

Linguagem de Programação Java

Profª Me. Cristiane Fidelix



(Simbologia clássica)

Conceitos

**Porque estudamos
Linguagem de Programação?**



RAZÕES PARA ESTUDAR LPs

- ⦿ Maior capacidade de desenvolver soluções computacionais para problemas.
- ⦿ Maior habilidade ao usar uma LP.
- ⦿ Maior capacidade para escolher LPs apropriadas.
- ⦿ Maior habilidade para aprender novas LPs .
- ⦿ Maior habilidade para projetar novas LPs.

Propriedades desejáveis em Lps:

- ◉ Legibilidade – Facilidade para se ler e entender um programa.
- ◉ Redigibilidade – Facilidade de redigir um programa. Possibilita ao programador se concentrar nos algoritmos e não na linguagem
- ◉ Confiabilidade – mecanismos fornecidos pelas LPs para incentivar a construção de programas confiáveis.
- ◉ Tratamento de Exceções
- ◉ Eficiência – programas mais ágeis
- ◉ Facilidade de Aprendizado – programas fáceis de aprender
- ◉ Modificabilidade – facilidade de alteração de programas
- ◉ Reusabilidade – facilidade de reutilização de código
- ◉ Portabilidade – utilização em diferentes plataformas

Como funciona as Lps?



Funcionamento das Lps:

- ◉ O computador é
 - *hardware que só entende operações muito básicas (zeros e uns, processadas logicamente);*
- ◉ Programa executável = coleção de instruções em linguagem de máquina;
- ◉ Criar programas em linguagem de máquina é extremamente difícil e improdutivo;
- ◉ Para facilitar
 - Usamos linguagens de programação menos complexa e utilizamos um programa que transforme uma linguagem em outra: um tradutor.

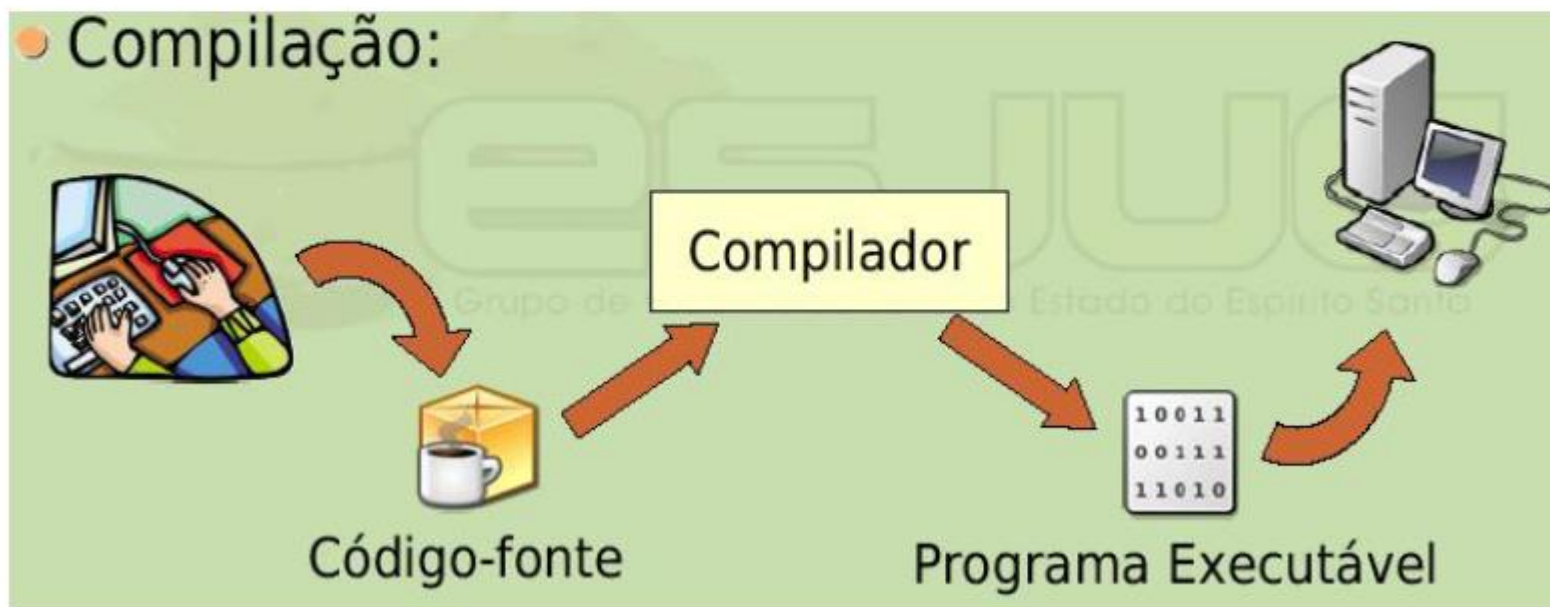
Papel das Lps no Desenvolvimento do Software

