem Orientação a Objeto, uso da herança, polimorfismo e as principais bibliotecas, são os mesmos para muitas versões da linguagem.

Ou seja, a dica é focar naquilo que é importante, que é o que passaremos aqui, e não nas versões mais recentes. A versão 10, provavelmente virá com muito menos novidades, já que as versões seguirão a tendência de serem lançadas mais rapidamente, não de 3 em 3 anos, e sim de 6 em 6 meses.

Minha recomendação é a de que você siga os passos feitos neste curso, respeitando a instalação do Java 8 e do *Eclipse Neon*. No entanto, se você realmente quiser utilizar a versão mais recente de cada um deles, por sua própria conta e risco, vá em frente. É bem provável que você não encontre problemas!

Porém, se você é iniciante em programação e nunca viu Java antes, indica-se a utilização das versões citadas neste curso.

02

## Instalação do JDK no Windows

### Transcrição

Vamos instalar o Java e tudo aquilo de que precisamos para começarmos a trabalhar! Usando o Windows, iremos fazer uma busca no Google por "Java" para ver as opções fornecidas para download.

Um dos primeiros resultados mostrados é o [java.com](http://www.java.com) (<http://www.java.com>), com uma aparência um tanto ultrapassada, e o botão "Download Gratuito do Java", em português ou inglês. Indo por este caminho, você perceberá que baixará uma versão que costumamos usar para **rodar uma aplicação Java**.

Para nós, desenvolvedores, baixar isto não é o suficiente. A versão recomendada que aparece na página de download, no caso "Version 8 Update 121", é o que chamamos de ***Java Runtime Environment***, ou "ambiente de execução Java", que é necessário para **executar uma aplicação Java**.

Lembra da época dos *applets*, em que precisávamos instalar plugins e similares para serem rodados no browser, ou em aplicativos? É para estes casos que a instalação desse ambiente de execução serve, o tal do **JRE**, que vem com a *virtual machine* e as bibliotecas.

Como desenvolvedores, precisaremos do **JDK**, ou ***Java Development Kits***, o "kit para desenvolver aplicações Java".

Pesquisando no Google por "download java jdk" ou simplesmente "jdk", aí sim, cairemos em um link mais específico, como no da **Oracle**, com diversas opções. Queremos a versão 8 , ou outra mais recente.

Na descrição, lê-se "8u112", por exemplo, em que "u" indica "*update*", ou "atualização" em português, que tem a ver com correção de *bugs*. Nesta página, estão disponíveis para download o JDK, bem como o JRE, visto no link anterior.

A opção que queremos baixar trará, além da *virtual machine* e das bibliotecas, o compilador, que pegará o código Java e o transformará em um formato que ele entenderá. Vamos fazer o download do JDK de acordo com o sistema operacional - no Mac ou no Linux pode ser que já venha instalado, ou seja mais fácil de se baixar.

Neste caso, optaremos por `jdk-8u121-windows-x64.exe` , não esquecendo de aceitar os termos da Oracle. Terminado o download, abriremos o arquivo executável, a ser salvo em um diretório apropriado seguindo os passos de instalação no modo *default*.

O JDK, o compilador, nada mais é do que uma versão menor daquilo que existe no site [java.com \(http://www.java.com\)](http://www.java.com), **mais** as ferramentas para o desenvolvimento de aplicações Java. Em seguida, continuaremos instalando o JRE, com a *virtual machine* e tudo o mais que é necessário para rodar esta aplicação.

Confirmaremos a instalação acessando o prompt do MS-DOS, que é algo muito similar ao terminal do Linux, Bash, Shell, e do Mac. O PowerShell da Microsoft hoje em dia é mais raro, mas ainda existe. Não se preocupe, muito em breve estaremos utilizando uma IDE, o Eclipse. Neste momento, porém, queremos enxergar o que está "por trás".

Pode-se pesquisar por "cmd" para acessar o prompt, uma tela preta em que digitaremos comandos, sendo o primeiro deles aquele que chamará o Java, `java` ,

seguido da tecla "Enter". Ele retorna uma mensagem dizendo para usarmos um `class`, porém ainda veremos sobre.

O comando que usaremos em seguida será `javac`, de *java compiler*, o compilador que pegará o código em Java e "traduzirá" para o que a *virtual machine* entende. Porém, ao digitarmos isso, obteremos o seguinte:

'javac' não é reconhecido como um comando interno ou externo, um pr

[COPIAR CÓDIGO](#)



Você deve estar se perguntando o que aconteceu, já que o JDK foi instalado, e é verdade, ele deveria ser executável. O que acontece é que ele está em outro local, e por isto não está sendo encontrado.

Abrindo o explorador de arquivos, em `C:\`, "Arquivos de Programas" contém a pasta "Java", que por sua vez possui dois diretórios referentes a JRE (onde se encontra a *virtual machine*) e JDK (onde está o compilador). Por *default* de instalação, a Oracle modifica os arquivos de configuração do Windows e deixa apenas o Java do JRE pronto para ser chamado em linha de comando.

Se você for usar o Java em linha de comando, como não é tão raro de acontecer, precisaremos do "jdk1.8.0\_121", dentro do qual há, em "bin" (de "binário"), o arquivo `javac.exe`. Vamos selecionar o caminho do diretório onde se encontra este executável, e copiá-lo por meio do atalho "Ctrl + C" e, no Painel de Controle, informaremos ao Windows para que toda vez que inserirmos algum comando, o caminho abaixo seja consultado também:

`C:\Program Files\Java\jdk1.8.0_121\bin`

[COPIAR CÓDIGO](#)

No Painel de Controle, portanto, selecionaremos "Sistema > Configurações avançadas do sistema" e, na nova janela, clicaremos em "Variáveis de Ambiente...", utilizável por programas como se fossem variáveis globais do Windows.

Veremos no box abaixo de "Variáveis do sistema" o "Path". Clicaremos em "Editar..." e observaremos todos os seus componentes. Queremos incluir mais um diretório nele, portanto clicaremos em "Novo" e usaremos o atalho "Ctrl + V" para colar o caminho que copiamos anteriormente. Seleccionaremos "OK" em todas as janelas que ficaram abertas.

Teremos que reabrir o Prompt de Comando, após o qual digitaremos `javac`, que desta vez funcionará corretamente! Quando se instala uma nova linguagem de programação, é comum que a variável de ambiente seja alterada para que não haja necessidade de criarmos arquivos ou entrarmos em diretórios específicos para trabalhar.

A partir deste momento, então, temos não só o Java, mas o JDK, o Kit de Desenvolvimento do Java, instalado e configurado no Windows, tanto para execução quanto para compilação de uma aplicação Java!

 04

## Para saber mais: JVM vs JRE vs JDK

O mundo Java é cheio de siglas com 3 ou 4 letras começando com J. Você já conhece duas famosas: o **JRE** e **JDK**. O primeiro é o ambiente de execução, o segundo são as ferramentas de desenvolvimento *junto* com o ambiente de execução. Simplificando podemos dizer:

JDK = JRE + ferramentas de desenvolvimento

[COPIAR CÓDIGO](#)

Existe uma terceira sigla, **JVM** (*Java Virtual Machine*), que também já usamos durante o curso. A responsabilidade da Java Virtual Machine é executar o Bytecode! Então qual é diferença entre JVM e JRE? Ambos executam o Bytecode, certo?

A resposta é simples: O JRE (o nosso ambiente de execução) contém a JVM, mas também possui um monte de bibliotecas embutidas. Ou seja, para rodar uma aplicação Java não basta ter apenas a JVM, também é preciso ter as bibliotecas.

Assim podemos simplificar e dizer:

JRE = JVM + bibliotecas

[COPIAR CÓDIGO](#)

É importante entender que você não pode baixar a JVM apenas. Você sempre baixa o JRE que tem a JVM e as bibliotecas em conjunto.



05

## Compile e rode seu primeiro programa Java

### Transcrição

Como falamos no início, em um primeiro contato, o código em Java pode ser complicado de ser escrito e compreendido. Às vezes precisamos escrever um pouco mais do que gostaríamos para fazer algo.

Antes de usarmos um IDE para lidarmos com o código, é legal que você o faça em um sistema bem simples, como o bloco de notas - outras opções são o TextPad, Atom, Visual Studio Code, Sublime, ou qualquer outro.

Nosso primeiro código Java será feito no editor de texto mais simples possível, em *plain text*. Faremos o "Olá mundo" para testarmos e vermos como funciona a compilação e execução de programas Java.

O Java veio da linguagem C na década de 90, então, não é tão simples quanto digitarmos `print("olá mundo")`. A linha que faz um print na tela, por exemplo, é

```
System.out.println("olá mundo");
```

[COPIAR CÓDIGO](#)

Nesta linguagem, toda instrução que damos sem as chaves necessita do ponto e vírgula ( ; ). Todo código Java também precisa estar dentro de uma classe, que pode ser uma interface, um `Enum`. Neste caso, ele se insere na classe `Programa`.

Uma instrução como esta, com `System.out.println()`, precisa estar dentro de um método chamado `main`, que ainda não vimos, acompanhado de outros termos que também aprenderemos depois.

É muito comum o uso de `public` antes de `class Programa`, e embora isto não seja estritamente necessário no nosso caso, vamos colocá-lo para quando formos ler códigos de outros programadores e IDEs.

```
public class Programa {  
  
    public static void main(String[] args) {  
        System.out.println("olá mundo");  
    }  
}
```

[COPIAR CÓDIGO](#)

No momento, focaremos na linha `System.out.println("olá mundo");`, que poderá ser considerado um comando apesar de não ser um, e mostrará algo na saída padrão, no caso o prompt do MS-DOS.