- ◉ Etapas do Desenvolvimento de Software
 - Planejamento.
 - Especificação de Requisitos.
 - Análise.
 - Projeto.
 - Implementação .
 - Validação (Testes).
 - Implantação.
 - Manutenção.

Tradução de programas:

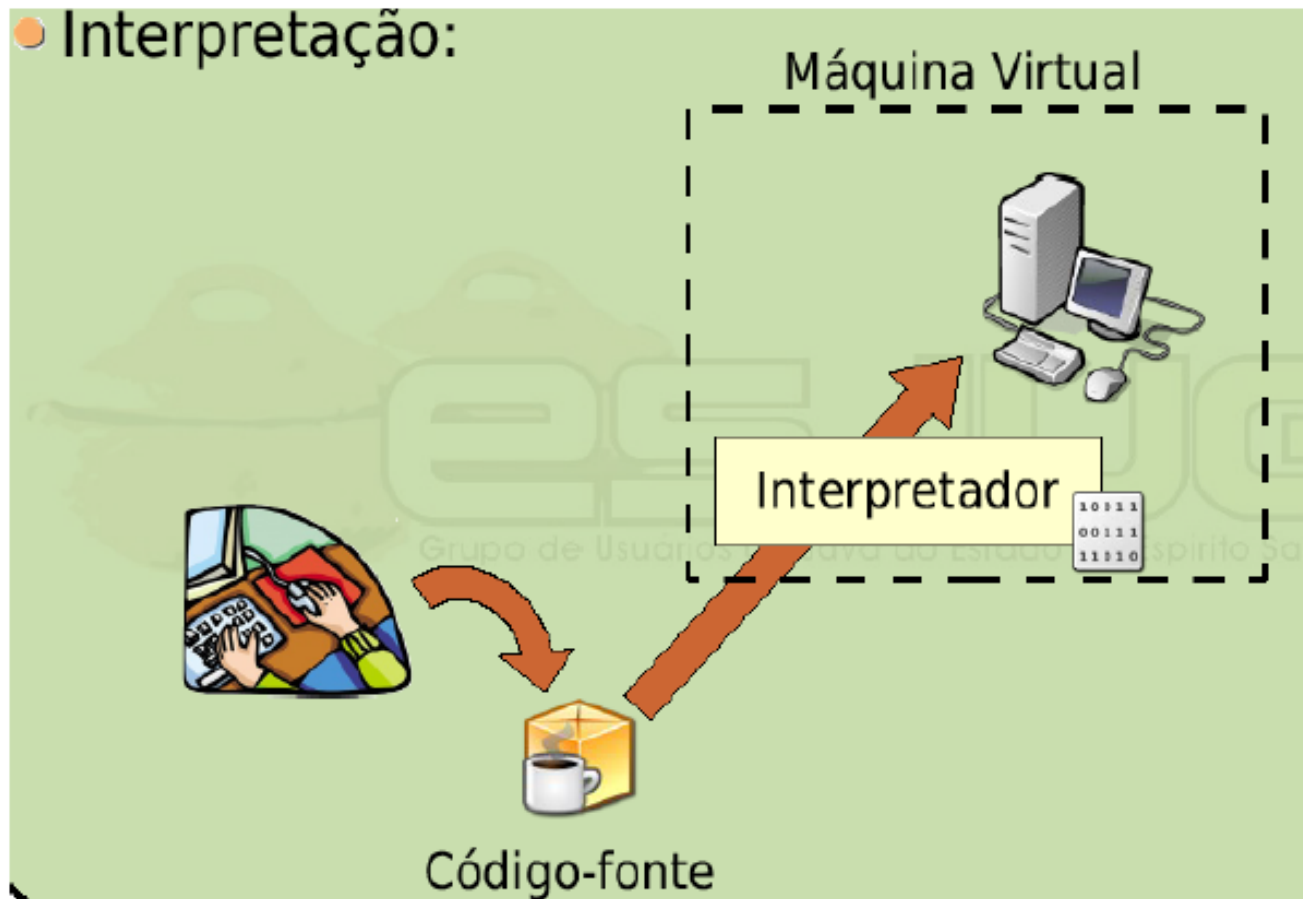
- Existe duas maneiras de se traduzir um programa:
 - compilação e interpretação.