O menor programa Java seria similar ao código acima. Vamos tentar ver como funciona sua compilação e execução? Antes disso, salvaremos o arquivo nomeando-o com "Programa.java", em uma nova pasta denominada "java-codigo".

O nome do arquivo é muito importante - entenderemos melhor o motivo mais adiante, mas ele precisa ser o mesmo da `class` inserida no código.

No Prompt de Comando, digitaremos `cd ..` duas vezes, seguidos de "Enter", e `dir`, para a listagem de todos os diretórios. Depois, usaremos `cd java-codigo` para acessar o diretório, e em seguida digitaremos `dir` novamente.

Dica: é possível usar a tecla TAB para autocompletar palavras!

Ali, é listado um arquivo "Programa.java"! No Windows, há um comando chamado type (equivalente ao cat do terminal do Linux), o qual permite a visualização do conteúdo do arquivo. Neste caso, usariamos type Programa.java .

A extensão .java não é entendida pela *virtual machine*, que entende o formato "meio máquina" de *Virtual Machine Java*, o **bytecode**, um arquivo com extensão .class .

A seguir, usaremos o comando javac Programa.java , e daremos um "Enter", com o qual serão mostradas as mensagens de erro de compilação, fundamentais para o aprendizado.

Apesar de não entendermos o que é public class ou static void main ainda, sabemos que System.out.println() seguido de aspas e o conteúdo, irá mostrar uma mensagem.

Por meio de dir no prompt, você verá que há dois arquivos: "Programa.java" e "Programa.class", este último no formato binário, em *bytecode*. E para chamarmos a *virtual machine*, usaremos o comando java Programa , e veremos a impressão de "olá mundo". Trata-se da primeira execução do nosso programa Java!

Agora, veremos os principais erros e características deste código. O primeiro surge ao digitarmos java Programa.class , o que traz a seguinte mensagem de erro na execução do programa:

Erro: Não foi possível localizar nem carregar a classe principal Pr

**COPiar CÓDIGO**

Isto acontece porque o programa não se chama "Programa.class", e sim simplesmente "Programa", apesar de estar contido no arquivo "Programa.class".

Outros erros mais comuns são os de compilação, como quando esquecemos de colocar o ponto e vírgula no fim da linha. Além disso, o Java possui palavras chave (*keywords*, ou palavras reservadas), dentre os quais utilizamos "public", "class", "static" e "void", que devem estar em letra minúscula, uma vez que o Java é **case sensitive** (reconhece o uso de letras maiúsculas ou minúsculas).

Em um ambiente mais complexo, veremos que isto ficará mais claro e fácil de ser trabalhado. É importante **praticar e não ter medo das mensagens de erro de compilação**.

As chaves abrem e fecham os blocos de códigos, indicando por exemplo que tudo aquilo que se encontra em `public static void main` pertence ao `public class Programa`, da mesma forma que `System.out.println()` pertence ao `public static void main` visível também por meio das indentações.

O Java possui outras particularidades, como o "Enter" e a barra de espaço serem opcionais; são convenções do código. Agora, o importante é escrever, entendendo o que está por trás do código, errar e fazer vários testes!

01

## Instalando o Eclipse

### Transcrição

Por enquanto, temos o nosso primeiro programa Java escrito, e agora passaremos a entender como declarar variáveis, fazer `if`, laços e afins. Queremos um editor um pouco melhor do que o bloco de notas, de acordo com sua preferência.

A comunidade geral do Java costuma usar não um editor, mas um IDE (*Integrated Development Environment*, que em português seria algo como "Ambiente Integrado de Desenvolvimento"). Um IDE não é simplesmente um editor pois integra em um único local a linguagem, o editor, o compilador, a biblioteca e a documentação.

Os principais IDEs utilizados por quem programa em Java são: o *NetBeans*, da própria Oracle, o *IntelliJ IDEA*, usado como base para Android, e também conhecido por *Android Studio*, e o *Eclipse*, projeto em código aberto absorvido pela IBM e, hoje em dia, um consórcio de muitas empresas que tomam conta do programa, que você pode baixar [aqui](https://www.eclipse.org/downloads) (<https://www.eclipse.org/downloads>).

O Eclipse quer te ajudar na hora de codificar, muito mais do que focar em *wizards* e na grande quantidade de opções de menu. À primeira vista, o IDE pode parecer pequeno demais (pelo peso que possui), mas é porque há muitos plugins instaláveis para se facilitar o desenvolvimento de *features* e recursos.

Quando formos instalá-lo, aparecerá uma janela perguntando o que queremos, e escolheremos "Eclipse IDE for Java Developers". A opção "Eclipse IDE for Java EE

"Developers" requer um conhecimento maior, e serve para desenvolvimento de aplicativos web e softwares, e poderá ser explorada futuramente.

Após instalação e durante a execução, a primeira pergunta que o Eclipse fará tem a ver com o *workspace*, o diretório a ser utilizado para guardar todos os projetos Java. Isto pode ficar a seu critério, lembrando que iremos trabalhar sempre no Eclipse, então isso acabará não sendo tão relevante, pois você não precisará mais do Prompt de Comando para acessá-lo.

É possível ter mais do que um *workspace*, um só para exercícios da Alura e outros para projetos da empresa, por exemplo.

A primeira execução trará muitas janelas diferentes, mesmo se fecharmos o "Help". Vamos maximizar o Eclipse e fechar a aba "Welcome".

O IDE, ao ser aberto, pode te assustar um pouco, mas você verá que assusta menos do que outros com muito mais janelas e perguntas de *wizards*. No centro, ficam os arquivos que queremos editar, do lado direito estão os "Task List" (Lista de Tarefas), embaixo, "Problems" (Problemas). À esquerda, há "Package Explorer" (Explorador de pacotes).

O Eclipse denomina este conjunto de janelas de **perspectiva**, e cada uma delas é uma **view**. Então, veremos diversas *views* que irão nos ajudar em diferentes situações, tanto que se clicarmos em "Window > Show View" no menu superior, há várias opções. Não nos preocuparemos com isso agora.

No momento, queremos criar um projeto Java, e veremos poucos *wizards*! Para criarmos um projeto e uma classe Java, clicaremos em "File > New > Java Project" e, na nova janela, definiremos o projeto como "sintaxe-basica".

Provavelmente o Java já está instalado em seu computador; verifique sua versão, se é 8 ou posterior, pois utilizaremos recursos desta versão. Clicando em "Finish", o projeto é criado e aparecerá em "Package Explorer", contendo um diretório "src" (onde deve estar nosso código fonte), e "JRE System Library", uma biblioteca com tudo que temos e acessível pelo Java. Todos os comandos que utilizaremos estarão nestes arquivos .jar.

Agora, queremos colocar nosso arquivo Java, o "Programa.class", no diretório de código fonte.

05

## Nosso programa rodando no Eclipse

### Transcrição

Vamos colocar o código do programa no Eclipse, utilizando o editor mais poderoso, o **IDE**. Para isto, há várias opções: acessando "File > New > Class", ou clicando com o lado direito do mouse em "src" e selecionando "New > Class", o que abrirá um *wizard* mas, como dito anteriormente, focaremos mais no código em si do que nas "mágicas" que os editores fazem para nós.

O nome desta classe será "Programa", porém não se preocupe ainda com as diversas opções que aparecem nesta janela. Há até um checkbox para o caso de querermos `public static void main(String[] args)`, o que não é o caso, pois por ora queremos praticar bastante e escrever um código básico.

Clicando-se em "Finish", teremos o programa simples que escrevemos antes:

```
public class Programa {  
  
    public static void main(String[] args){  
  
        System.out.println("ola mundo");  
    }  
}
```

**COPIAR CÓDIGO**

Conforme vamos digitando o código, o programa vai tentando completar, para nos ajudar. Na lateral esquerda, a bolinha vermelha com "x" indica erro de compilação em determinadas linhas.

O asterisco ( \* ) ao lado do nome do arquivo indica que ele não foi salvo!

Para executarmos o código, basta acessarmos "Run > Run As > Java Application", o que abre uma *view* para Console, que abre e executa, no caso, o `javaw.exe`, uma versão do Java que não abre no Prompt do MS-DOS, utilizado internamente pelo Eclipse para chamar o nosso programa, que é o que gostaríamos de fazer neste ambiente.

Por enquanto, não há tantas vantagens em relação ao Notepad, e não é à toa que a Microsoft, quando lançou o Visual Studio, foi atrás dos recursos apresentados pelo Eclipse, com o *ReSharper*, contratando o Erich Gamma, autor de *Design Patterns: Elements of Reusable Object-Oriented Software*, para trabalhar com o IDE deles. O Eclipse foi uma inspiração para muitos, e é considerado uma ferramenta incrível.

Mas onde se encontra o "Programa.class", o *bytecode* que a *virtual machine* entende?

O "Package Explorer", que é uma *view*, esconde arquivos e diretórios que julga não serem relevantes. E faz sentido, pois no momento estamos focados no programa Java. Acessando-se "Window > Show View > Navigator", ele irá mostrar o *File System*.

Clicando em "sintaxe-basica" para abri-lo, além do "src", existem outros diretórios e arquivos, dentre os quais "Programa.class" na pasta "bin". ".classpath" e ".project" são arquivos de configuração utilizados pelo Eclipse para obter informações sobre seu projeto. Eles não devem ser editados diretamente e, clicando-se na aba "Source" na parte inferior da interface, você verá que trata-se de um `.xml`.

Não precisaremos nos preocupar com estes arquivos, pois é muito raro termos que mexer neles. Há muito tempo, só existia esta *view*, o "Navigator" (similar ao Windows Explorer). O "Package Explorer" surgiu para ajudar quem trabalha com Java, e o "Navigator", para quem trabalha com tudo, de forma geral. Na maioria das vezes, usaremos o "Package Explorer".

Se clicarmos em "src > Programa.java" com o lado direito do mouse, e em seguida em "Properties", veremos a localização exata do arquivo, possível de ser confirmado por meio do Prompt de Comando.

A partir de agora deixaremos de utilizar o prompt, pois faremos tudo no Eclipse. De qualquer forma, é importante lembrarmos que o Java pode ser usado por linhas de comando, se você preferir, o que acaba sendo até necessário em alguns casos.

01

## Tipo inteiro: int

### Transcrição

Trabalharemos com sintaxes de variáveis e controles de fluxo - laços e condicionais - pela criação de um novo projeto acessando-se "New > Java Project". Poderíamos fazer tudo isto no mesmo arquivo, mas o intuito aqui é de treinar a codar e perder o medo das janelas e suas diversas opções.

Criaremos o "sintaxe-variaveis-e-fluxo", os dois tópicos que começaremos a ver. O novo projeto contendo o diretório "src" estará visível na view de "Package Explorer". No prompt, há um diretório "bin" escondido, pois o programa não quer mostrar o `.class`, e sim o código fonte Java. Reparem que no momento estou usando Mac, o que pouco importa, já que o Eclipse funciona da mesma maneira em todos os sistemas operacionais.

Criaremos nossa classe para começar a trabalhar com variáveis. Clicaremos com o lado direito do mouse em "src" e depois em "New > Class", e a classe se chamará "TestaVariaveis". No Java, um *statement* (ou instrução) não funciona fora dos métodos, portanto precisaremos do ponto inicial, do `public static void main(String[] args)`, após o qual salvaremos:

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

```
}
```

[COPIAR CÓDIGO](#)

Poderíamos rodar a aplicação assim como está, mas não aconteceria nada. Então, digitaremos:

```
public class TestaVariaveis {  
  
    public static void main(String[] args) {  
        System.out.println("ola novo teste");  
    }  
}
```

[COPIAR CÓDIGO](#)

Salvaremos novamente e rodaremos a aplicação indo à "Run > Run As > Java Application", ou clicando com o lado direito do mouse na classe com `main`, e em "Run As > Java Application". Também há o atalho "Ctrl + S". O Console mostrará o print, e com isto repetimos o mesmo teste do "ola mundo" feito anteriormente.

As palavras que aparecem em roxo no editor são as palavras chave, reservadas, e deverão estar sempre em caixa baixa. Agora, para criarmos uma variável denominada `idade`, que armazenará nossas idades, digitaremos:

```
public class TestaVariaveis {  
  
    public static void main(String[] args) {  
        System.out.println("ola novo teste");  
  
        idade = 37;  
    }  
}
```

[COPIAR CÓDIGO](#)

No Java, como o Eclipse já está dando a entender sublinhando `idade` com vermelho, não compila isto, pois trata-se de uma linguagem **estaticamente ou fortemente tipada**, ou seja, que necessita da declaração de todas as variáveis e tipos a serem utilizados. Passando o mouse sobre a palavra sublinhada, lê-se a mensagem de erro "*idade cannot be resolved to a variable*".

Significa que " `idade` não pode ser entendida como uma variável", pois não foi declarada. O Eclipse inclusive dará algumas opções de "rápido conserto", ou *quick fix*, para a criação local da variável, ou remoção da linha, por exemplo. `idade = 37` é uma **atribuição**, em que `37` se encontra dentro de `idade`.

Precisaremos declará-la informando que ela é do tipo numérico e que guarda um valor inteiro, sem decimais ou pontos flutuantes. `int` vem de *Integer*:

```
public class TestaVariaveis {  
  
    public static void main(String[] args) {  
        System.out.println("ola novo teste");  
  
        int idade;  
        idade = 37;  
    }  
}
```

[COPIAR CÓDIGO](#)

Salvaremos e rodaremos este código. Clicando-se na setinha ao lado do ícone verde que indica *play* na barra de ferramentas superior, vê-se os últimos programas que foram rodados no programa. E clicando no ícone verde, roda-se o último deles.

O valor foi guardado, mas parece que nada aconteceu de fato. Além de atribuirmos uma variável, pode-se usar o valor, mostrando-o na tela. Para isto, utilizaremos o `System.out.println` de novo, desta vez sem as aspas, pois queremos a *evaluation*, o resultado daquela expressão, e não uma cadeia de caracteres, uma *string*:

```
public class TestaVariaveis {  
  
    public static void main(String[] args) {  
        System.out.println("ola novo teste");  
  
        int idade;  
        idade = 37;  
  
        System.out.println(idade);  
    }  
}
```

[COPIAR CÓDIGO](#)

Inclusive, é possível ver que todas as menções à variável `idade` ficam em *highlight*, destacadas para mostrar que tratam-se da mesma variável. Vamos rodar o código acima para imprimirmos o valor de `idade` ! No "Console", obteremos:

ola novo teste

37

[COPIAR CÓDIGO](#)

Poderemos trabalhar com os operadores aritméticos junto a estas variáveis, também:

```
idade = 30 + 10;  
idade = 7 * 5 + 2;
```

[COPIAR CÓDIGO](#)

Como na maioria das linguagens, no Java também há precedência, então as operações matemáticas seguem uma determinada ordem de prioridade, mas podemos usar parênteses, desta forma:

```
idade = (7 * 5) + 2;
```

[COPIAR CÓDIGO](#)

E assim por diante. Imprimiremos a idade três vezes:

```
int idade;  
idade = 37;  
  
System.out.println(idade);  
  
idade = 30 + 10;  
  
System.out.println(idade);  
  
idade = (7 * 5) + 2;  
  
System.out.println(idade);
```

[COPIAR CÓDIGO](#)

E obteremos o resultado esperado, na aba "Console":

```
37  
40  
37
```

[COPIAR CÓDIGO](#)

No código, usamos algumas convenções: ao criarmos a classe `TestaVariaveis`, cuja funcionalidade ainda desconhecemos, usamos a primeira letra em maiúscula e, ao acrescentarmos a segunda palavra, não utilizamos *underscore* ou algo do tipo, e sim a primeira letra em caixa alta de novo. Isto se chama ***Camel Case***, e aparece com frequência no Java e em muitas outras linguagens - é uma **convenção de código**, e seu uso não é obrigatório.

Da mesma forma, a variável iniciando-se com "i" minúsculo é o padrão, bem como não há o costume de se abreviar palavras. No Java, vocês verão nomes gigantescos de variáveis! É legal nos atentarmos a estas práticas para começarmos a nos acostumar com estes hábitos essenciais para quando formos trabalhar com grandes equipes.

Para mostrarmos uma frase antes de um número, basta imprimirmos uma *string*, como "a idade é", juntamente com a variável `idade`, assim:

```
System.out.println("a idade é " + idade);
```

[COPIAR CÓDIGO](#)

O operador `+`, na maioria das vezes, tem a função de somar variáveis de tipo numérico, sendo a única exceção estes casos em que acompanham *strings*, com os números sendo convertidos em letras e tudo sendo concatenado. Este operador, portanto, também serve para concatenar algo com uma palavra ou frase (uma *string*).

Salvando e rodando a aplicação, teremos:

A idade é 37

[COPIAR CÓDIGO](#)

Pode-se acrescentar mais *strings* após a variável usando-se o operador.

Há outra versão do `System.out.println()`, o `System.out.print()`, sem o `ln`, isto é, sem o *line*, que pula a linha, que poderá ser utilizado de acordo com sua preferência.

04

## Tipo flutuante: double

### Transcrição

Também queremos trabalhar com outros tipos de variáveis, pois se tentarmos colocar no lugar da idade, em `idade = 37;`, um valor como `37.5`, a compilação não irá ocorrer. O erro que se lê ao passarmos o mouse em cima, é "*Type mismatch: cannot convert from double to int*", isto é, a conversão não é possível. Lembrando que no Java nunca usaremos a vírgula para separar o decimal no código fonte.

Vamos criar uma nova classe para testar os números com **ponto flutuante**, clicando em "(default package)" com o lado direito do mouse e em "New > Class", nomeando-a de "TestaPontoFlutuante". Teremos, então:

```
public class TestaPontoFlutuante {  
  
    public static void main(String[] args) {  
        double salario;  
        salario = 1250.70;  
        System.out.println("meu salário é " + salario);  
    }  
}
```

COPIAR CÓDIGO

Há dois tipos de variáveis em que cabem o tal de ponto flutuante, sendo que a mais utilizada é o `double`, como visto acima. Salvaremos o código e o rodaremos! Na aba "Console", obteremos o resultado:

meu salário é 1250.7

[COPIAR CÓDIGO](#)

Dica: as *views* podem ser customizadas de acordo com sua necessidade, sendo possível fechar aquelas que não estão em uso, por exemplo.