● Compilação:



Tradução de programas:

● Interpretação:



Compilação X Interpretação:



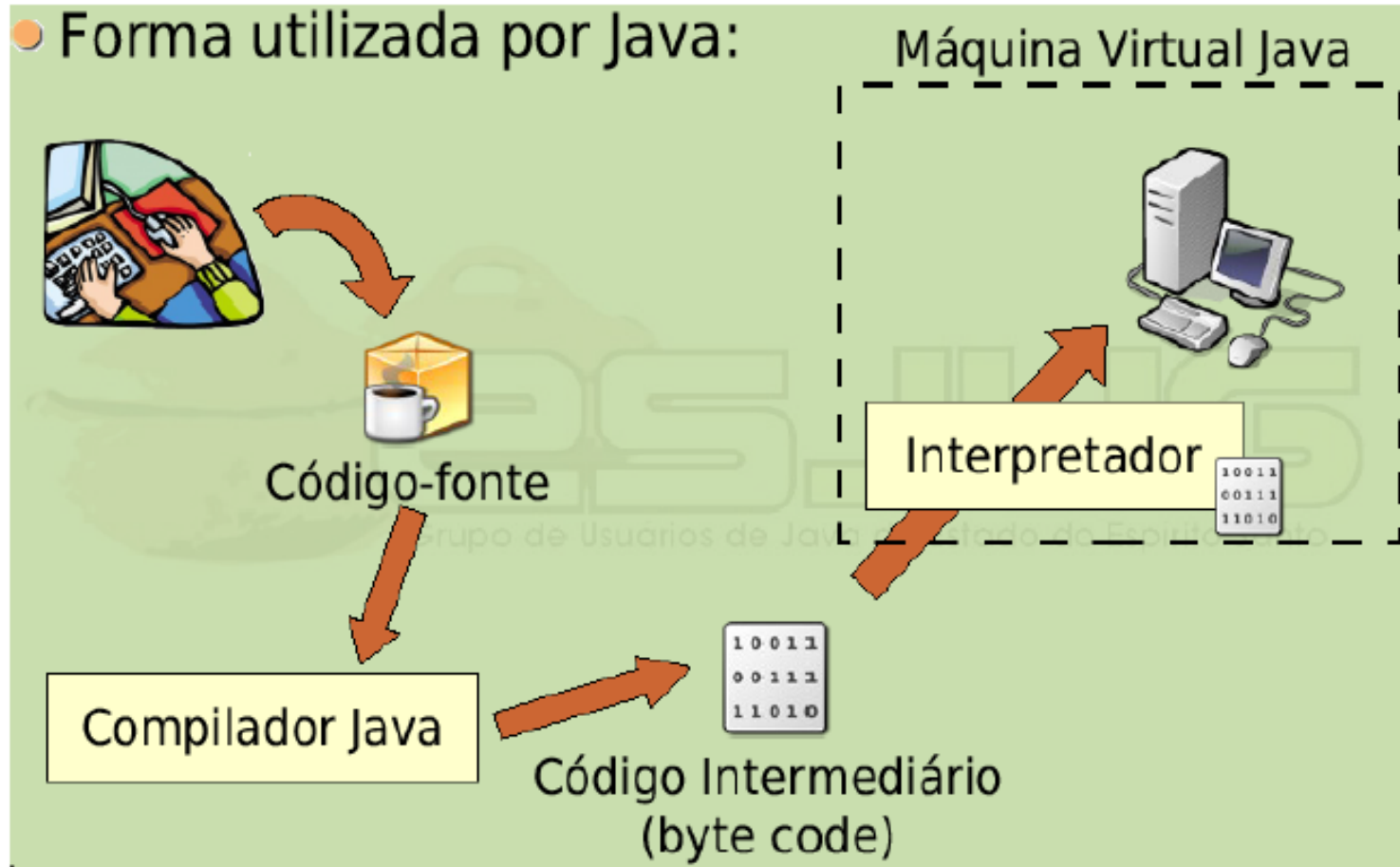
- ⦿ **Compilação:**
 - Execução mais rápida;
 - Somente o executável é carregado em memória.