O `0` (zero) referente aos centavos não apareceram porque é assim que a variável `double` é convertida para se juntar à `string`. Poderemos formatar para aparecerem duas, três casas decimais, ou apenas uma, por meio dos *formatters* do Java, inclusos na biblioteca. Não veremos isto neste curso, porém há diversos recursos disponíveis, como o `printf`, da linguagem C, para colocarmos porcentagens e afins.

No `double` cabem variáveis do tipo inteiro, isto é, poderemos fazer o caminho inverso, indicando que temos uma variável que guarda `idade`, com número `37`. Reparem que este valor não possui decimal. Não tem problema, um número de tipo inteiro cabe em um tipo `double`. O inverso, um decimal em um `int`, é que não é compatível.

O Java tem regras um tanto rígidas, portanto não aceitará `3.0` como `int`, já que não aceita pontos flutuantes. Ele é uma linguagem com peculiaridades que algumas pessoas podem estranhar, sendo vantajoso para se trabalhar em equipe pois reforça uma padronização, e as pessoas trabalham de formas parecidas.

Para enxergarmos a forma como o `double` funciona, podemos fazer uma conta de divisão, por exemplo:

```
double divisao = 3.14 / 2;  
System.out.println(divisao);
```

[COPIAR CÓDIGO](#)

Rodando o código acima, teremos:

1.57

[COPIAR CÓDIGO](#)

Parece básico para quem já conhece linguagem estaticamente tipadas, mas mesmo nessa parte mais básica da linguagem, nos aprofundaremos mais, para sentirmos algumas das características do Java.

O que aconteceria no caso de digitarmos `int outraDivisao = 5 / 2;`? Alguns podem pensar que isso não é compilado, que dará erro, pois o resultado é `2.5`, o que não cabe em `int`.

No Java, entretanto, há uma regra: quando há uma divisão entre dois números inteiros, ele "forçará" um número inteiro como resultado. Se printarmos esta divisão, obteremos como resultado o valor `2`. Estranho, não?

E se quiséssemos que o resultado fosse `2.5` de fato, poderíamos tentar `double novaTentativa = 5 / 2;`, e pediríamos sua impressão, que traria `2.0`. Piorou! O Java irá ler apenas o lado que vem antes da atribuição na linha de código relativa ao `double`. Ou seja, primeiro, ele irá executar a divisão `5 / 2`, e depois ele tentará colocar o resultado em um `double`.

Na verdade, o que gostaríamos é que a conta tivesse sido feita partindo-se do `double` e, neste caso, bastaria que um dos valores da divisão fosse deste tipo, como em `5.0 / 2`. Desta forma, como trata-se de um `double` dividido por um `int`, a conta é feita levando-se em consideração o ponto flutuante.

Esta divisão, sim, trará `2.5` como resultado. Parece pegadinha, mas são características de linguagem que vão te deixar mais a par de como o Java funciona.

Fizemos um truque: estávamos sempre declarando a variável primeiro, e na linha seguinte, fazendo a atribuição. Depois, fizemos `double idade = 37;`. Ao declararmos variáveis, é muito comum inicializá-las, porque é estranho declararmos uma variável, digitarmos um monte de código e só depois inicializarmos um valor.

O comum é fazermos tudo na mesma linha. Desse modo, as linhas abaixo,

```
double salario;  
salario = 1250.70;
```

[COPIAR CÓDIGO](#)

pela proximidade, equivalem a escrevermos isto:

```
double salario = 1250.70;
```

[COPIAR CÓDIGO](#)

Declaramos a variável informando seu tipo, e a atualizamos, isto é, inicializamos ela, fazendo uma atribuição.

Vamos salvar o código!

07

## Conversões e outros tipos

### Transcrição

Haverá momentos em que queremos misturar os tipos de variáveis, como o `double` e o `int`. Vimos que um `int` cabe no `double`, mas o caminho inverso não funciona. Vamos, então, criar uma classe denominada "TestaConversao".

Incluiremos uma variável do tipo `salario` com os `1270.50`, que por algum motivo queremos que esteja em uma variável do tipo inteiro. E então guardaremos `salario` em `valor`:

```
public class TestaConversao {  
  
    public static void main(String[] args) {  
        double salario = 1270.50;  
        int valor = salario;  
    }  
}
```

[COPIAR CÓDIGO](#)

Já vimos que isto não funciona, pois o compilador do Java é rígido e não deixa que isto ocorra sem que afirmemos com total segurança de estarmos cientes de que perderemos o `.50`. Por conta disso, deixaremos as duas linhas comentadas, e mostraremos que o caminho inverso é possível:

```
public class TestaConversao {  
  
    public static void main(String[] args) {  
        // double salario = 1270.50;  
        // int valor = salario;  
  
        double valor = 3;  
    }  
}
```

[COPIAR CÓDIGO](#)

Ou seja, a conversão de um valor inteiro para um tipo `double` é possível, academicamente chamada de **promoção**, ou "ser promovido a um `double`", e acontece de maneira automática.

Para tentarmos fazer com que a parte do código comentada acima funcione, poderemos forçar a conversão, moldando um `double` para que ele se encaixe em um `int`.

É claro que não haverá encaixe perfeito, resultando em arestas que provavelmente serão perdidas. Faremos isso utilizando uma sintaxe comum a outras linguagens, o *casting*, para que o `double` seja transformado em um `int`.

```
public class TestaConversao {  
  
    public static void main(String[] args) {  
        double salario = 1270.50;  
        int valor = (int) salario;  
        System.out.println(valor);  
    }  
}
```

[COPIAR CÓDIGO](#)

Se printarmos `valor`, será mostrada apenas a parte inteira daquele número: 1270 . É isso que chamamos de *casting* que, nestas variáveis que guardam números, não é algo muito complexo.

Mais adiante, veremos o *casting* de variáveis que são referência, e têm a ver com orientação a objetos, se são compilados ou não, se darão *exceptions*; é um mundo à parte.

Basicamente, para os tipos chamados **primitivos**, as variáveis básicas que estamos vendo aqui e são `double` com "d" minúsculo, e na cor roxa, possuem funcionamento mais simples. O *casting* faz a conversão quando ela não é possível de forma automática.

Neste caso, sem o `(int)`, assim, entre parênteses, a compilação não ocorre, e a aplicação não rodará.

Como saberemos quais valores se encaixam em quê, e outros tipos numéricos?

No Java, o `int` e o `double` são os tipos mais usados, os outros aparecem de maneira muito esporádica. A nível de curiosidade, em `int` cabem 32bits com sinais, isto é, números positivos e negativos. Mais especificamente, cabem de  $2^{31}$  negativos, a  $2^{31}$  positivos menos 1 , por conta do 0 (zero), o que dá uma quantidade de cerca de 2 bilhões.

O `int` pode guardar até 2 bilhões e, passando dessa quantidade, ocorrerá um *overflow*. Caso se queira guardar um número maior ou menor que este, será preciso um número com 64bits , que no Java é o `long` , e guarda um número de até  $2^{63}$  menos 1 . É um número absurdo, que inclusive precisa de um `L` no fim, em caixa alta ou baixa, para indicar que estouramos os 2 bilhões!

```
long numeroGrande = 32432423523L;
```

[COPIAR CÓDIGO](#)

Por padrão, quando não é um `double`, um número no Java é considerado um `int`. O `L` indica "literal", um valor específico, como um `long`. Em contrapartida, há números menores: o `short`, que guarda um número de 16bits menos 1, e o `byte`, que é menor ainda, de até 2 elevado a 8, que dá 256 com 128 negativos, a 127 com 1 a menos:

```
short valorPequeno = 2131;  
byte b = 127;
```

[COPIAR CÓDIGO](#)

E se o número for maior do que 64bits, um número gigantesco? Daí, não serão usados tipos primitivos, ou estas variáveis. Podem ser objetos, e então usaremos bibliotecas.

Nesse caso, usaremos este exemplo:

```
double valor1 = 0.2;  
double valor2 = 0.1;  
double total = valor1+valor2;
```

[COPIAR CÓDIGO](#)

Esta operação deveria resultar em `0.3`, certo? Ao acrescentarmos `System.out.println(total);` e rodarmos o código, porém, obteremos `0.3000000000000004`. Que número maluco é esse?

Há várias questões matemáticas por trás dele. Se pesquisarmos o valor no Google, encontramos diversos resultados de pessoas buscando uma explicação. Existe até

site [0.3000000000000004.com/](http://0.3000000000000004.com/), com a explicação matemática para esse *floating point*, do porquê, em muitas linguagens, essa soma dar exatamente esse valor.

Não é à toa - como uma representação de decimal do inteiro é utilizada para se obter um ponto flutuante, fica complicado fazer uma operação aritmética deste tipo e guardar o resultado internamente. Por isto, o Java, como muitas outras linguagens, segue a especificação **IEEE 754**, de leitura complexa, que remete à Engenharia. De qualquer forma, é normal que este resultado apareça quando utilizamos o `double`.

Para lidarmos com dinheiro sem que apareçam centavos, por exemplo, usariámos o `BigDecimal`, de que falaremos mais para a frente. Por ora continuaremos com o `double` pois ainda estamos iniciando na linguagem, e queremos usar variáveis que são palavras chave do Java.

Os quatro tipos de tipo primitivo são: `int`, `long`, `byte` e `short`. Quanto aos tipos flutuantes, além do `double`, há o `float` e, se tentarmos definir a variável como recebendo `3.14`, ocorre o mesmo problema do `long`, mesmo se tratando de ponto flutuante.