- ⦿ **Interpretação:**
 - Portabilidade.

- ⦿ **Compilação + Interpretação = Híbrido**
 - Une as vantagens de ambos.

Tradução Híbrida:

- Forma utilizada por Java:



Implementação:

- Compilação
 - Maior eficiência
 - Problemas com portabilidade e depuração
 - Exemplo: C
- Interpretação Pura
 - Portabilidade e facilidade para depuração
 - Problemas com eficiência e maior consumo de memória
 - Raramente usada
- Híbrido
 - Une vantagens dos outros métodos
 - Exemplo: Java – o código intermediário é o *bytecode* e o seu interpretador é o JVM (*Java Virtual Machine*)

**Quais são os tipos de
Programação?**



Tipos de programação

- Programação estruturada;
- Programação modular;
- Programação orientada à eventos;
- Programação orientada à objetos.

O que é Paradigma?

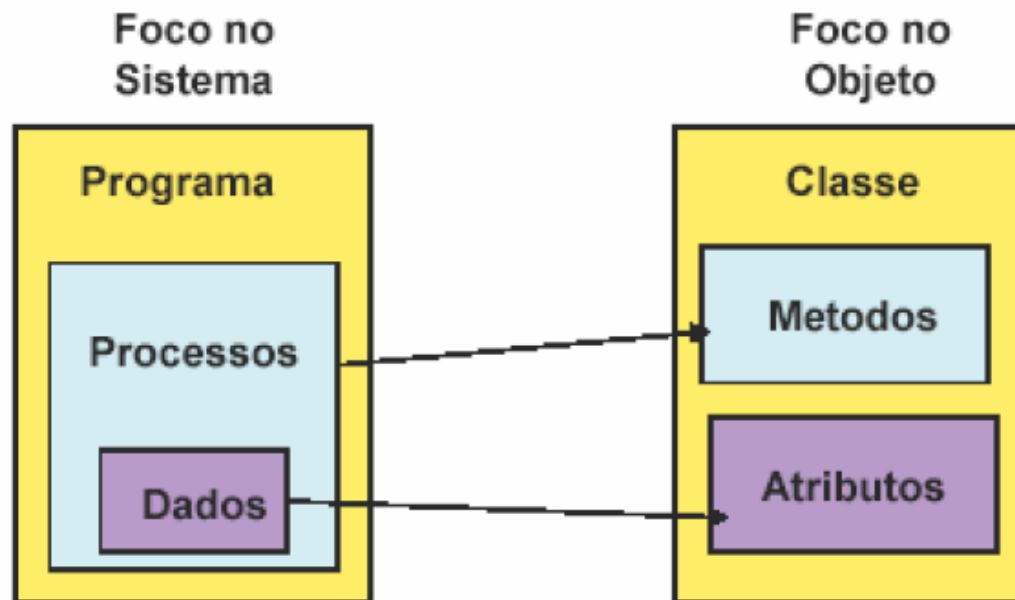
“Paradigma é um conjunto de regras que estabelecem fronteiras e descrevem como resolver os problemas dentro dessas fronteiras. Os paradigmas influenciam nossa percepção; ajudam-nos a organizar e a coordenar a maneira como olhamos para o mundo...”

Daniel C. Morris e Joel S. Brandon,
Reengenharia: reestruturando sua empresa,
São Paulo, Makron Books, 1994.

Metodologia Estruturada X

Orientada a Objetos

- Metodologia estruturada = o foco é na especificação e decomposição da funcionalidade do sistema
- Metodologia orientada a objetos = primeiro identificam-se os objetos contidos no domínio do sistema e depois os procedimentos relativos a ele.



Exemplos:

Jogo de Futebol

- Análise de um campo de futebol utilizando metodologia estruturada
 - Passe da bola;
 - Fazer gol;
 - Cobrar lateral;
 - Cobrar tiro de meta;
 - Driblar o jogador;
 - Cobrar falta;
 - Marcar penault;
 - Passar a bola.
- Análise de um campo de futebol utilizando metodologia orientada a objetos
 - Jogador;
 - Juiz;
 - Bola;
 - Campo.

Programação Estruturada X OO

| PROGRAMAÇÃO ORIENTADA A OBJETOS | PROGRAMAÇÃO ESTRUTURADA |
|---------------------------------|---------------------------------------|
| Métodos | Procedimentos e Funções |
| Instâncias de Variáveis | Variáveis |
| Mensagens | Chamadas a procedimentos e funções |
| Classes | Tipos de Dados definidos pelo usuário |
| Herança | - |
| Polimorfismo | - |

1. O que é Java

Java

Entender um pouco da história da plataforma Java é essencial para enxergar os motivos que a levaram ao sucesso.

Quais eram os seus maiores problemas quando programava na década de 1990?

- ponteiros?
- gerenciamento de memória?
- organização?
- falta de bibliotecas?
- ter de reescrever parte do código ao mudar de sistema operacional?
- custo financeiro de usar a tecnologia?

Histórico do Java

- ◉ Uma linguagem de programação;
- ◉ Um software distribuído pela Sun Microsystems;
- ◉ Uma ilha da Indonésia.
- ◉ Mantida por uma comunidade.



HORA do Vídeo JAVA !!!!

<http://youtube.com/watch?v=sTX0UEplF54>

JAVA É

- Uma especificação criada pela SUN, entretanto a linguagem Java é Mantida pelo Java Community Proccess (JCP) que reúne experts em Java, empresas e universidades que por meio de processos democráticos definem a evolução da linguagem.

⦿ **Por que Java?**

- Uma das linguagens OO mais usadas;