Para o Java, `3.14` é um `double` com 64bits. É um valor que cabe em um tipo flutuante com 32bits? Não, e informações podem ser perdidas. Neste caso, usa-se o *casting*, o que seria estranho, ou se indica que este literal, o valor `3.14`, é um `float`, colocando-se "f" no fim:

```
float pontoFlutuante = 3.14f;
```

**COPIAR CÓDIGO**

Mais uma vez, o mais importante é o enfoque no `double` e no `int`, que aparecem com muito mais frequência. E no `long` em alguns casos, o qual será visto em alg

exercícios.

 10

## Para saber mais: Type Casting

Como foi visto nos vídeos, quando tentamos colocar um valor inteiro em uma variável do tipo double o Java não mostra erro. Quando tentamos, porém, colocar um double numa variável do tipo inteiro temos um erro de compilação.

Esta propriedade se dá porque o Java faz conversão implícita de um tipo menor para os tipos "maiores". De inteiro para double, por exemplo.

O contrário não é verdade por que existe perda de dados quando é feita a conversão. Acarretando em um "type mismatch" mostrando que esta instrução é de tipos incompatíveis.

Para fazer uma conversão onde pode haver perda de informações é necessário fazer um type casting. Veja a instrução abaixo.

```
int idade = (int) 30.0;
```

COPIAR CÓDIGO

No caso acima, está explícito que será feito o cast de double para inteiro. Veja como funciona o cast implícito e explícito na tabela abaixo.

<b>DE/PARA</b>	<b>byte</b>	<b>short</b>	<b>char</b>	<b>int</b>	<b>long</b>	<b>float</b>	<b>double</b>
byte	---	<i>Impl.</i>	(char)	<i>Impl.</i>	<i>Impl.</i>	<i>Impl.</i>	<i>Impl.</i>
short	(byte)	---	(char)	<i>Impl.</i>	<i>Impl.</i>	<i>Impl.</i>	<i>Impl.</i>
char	(byte)	(short)	---	<i>Impl.</i>	<i>Impl.</i>	<i>Impl.</i>	<i>Impl.</i>
int	(byte)	(short)	(char)	---	<i>Impl.</i>	<i>Impl.</i>	<i>Impl.</i>
long	(byte)	(short)	(char)	(int)	---	<i>Impl.</i>	<i>Impl.</i>
float	(byte)	(short)	(char)	(int)	(long)	---	<i>Impl.</i>
double	(byte)	(short)	(char)	(int)	(long)	(float)	---

Para comparar cada tipo primitivo de forma mais clara, a tabela abaixo mostra qual o tamanho de cada um.

<b>TIPO</b>	<b>TAMANHO</b>
boolean	1 bit
byte	1 byte
short	2 bytes
char	2 bytes
int	4 bytes
float	4 bytes
long	8 bytes
double	8 bytes

01

## Char e String

### Transcrição

A seguir, trabalharemos com caracteres e palavras! Criaremos uma nova classe mais uma vez, a "TestaCaracteres". Existe uma variável primitiva básica do Java que trabalha com *chars*, isto é, caracteres, cuja peculiaridade é guardar um único caractere de 16bits .

Usaremos as aspas simples para guardar a letra `a` , por exemplo:

```
public class TestaCaracteres {  
  
    public static void main(String[] args) {  
        char letra = 'a';  
        System.out.println(letra);  
    }  
}
```

[COPIAR CÓDIGO](#)

Ao salvarmos e rodarmos este código, lê-se `a` no Console, nada muito especial.

Quando trabalhamos com `char`s, estamos realmente "presos" a um único caractere. Se substituirmos `a` do código acima por `ab` , o código não compilará, e o mesmo ocorrerá se utilizarmos aspas duplas em vez das simples. O `char` guarda em si um único código, um número da tabela de Unicode, como a ASCII, porém muito maior e sem limite definido.

letra , portanto, é um número e, se observarmos bem, o char guarda em seu valor um número, mas é uma variável do tipo numérico equivalente àquele short , mas ele contém apenas valores positivos, possuindo mais detalhes. No momento, é interessante sabermos que ele é um número que é convertido em uma letra, como no trecho a seguir:

```
char valor = 66;  
System.out.println(valor);
```

[COPIAR CÓDIGO](#)

A partir do qual obteremos:

B

[COPIAR CÓDIGO](#)

Isto ocorre pois na tabela Unicode o 65 corresponde à letra a , portanto 66 refere-se a b . Testando-se o código abaixo,

```
valor = valor + 1;  
System.out.println(valor);
```

[COPIAR CÓDIGO](#)

há um erro de compilação em valor + 1 , por conta da regra do Java quando se trabalha com dois tipos distintos em uma mesma operação, de dar o resultado no maior deles. Neste caso, o valor é do tipo char , e 1 é um int , que é maior. O resultado desta operação, portanto, será dado em int . No entanto, um inteiro cabe em um char ? Não! Porém, novamente, o inverso é possível.

Se queremos que isto seja válido, devemos informar que a resposta disso passará pelo casting, moldando-se para o char :

```
valor = (char) (valor + 1);
System.out.println(valor);
```

[COPIAR CÓDIGO](#)

Salvando e rodando o código, receberemos a letra `c`. O `char` é interessante, mas não é tão usado no dia a dia, como no caso de `String`, com `S` em maiúsculo. Ela não é palavra chave do Java, não guarda valor, é um tipo referência. As diferenças ficarão mais claras quando formos entender melhor sobre orientação a objetos.

Atenção: o funcionamento básico de uma `String` exige aspas duplas, e não simples, as quais podem inclusive ficar vazias (`""`). Em `char`, por outro lado, não é possível deixar as aspas simples sem nada dentro (`' '`) - um espaço seria algo, e compilaria. Um `char` vazio, não.

```
String palavra = "alura cursos online de tecnologia";
System.out.println(palavra);
```

[COPIAR CÓDIGO](#)

Salvando e rodando o código, teremos a impressão `alura cursos online de tecnologia`, como esperado. E é possível utilizarmos o operador de soma (`+`) para concatenar `String`s, criando uma nova, como no exemplo abaixo:

```
palavra = palavra + 2020;
System.out.println(palavra);
```

[COPIAR CÓDIGO](#)

Isto nos retornará `alura cursos online de tecnologia2020`. A `String`, então, não se comporta como um `int` ou um `char`, mas aparecerá recorrentemente. Em breve

veremos que ela faz referência a um objeto e possui vários métodos. Ainda precisaremos aprender o básico e aprofundarmos nossos conhecimentos com calma!

04

## Variáveis guardam valores

### Transcrição

Um último detalhe muito interessante sobre estas variáveis do tipo primitivo - todas aquelas que vimos exceto a `String` - é seu funcionamento interno. O que são guardadas na memória delas?

Vamos criar mais uma classe, o `TestaValores`. E para não ficarmos digitando `public static void main(String[] args) {}` à mão o tempo todo, aprenderemos um atalho. Digitaremos "main" e apertaremos "Ctrl + barra de espaço" que, assim como em outros editores, tem a ver com o *autocomplete*. No Eclipse, também envolve *templates*.

Por meio deste atalho, aparecerão algumas opções, apertaremos a tecla "Enter", e o código aparece pronto no editor de texto. Isso passará a ser frequente para vocês.

Para entendermos como é guardado o valor de uma variável no Java, a **passagem por valor**, vamos fazer um desafio:

```
public class TestaValores {  
  
    public static void main(String[] args) {  
        int primeiro = 5;  
        int segundo = 7;  
  
        System.out.println(segundo);
```

COPIAR CÓDIGO

Ao rodarmos o código, obteremos 7.

```
public class TestaValores {  
  
    public static void main(String[] args) {  
        int primeiro = 5;  
        int segundo = 7;  
        segundo = primeiro;  
  
        System.out.println(segundo);  
    }  
}
```

[COPIAR CÓDIGO](#)

Salvando e rodando este código, obteremos 5!

```
int primeiro = 5;  
int segundo = 7;  
segundo = primeiro;  
primeiro = 10;  
  
// quanto vale o segundo?  
  
System.out.println(segundo);
```

[COPIAR CÓDIGO](#)

No segundo, tínhamos guardado o primeiro, mas agora primeiro vale 10. Quanto vale segundo?

As linguagens de programação trabalham de formas diferentes dependendo do uso de um símbolo específico, ou da existência de alguma referência, e por aí vai. Estas variáveis do tipo primitivo são trabalhadas com o valor do conteúdo, da variável,

então, quando copiamos 5 para dentro de segundo , e depois copiamos 10 para primeiro , a linha segundo = primeiro; não diz nada.

Quando se faz uma atribuição no Java, não se diz que uma variável **sempre** segue a outra, e sim que estamos **copiando e colando valores**. Deste modo, `primeiro = 10;` não surtirá efeito para `segundo` . Confirmaremos isto rodando a aplicação, pois continuaremos recebendo 5 .

Isso significa que a variável guarda um valor, e não uma referência, e este exemplo dará base para as entendermos melhor.

Estamos prontos para o próximo passo, que consiste em finalmente começarmos com controle de fluxos, com `if` , `while` e `for` , para estruturarmos nossos primeiros programas! E então veremos a orientação a objetos (O.O.) de maneira contra procedural. Vamos lá?

01

## Testes com IF

### Transcrição

Passaremos pelo nosso primeiro controle de fluxo, e testaremos a condicional `if`! Para isso, criaremos uma classe denominada "TestaCondicional", com uma variável inteira `idade`, inicializada na mesma linha:

```
public class TestaCondicional {  
  
    public static void main(String[] args) {  
        System.out.println("testando condicionais");  
        int idade = 20;  
        if (idade >= 18) {  
            System.out.println("você tem mais de 18 anos");  
            System.out.println("seja bem vindo");  
        }  
    }  
}
```

[COPIAR CÓDIGO](#)

Dica: pode-se usar "Ctrl + barra de espaço" após digitarmos "sysout" e apertarmos "Enter" para autocompletar o `System.out.println();` também!

Feito isso, salvaremos, e com o lado direito do mouse acessaremos "Run As > Java Application". Obteremos o seguinte:

```
testando condicionais  
você tem mais de 18 anos  
seja bem vindo
```

[COPIAR CÓDIGO](#)

Entre `if` e os parênteses que vêm a seguir, não é obrigatório ter espaço, mesmo que geralmente se use. No Java, o espaço, as teclas "TAB" e "Enter" não possuem papel fundamental. No Eclipse, ao acessarmos "Source > Format", o código é formatado de maneira correta.

Até aqui, nenhuma grande novidade. Neste caso, há duas instruções no bloco do `if` ... Existe algo proveniente do C no Java, em que as chaves não são necessárias, quando se quer apenas uma instrução na condicional. Isto é, se a linha referente ao texto "seja bem vindo" não existisse, poderíamos remover as chaves, deixando assim:

```
public class TestaCondisional {  
  
    public static void main(String[] args) {  
        System.out.println("testando condicionais");  
        int idade = 20;  
        if (idade >= 18)  
            System.out.println("você tem mais de 18 anos");  
            // System.out.println("seja bem vindo");  
  
    }  
}
```

[COPIAR CÓDIGO](#)

O `System.out.println();` que não está comentado (não está com as duas barras antes) faz parte do caso em que o `if` é `true`, verdadeiro. Quando temos um `if` um `else` sem o uso das chaves, não é possível ter duas instruções, e sim apenas

uma. Por isso, a boa prática implica em usarmos as chaves independentemente da quantidade de instruções existentes.

Isso facilita enxergarmos quem faz parte do quê, deixando menos margem para dúvidas e erros, mas isso vai da preferência de quem programa.

Para o `else`, alteraremos `idade` para que se receba `16`, e digitaremos:

```
public class TestaCondicional {  
  
    public static void main(String[] args) {  
        System.out.println("testando condicionais");  
        int idade = 16;  
        if (idade >= 18) {  
            System.out.println("você tem mais de 18 anos");  
            System.out.println("seja bem vindo");  
        } else {  
            System.out.println("infelizmente você não pode entrar")  
        }  
    }  
}
```

[COPIAR CÓDIGO](#)

Com isso, veremos a impressão de `infelizmente você não pode entrar` no Console. Para o caso da pessoa estar acompanhada, ela poder entrar, então acrescentaremos `int quantidadePessoas = 3;`, e um `if` após `else`. Em seguida, incluiremos outro `else` para o caso da pessoa ter menos de `18` e estar desacompanhada:

```
public static void main(String[] args) {  
    System.out.println("testando condicionais");  
    int idade = 16;  
    int quantidadePessoas = 3;
```

```
if (idade >= 18) {  
    System.out.println("você tem mais de 18 anos");  
    System.out.println("seja bem vindo");  
} else {  
    if(quantidadePessoas >= 2) {  
        System.out.println("você não tem 18, mas " + "pode entrar");  
    } else {  
        System.out.println("infelizmente você não pode entrar")  
    }  
}  
}
```

[COPIAR CÓDIGO](#)



Quando o código começa a se estender demais pela tela, dificultando a visualização integral, pode-se apertar "Enter", o que, no Eclipse, faz com que as *strings* sejam separadas por aspas e + automaticamente.

Salvando e rodando o código, obteremos:

você não tem 18, mas pode entrar, pois está acompanhado

[COPIAR CÓDIGO](#)

Outra dica: com duplo clique em qualquer uma das *views*, ela é maximizada. Fazemos o mesmo para minimizá-la. Isto pode facilitar nosso trabalho!



04

## Boolean condicionais

### Transcrição

Vamos explorar um pouco mais o funcionamento do `if`, para o qual criaremos mais uma classe. É recomendado criá-las para termos um histórico do que está sendo montado, passo a passo. Em `TestaCondicional2`, teremos o código mais ou menos parecido com o que estávamos vendo até então:

```
public class TestaCondicional2 {  
    public static void main(String[] args) {  
        System.out.println("testando condicionais");  
        int idade = 16;  
        int quantidadePessoas = 3;  
  
        if (idade >= 18) {  
            System.out.println("você tem mais de 18 anos");  
            System.out.println("seja bem vindo");  
        } else {  
            if(quantidadePessoas >= 2) {  
                System.out.println("você não tem 18, mas " + "pode " + "entrar");  
            } else {  
                System.out.println("infelizmente você não pode entrar")  
            }  
        }  
    }  
}
```

[COPIAR CÓDIGO](#)

Porém, não é muito legal quando o código tem muitos `if` se `else` encadeados, algo academicamente denominado **complexidade ciclomática** ou **complexidade condicional**. Neste nosso exemplo, poderíamos juntar os casos em que a pessoa tem mais de 18 anos e está acompanhada em uma condicional única.

Para isso, utilizaremos o operador `ou`, `||` - no Java, não existe `or` `ou` `and` como palavras chave.

```
public class TestaCondisional2 {  
    public static void main(String[] args) {  
        System.out.println("testando condicionais");  
        int idade = 16;  
        int quantidadePessoas = 3;  
  
        if (idade >= 18 || quantidadePessoas >= 2) {  
            System.out.println("seja bem vindo");  
        } else {  
            System.out.println("infelizmente você não pode entrar")  
        }  
    }  
}
```

[COPIAR CÓDIGO](#)

Vamos salvar e rodar o código para ver o que acontece? Será impresso no Console:

```
testando condicionais  
seja bem vindo
```

[COPIAR CÓDIGO](#)

Para este operador, basta apenas uma das condições ser `true`. Há também o `e`, ou `&&`, para quando houver necessidade de se ter mais de 18 anos e estar

acompanhado, por exemplo. Isto é, se mantivermos `idade` como 16 e `quantidadePessoas = 1;`, obteremos infelizmente você não pode entrar.

Aprendemos sobre tipos de variáveis como o `int` e o `double`, para inteiros e pontos flutuantes, respectivamente, o `char` para quando se usa apenas um caractere, entre outros. Além deles, existe o `boolean`, palavra chave do Java que é um tipo de variável que só aceita `true` (verdadeiro) ou `false` (falso), e fazem parte das palavras reservadas do Java.

```
public class TestaCondicional2 {  
    public static void main(String[] args) {  
        System.out.println("testando condicionais");  
        int idade = 16;  
        boolean acompanhado = true;  
  
        if (idade >= 18 && acompanhado) {  
            System.out.println("seja bem vindo");  
        } else {  
            System.out.println("infelizmente você não pode entrar")  
        }  
    }  
}
```

[COPIAR CÓDIGO](#)



No Java, `=` atribui, enquanto `==` compara. Em `boolean`, no caso de `acompanhado == true`, o próprio `acompanhado` já é um valor booleano, portanto, `== true` não é necessário.

O que também aparece com certa frequência é, à direita do `boolean`, colocarmos uma **expressão booleana** como `idade >= 18 && acompanhado`. Sendo assim, poderíamos usar simplesmente `boolean acompanhado = quantidadePessoas >= 2;`, o

que fará com que se conclua se a pessoa está acompanhada ou não. Com a idade sendo 20 , se rodarmos o código, obteremos seja bem vindo .

Também é possível imprimirmos "valor de acompanhado" e concatená-lo com acompanhado , deixando o código final assim:

```
public class TestaCondicional2 {  
    public static void main(String[] args) {  
        System.out.println("testando condicionais");  
        int idade = 20;  
        int quantidadePessoas = 3;  
        boolean acompanhado = quantidadePessoas >= 2;  
  
        System.out.println("valor de acompanhado = " +acompanhado);  
  
        if (idade >= 18 && acompanhado) {  
            System.out.println("seja bem vindo");  
        } else {  
            System.out.println("infelizmente você não pode entrar")  
        }  
    }  
}
```

**COPIAR CÓDIGO**

Salvaremos e rodaremos mais uma vez, e imprimiremos o seguinte:

```
testando condicionais  
valor de acompanhado = true  
seja bem vindo
```

**COPIAR CÓDIGO**



08

## Escopo e inicialização de variáveis

### Transcrição

Seguindo com as condicionais, veremos os **escopos de variáveis**. Já sabemos que o `boolean` acompanhado passa a valer ao declararmos as variáveis. Se tentássemos usá-la antes, logo após `idade`, ocorreria erro de compilação, pois a declaração ainda não foi feita.

Vamos criar a classe `TestaEscopo`, em que colaremos o código de `TestaCondicional12` pois trabalharemos em cima dele. Comentaremos a linha com o `boolean` para entendermos melhor o `if`, não esquecendo da declaração da variável `acompanhado` antes.