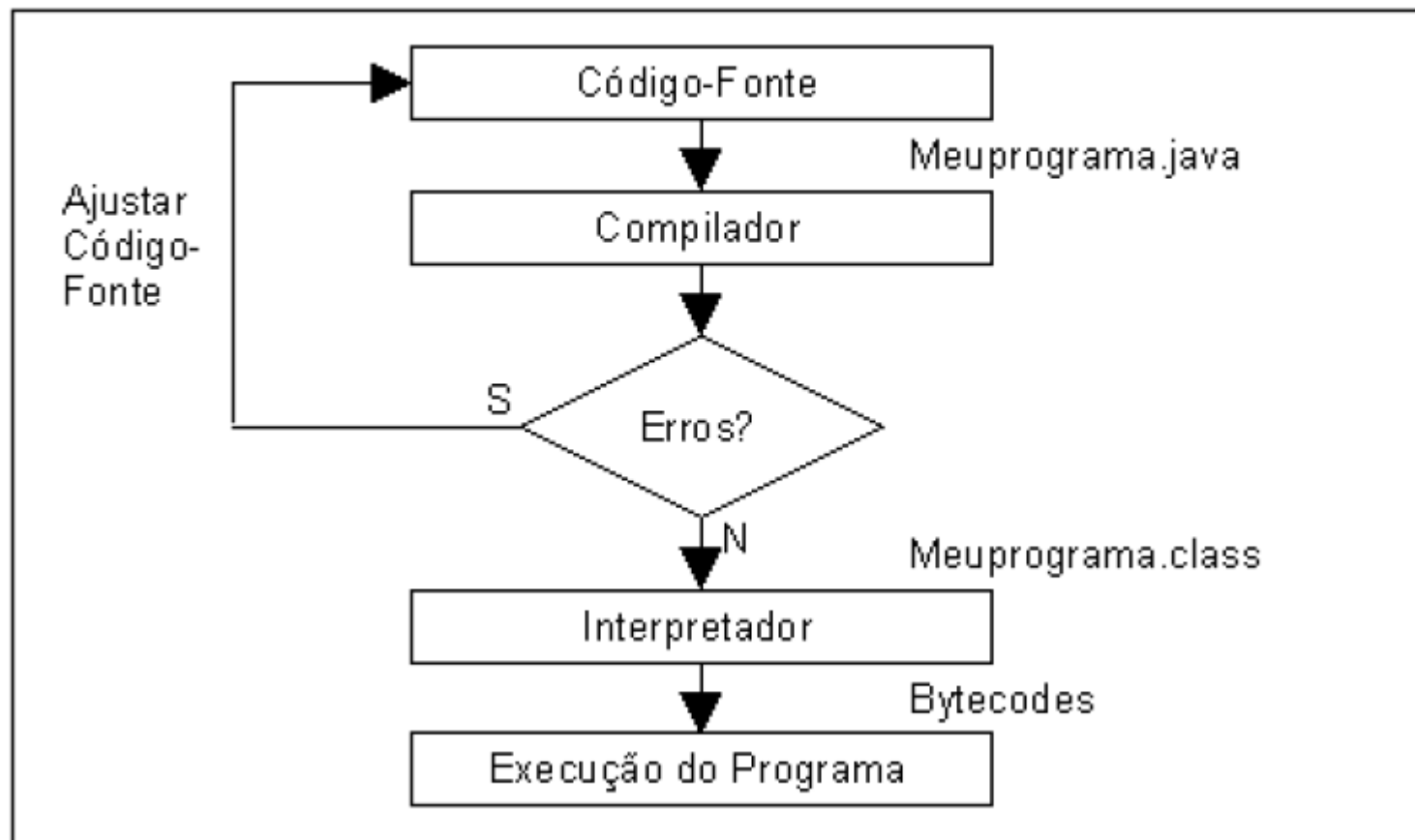
⦿ **Características de Java:**

- Simples, porém versátil, robusta e muito segura;
- Portável (independente de sistema operacional);
- Gratuita e com código disponível para consulta;
- Popular, rodeada por uma comunidade muito ativa;
- De alta aceitação e com suporte da indústria;
- Muitas ferramentas disponíveis;
- Muita documentação disponível.

CARACTERÍSTICAS DE JAVA

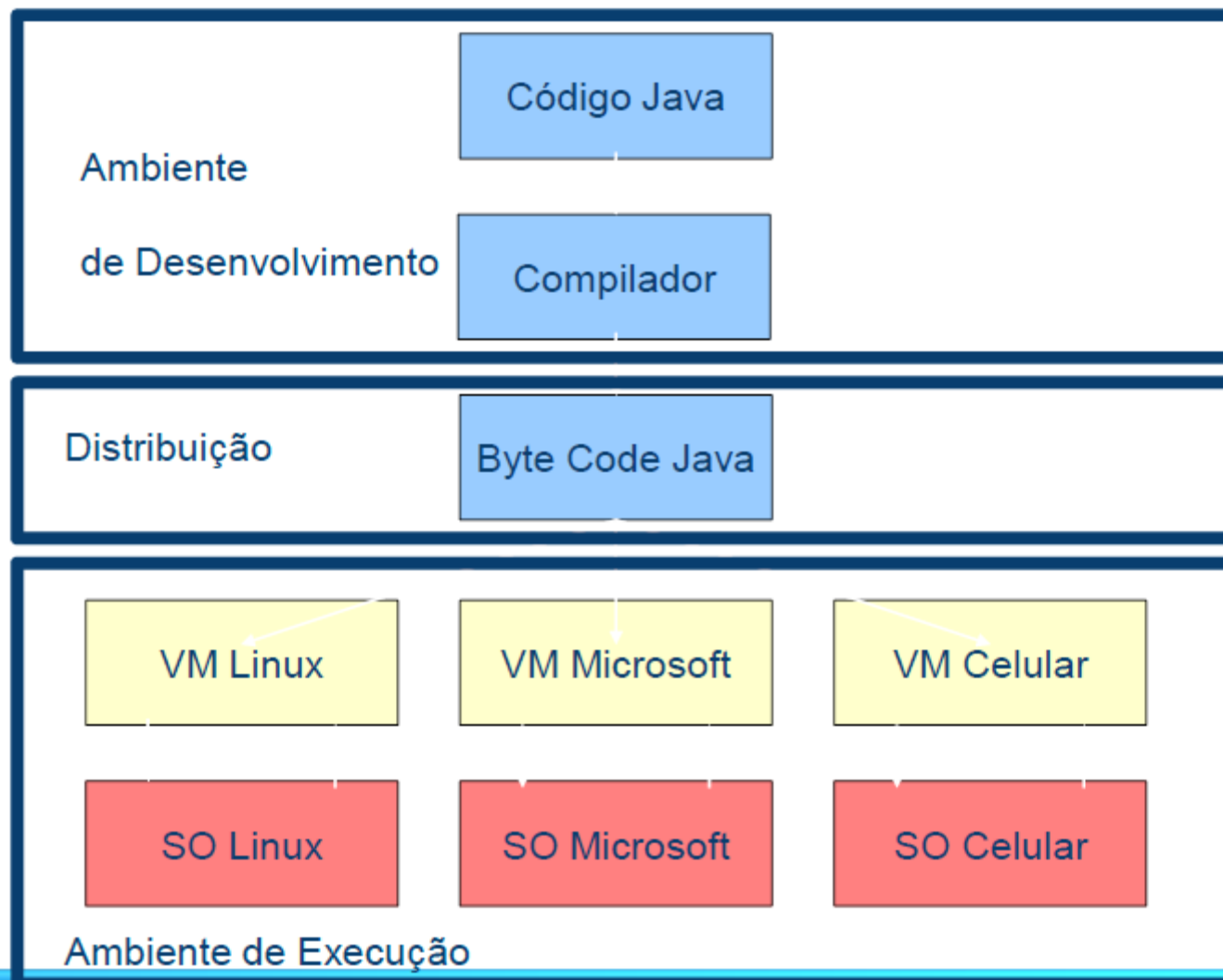
- ◉ Orientada a Objetos.
- ◉ Multithreading. Permite a execução de várias rotinas ao mesmo tempo.
- ◉ Suporte a comunicação.
- ◉ Acesso remoto a banco de dados.
- ◉ Baseada em C++:
 - Sintaxe semelhante;
 - Porém mais simples.
- ◉ Portabilidade (multiplataforma):
 - Compilação para bytecode e interpretação na JVM;