Uma variável, a partir de sua declaração, passa a valer entre as chaves correspondentes, o que se denomina **escopo**. Sendo assim, tanto `acompanhado = true` quanto `acompanhado = false` são necessários, pois fazem parte de **escopos diferentes**, com a inicialização sendo feita antes, em `boolean acompanhado;`, como se vê abaixo:

```
public class TestaEscopo {  
    public static void main(String[] args) {  
        System.out.println("testando condicionais");  
  
        int idade = 20;  
        int quantidadePessoas = 3;
```

```
// boolean acompanhado = quantidadePessoas >= 2;

boolean acompanhado;

if (quantidadePessoas >= 2) {
    acompanhado = true;
} else {
    acompanhado = false;
}

System.out.println("valor de acompanhado = " + acompanhado)

if (idade >= 18 && acompanhado) {
    System.out.println("seja bem vindo");
} else {
    System.out.println("infelizmente você não pode entrar")
}
}
```

**COPIAR CÓDIGO**

Qual o valor *default* de um `boolean`?

No Java, essas variáveis do tipo local, como as que estamos vendo aqui, dentro de `main`, são temporárias e não possuem valor padrão, sendo necessária sua inicialização **antes** de sua impressão, acesso, em uma operação, e assim por diante.

O Eclipse "percorre" o caminho de seus `if`s e da árvore de possibilidades, e identifica a existência de uma situação em que determinada variável pode ou não ter sido inicializada.



 12

## Para saber mais: o comando switch

Vimos como fazer testes com o `if`, mas se precisarmos fazer vários testes? Um exemplo, temos uma variável `mes`, precisamos testar o seu número e imprimir o seu mês correspondente. Então, vamos fazer 12 `if`s?

Para esses casos, existe o comando `switch`, onde podemos colocar todas as opções ou rumos que o nosso programa pode tomar. Ele funciona da seguinte maneira:

```
switch (variavelASerTestada) {  
    case opção1:  
        // comando(s) caso a opção 1 tenha sido escolhida  
        break;  
    case opção2:  
        // comando(s) caso a opção 2 tenha sido escolhida  
        break;  
    case opção3:  
        // comando(s) caso a opção 3 tenha sido escolhida  
        break;  
    default:  
        // comando(s) caso nenhuma das opções anteriores tenha sido escolhida  
}
```

[COPIAR CÓDIGO](#)

O código que será executado, que no nosso caso será a impressão do nome do mês, será o código em que a condição for verdadeira:

```
public class TestaMes {  
  
    public static void main(String[] args) {  
  
        int mes = 10;  
  
        switch (mes) {  
            case 1:  
                System.out.println("O mês é Janeiro");  
                break;  
            case 2:  
                System.out.println("O mês é Fevereiro");  
                break;  
            case 3:  
                System.out.println("O mês é Março");  
                break;  
            case 4:  
                System.out.println("O mês é Abril");  
                break;  
            case 5:  
                System.out.println("O mês é Maio");  
                break;  
            case 6:  
                System.out.println("O mês é Junho");  
                break;  
            case 7:  
                System.out.println("O mês é Julho");  
                break;  
            case 8:  
                System.out.println("O mês é Agosto");  
                break;  
            case 9:  
                System.out.println("O mês é Setembro");  
                break;  
            case 10:  
                System.out.println("O mês é Outubro");  
                break;  
        }  
    }  
}
```

```
        System.out.println("O mês é Outubro");
        break;

    case 11:
        System.out.println("O mês é Novembro");
        break;

    case 12:
        System.out.println("O mês é Dezembro");
        break;

    default:
        System.out.println("Mês inválido");
        break;
    }
}

}
```

**COPIAR CÓDIGO**

O `break` irá interromper a execução do caso que o contém, para os outros não serem executados, e se nenhuma condição for aceita, o código do `default` é que será executado. Por exemplo:

```
public class TestaMes {

    public static void main(String[] args) {

        int mes = 13;

        switch (mes) {
            case 1:
                System.out.println("O mês é Janeiro");
                break;
            case 2:
                System.out.println("O mês é Fevereiro");
                break;
            case 3:
```

```
        System.out.println("O mês é Março");
        break;

    case 4:
        System.out.println("O mês é Abril");
        break;

    case 5:
        System.out.println("O mês é Maio");
        break;

    case 6:
        System.out.println("O mês é Junho");
        break;

    case 7:
        System.out.println("O mês é Julho");
        break;

    case 8:
        System.out.println("O mês é Agosto");
        break;

    case 9:
        System.out.println("O mês é Setembro");
        break;

    case 10:
        System.out.println("O mês é Outubro");
        break;

    case 11:
        System.out.println("O mês é Novembro");
        break;

    case 12:
        System.out.println("O mês é Dezembro");
        break;

    default:
        System.out.println("Mês inválido");
        break;
    }
}
}
```

COPIAR CÓDIGO

A impressão será **Mês inválido**. Então, o `switch` é uma solução para os `if`s encadeados.

01

## Laço com while

### Transcrição

Finalmente chegamos nos laços, a última estrutura de controle de fluxo, básica e primordial em todas as linguagens! Criaremos uma classe específica para aprendermos sobre a estrutura de laço de repetição, o `TestaWhile`.

O `while` é uma palavra chave e, dentro dos parênteses, obrigatoriamente recebe uma expressão booleana, assim como o `if`. Por isso, precisaremos incluir algo lá dentro, que nos devolva `true` ou `false`.

```
public class TestaWhile {  
  
    public static void main(String[] args) {  
        int contador = 0;  
        while(contador <= 10) {  
            System.out.println(contador);  
            contador = contador + 1;  
        }  
    }  
}
```

[COPIAR CÓDIGO](#)

Ao salvarmos e rodarmos o código, serão impressos os números de `0` a `10`, como gostaríamos!

O `while` é uma instrução muito simples - lembrando que é preciso sempre inicializar e declarar a variável a ser utilizada, neste caso, em `contador`. Para reforçarmos algo que já foi visto, o escopo, poderemos imprimir `contador` novamente após o `while`:

```
public class TestaWhile {  
  
    public static void main(String[] args) {  
        int contador = 0;  
        while(contador <= 10) {  
            System.out.println(contador);  
            contador = contador + 1;  
        }  
        System.out.println(contador);  
  
    }  
}
```

[COPIAR CÓDIGO](#)

A partir do qual se obtém a impressão de `0` a `11`!

É claro que cabem outras condições booleanas no lugar de `contador <= 10`. Não é muito comum utilizarmos o formato `contador = contador + 1;` quando operamos sobre a própria variável, uma vez que existe uma forma mais sucinta, herdada do C:

```
contador += 1;
```

[COPIAR CÓDIGO](#)

Não é que seja "igual a mais um"! Queremos somar `1` nele mesmo. É uma sintaxe estranha, mas indica exatamente o mesmo que `contador = contador + 1;`. Para o mesmo efeito, existe ainda o `++`:

```
contador++;
```

[COPIAR CÓDIGO](#)

Esta, na verdade, é a forma mais comum de se somar o valor de si mesmo mais uma vez, e usar `++contador;` (o pré-incremento) também traria o mesmo resultado. Há casos em que existem diferenças, mas por ora não nos preocuparemos com isso.

É muito mais importante entendermos o escopo, que a variável precisa ser inicializada antes de se fazer qualquer ação com ela, pois isso não acontece automaticamente em condições temporárias, e que o `while` é o sistema de laço mais simples de todos.

04

## Escopo nos laços

### Transcrição

Para vermos o laço de forma mais estruturada e desafiadora, faremos uma somatória com os números de 0 a 10 , criando a classe TestaSomatoria :

```
public class TestaSomatoria {  
  
    public static void main(String[] args) {  
        int contador = 0;  
        while(contador <= 10) {  
            int total = 0;  
            total = total + contador;  
  
            System.out.println(total);  
            contador++;  
        }  
    }  
}
```

[COPIAR CÓDIGO](#)

Vamos imprimir as somatórias parciais para ver o que está acontecendo?

Queremos que se mostre 0 , seguido de 1 , e então 2 , 3 , por causa de 1 + 2 , e então 6 , de 1 + 2 + 3 . No entanto, obteremos:

0  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10

**COPIAR CÓDIGO**

Ué! Não funcionou! Isto porque toda vez que se entra no `while`, é criada uma nova variável `total` por causa do escopo e, ao voltarmos ao próximo laço, quando ocorre a **iteração**, ele zera de novo, pois a velha `total` já deixou de existir.

Falta acertarmos o escopo declarando e inicializando a variável `total` após a linha que contém `contador`:

```
public class TestaSomatoria {  
  
    public static void main(String[] args) {  
        int contador = 0;  
        int total = 0;  
  
        while(contador <= 10) {  
  
            total = total + contador;  
  
            System.out.println(total);  
            contador++;  
        }  
    }  
}
```

```
}
```

[COPIAR CÓDIGO](#)

Se salvarmos e rodarmos novamente, desta vez veremos o seguinte no Console:

```
0  
1  
3  
6  
10  
15  
21  
28  
36  
45  
55
```

[COPIAR CÓDIGO](#)

Ou seja, as somatórias parciais, incluindo a última, 55 , que é o número desejado. Se quisermos apenas este resultado final, poderemos deixar o código assim:

```
public class TestaSomatoria {  
  
    public static void main(String[] args) {  
        int contador = 0;  
        int total = 0;  
  
        while(contador <= 10) {  
            total = total + contador;  
            contador++;  
        }  
        System.out.println(total);  
    }  
}
```

```
}
```

[COPIAR CÓDIGO](#)

Salvando e rodando o código novamente, obtém-se a impressão de 55 .

É possível deixar este código mais enxuto, porém focaremos em `total = total + contador;`, que já vimos que pode ser escrito assim: `total += contador;`, o qual traz exatamente o mesmo resultado.

06

## Laço com for

### Transcrição

O `for` tem a sintaxe um pouco mais estranha. O `while` é uma estrutura de laço, e o `for` realiza a mesma tarefa, porém possui algumas vantagens em relação à legibilidade, mesmo que o resultado final - o *bytecode* - seja o mesmo. Criaremos `TestaFor`, em que incluiremos algo equivalente ao laço feito anteriormente, que conta de `0` a `10` imprimindo todos os números.

Diferentemente do `while`, não é preciso declararmos `contador` fora dele, pois o `for`, palavra chave do Java, tem uma sintaxe muito diferente. Até então, utilizamos apenas ponto e vírgula no fim dos *statements*, isto é, das linhas. Neste caso, usaremos o ponto e vírgula **dentro dos parênteses** (isto também herança do C).

Dentro dos parênteses, então, serão criados três "espaços" intercalados por ponto e vírgula, e então abriremos e fecharemos as chaves normalmente. O primeiro espaço é opcional e costuma ter a declaração e inicialização da variável, sendo executado **apenas uma vez**.

O segundo espaço é executado **todas as vezes** e contém a condição booleana para saber se ele deve ou não entrar no laço, ou seja, executar a próxima iteração. No nosso caso, queremos saber se `contador` é menor ou igual a `10`, como no `while`.

O terceiro espaço geralmente é ocupado por aquilo a ser executado ao fim de cada iteração, o que acaba sendo um tanto estranho para quem não está bem ambientado com isto. O código ficará desta maneira:

```
public class TestaFor {  
  
    public static void main(String[] args) {  
        for(int contador = 0; contador <= 10; contador++) {  
            System.out.println(contador);  
        }  
    }  
}
```

[COPIAR CÓDIGO](#)

Salvaremos e rodaremos o código, e obteremos o esperado, como em `while` :

```
0  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10
```

[COPIAR CÓDIGO](#)

Diferentemente do `while`, apesar de `int contador = 0` valer no escopo do `for` inteiro em todas as iterações, ele não é zerado, sendo executado apenas uma vez, e por isto sua sintaxe não é muito intuitiva. Se quisermos imprimir o último valor que o `contador` estava lendo, não conseguiremos, por conta do escopo.

O `for` oferece a possibilidade de haver uma variável que participa de todas as iterações, que é o que precisamos, mas depois do `for`, ela deixa de valer.

Não é melhor usarmos o `while`, então? Depende. Muitas vezes queremos utilizar a variável temporariamente, somente dentro do laço, e é por isso que o `for` é mais atrativo, e se adequa melhor a este tipo de caso.

No entanto, `while` e `for` são intercambiáveis, e inclusive existe outro laço, denominado *do-while*, que não veremos neste curso, mas que também poderia ser utilizado.

09

## Laços encadeados

### Transcrição

Já vimos todos os comandos básicos da sintaxe. Vamos praticar o uso dos laços, com o `if`, para sedimentarmos este conhecimento adquirido no curso! Criaremos uma classe para testarmos **laços encadeados**, aninhados uns aos outros: `TestaLacos`, com um `main` para imprimirmos de `0` a `10` dez vezes em linhas distintas, com a tabuada de cada número.

Usaremos o `int multiplicador`, começando pela tabuada do `1`, indo à do `10`. Dentro deste laço, queremos fazer outro, com valor diverso, como em um `contador`, também começando do `0` e indo a `10`.

```
public class TestaLacos {  
  
    public static void main(String[] args) {  
        for(int multiplicador = 1; multiplicador <= 10; multiplicador++) {  
            for(int contador = 0; contador <= 10; contador++) {  
                System.out.println(multiplicador * contador);  
            }  
        }  
    }  
}
```

[COPIAR CÓDIGO](#)

Se pedirmos para que seja impresso `multiplicador * contador`, obteremos algo gigantesco, como verificaremos salvando e rodando o código. Serão impressos os resultados contendo as tabuadas, mas queremos algo um pouco mais organizado. Para isso, em vez de utilizarmos o `System.out.println();`, usaremos `System.out.print();`, seguido de `System.out.print(" ")`, que nos trará os números todos alinhados horizontalmente.

Ainda não é isto que queremos! Queremos um "Enter" a cada tabuada, quer dizer, cada tabuada em uma linha. Vamos, então, incluir outro `System.out.println();` após o escopo do segundo `for`, assim:

```
public class TestaLacos {  
  
    public static void main(String[] args) {  
        for(int multiplicador = 1; multiplicador <= 10; multiplicador++) {  
            for(int contador = 0; contador <= 10; contador++) {  
                System.out.print(multiplicador * contador);  
                System.out.print(" ");  
            }  
            System.out.println();  
        }  
    }  
}
```

[COPIAR CÓDIGO](#)

Salvando e rodando o código acima, obteremos, como gostaríamos:

```
0 1 2 3 4 5 6 7 8 9 10  
0 2 4 6 8 10 12 14 16 18 20  
0 3 6 9 12 15 18 21 24 27 30  
0 4 8 12 16 20 24 28 32 36 40
```

```
0 5 10 15 20 25 30 35 40 45 50
0 6 12 18 24 30 36 42 48 54 60
0 7 14 21 28 35 42 49 56 63 70
0 8 16 24 32 40 48 56 64 72 80
0 9 18 27 36 45 54 63 72 81 90
0 10 20 30 40 50 60 70 80 90 100
```

**COPIAR CÓDIGO**

10

## Mais laços com break

### Transcrição

Feitas as tabuadas do vídeo anterior, vamos testar mais laços encadeados e ver como eles podem se comunicar? Criaremos para isto a classe `TestaLacos2`, para a qual copiaremos e colaremos o conteúdo de `TestaLacos`. Desta vez, substituiremos `multiplicador por linha`, enquanto `contador` passará a ser `coluna`. E não faremos mais multiplicações, e sim com que apareçam 10 linhas e 10 colunas. A partir do código abaixo, o que vocês acham que acontecerá?

```
public class TestaLacos2 {  
    public static void main(String[] args) {  
        for(int linha = 0; linha < 10; linha++) {  
            for(int coluna = 0; coluna < 10; coluna++) {  
                System.out.print("*");  
            }  
            System.out.println();  
        }  
    }  
}
```

COPIAR CÓDIGO

Na aba "Console", será mostrado algo não muito interessante:

```
*****  
*****
```

```
*****  
*****  
*****  
*****  
*****  
*****  
*****  
*****  
*****  
*****
```

[COPIAR CÓDIGO](#)

Uma grande quantidade de laços encadeados acaba não sendo esteticamente agradável e, às vezes, queremos que um laço se comunique com outro. Para que os asteriscos formem uma matriz triangular, por exemplo, acrescentaríamos ao código um `if` para quando `coluna` for maior que `linha`, fazendo com que o laço pare de ser executado e saia dali para ir à próxima linha do `for`, externo.

Bem como em outras linguagens, existe um comando no Java, a palavra chave `break`, que "corta" a execução do laço mais interno, isto é, mais próximo de onde ela mesma se encontra, resultando exatamente no efeito que buscamos:

```
public class TestaLacos2 {  
    public static void main(String[] args) {  
        for(int linha = 0; linha < 10; linha++) {  
            for(int coluna = 0; coluna < 10; coluna++) {  
                if(coluna > linha) {  
                    break;  
                }  
                System.out.print("*");  
            }  
            System.out.println();  
        }  
    }  
}
```

[COPIAR CÓDIGO](#)

Ao salvarmos e rodarmos o código, teremos:

```
*  
**  
***  
****  
*****  
*****  
*****  
*****  
*****  
*****  
*****  
*****
```

[COPIAR CÓDIGO](#)

No exemplo acima, poderíamos obter o mesmo efeito usando a condicional `if` sem as chaves, pois o `break` ocupa apenas uma linha, como seria possível também com `for` e `while`. No entanto, por boa prática, e visando à legibilidade e convenção, optaremos por usar as chaves sempre que possível.

E no segundo `for`, poderíamos ter substituído `coluna < 10` por `coluna <= linha`, modificando-se a instrução para não usarmos o `break`. Assim, o código completo ficaria da seguinte maneira:

```
public class TestaLacos2 {  
    public static void main(String[] args) {  
        for(int linha = 0; linha < 10; linha++) {  
            for(int coluna = 0; coluna <= linha; coluna++) {  
                System.out.print("*");  
            }  
        }  
    }  
}
```

```
        System.out.println();  
    }  
}  
}
```

[COPIAR CÓDIGO](#)

Há muitos exercícios a serem feitos e, mesmo que isso seja trivial para você, que já conhece outra linguagem de programação, ou esteja revendo comandos mais básicos, eles são interessantes para fixar erros de compilação. Senão, quando o conteúdo ficar mais complexo, as chances de se debater por aquilo que já deveria estar bem sedimentado serão maiores.

Portanto, não menospreze a sintaxe básica do Java! Se tiver dúvidas, use nosso fórum, com participação de instrutores e alunos, veja as dúvidas, busque se aprofundar cada vez mais.

Pratique bastante, pois no próximo curso encararemos os desafios de migrarmos da melhor forma de uma programação procedural, imperativa, para a tal da Orientação a Objetos. Muito obrigado!