CRIAÇÃO DE PROGRAMAS EM JAVA





CRIAÇÃO DE PROGRAMAS EM JAVA EDERAL



Confusão do Java2 !!!

Java 1.0 e 1.1 são as versões muito antigas do Java, mas já traziam bibliotecas importantes como o JDBC e o java.io.

Com o Java 1.2 houve um aumento grande no tamanho da API, e foi nesse momento em que trocaram a nomenclatura de Java para Java2, com o objetivo de diminuir a confusão que havia entre Java e Javascript. Mas lembre-se, não há versão "**Java 2.0**", o 2 foi incorporado ao nome, tornando-se Java2 1.2.

Depois vieram o Java2 1.3 e 1.4, e o Java 1.5 passou a se chamar Java 5, tanto por uma questão de marketing e porque mudanças significativas na linguagem foram incluídas. É nesse momento que o "2" do nome Java desaparece. Repare que para fins de desenvolvimento, o Java 5 ainda é referido como Java 1.5.

Hoje a última versão disponível do **Java é a 8**.

EDIÇÕES DE JAVA

◉ Java é distribuída em três edições:

- Java Standard Edition (Java SE);
- Java Enterprise Edition (Java EE);
- Java Mobile Edition (Java ME).

JVM? JRE? JDK?

O que devo baixar?

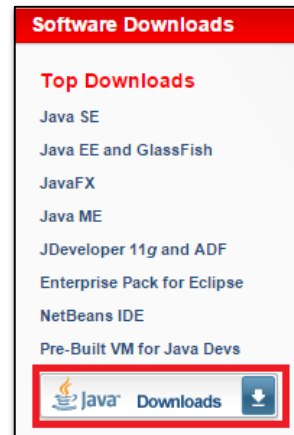
- JVM = apenas a virtual machine, esse download não existe, ela sempre vem acompanhada.
- JRE = **Java Runtime Environment**, ambiente de execução Java, formado pela JVM e bibliotecas, tudo que você precisa para executar uma aplicação Java. Mas nós precisamos de mais.
- JDK = **Java Development Kit**: Nós, desenvolvedores, faremos o download do JDK do Java SE (Standard Edition). Ele é formado pela JRE somado a ferramentas, como o compilador.

Tanto o **JRE** e o **JDK** podem ser baixados do site <http://www.oracle.com/technetwork/java/>. Para encontrá-los, acesse o link Java SE dentro dos top downloads. Consulte o apêndice de instalação do JDK para maiores detalhes.

3. Apêndice - Instalação do Java

Instalação do JDK em ambiente Windows

Para instalar o JDK no Windows, primeiro baixe-o no site da Oracle. É um simples arquivo executável que contém o Wizard de instalação: <http://www.oracle.com/technetwork/java/>



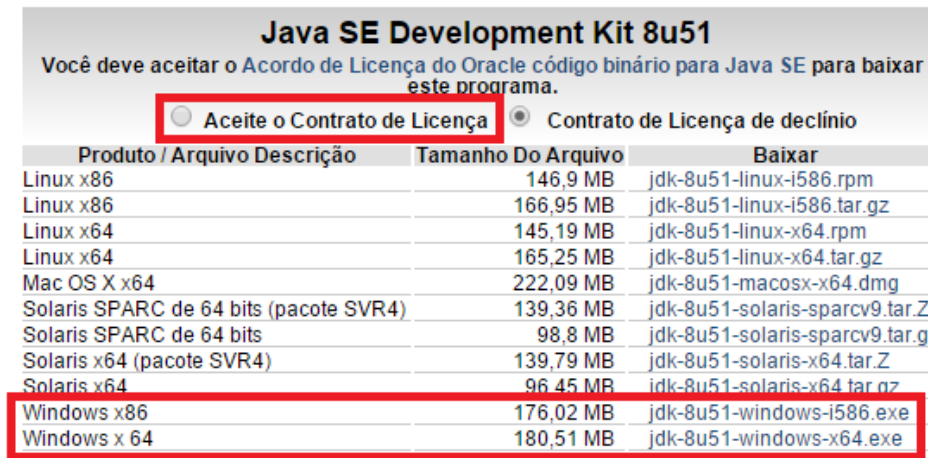
Instalação

1. Dê um clique na imagem Java.

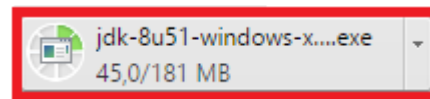


Instalação do JDK em ambiente Windows

2. Aceite o Contrato de Licença e escolha a arquitetura que representa seu Windows.



3. Aguarde o download do instalador.

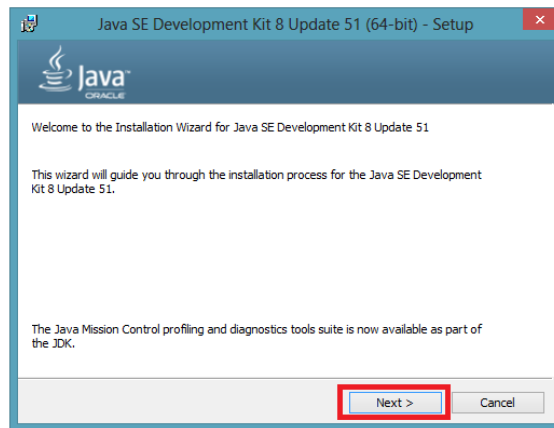


4. Dê um clique duplo no arquivo jdk-<versão>-windows-x<arquitetura>.exe e espere até ele entrar no wizard de instalação.



Instalação do JDK em ambiente Windows

5. Aceite os próximos dois passos clicando em *Next*. Após um tempo, o instalador pedirá para escolher em que diretório instalar o SDK. Pode ser onde ele já oferece como padrão. Anote qual foi o diretório escolhido, vamos utilizar esse caminho mais adiante.



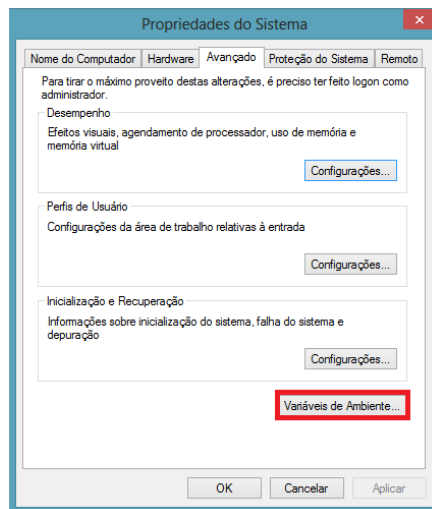
Configurando o ambiente

Precisamos configurar algumas variáveis de ambiente após a instalação, para que o compilador seja acessível via linha de comando. **Caso você vá utilizar diretamente o Eclipse, provavelmente não será necessário realizar esses passos.**

1. Clique com o botão direito em cima do ícone *Computador* e selecione a opção *Propriedades*.

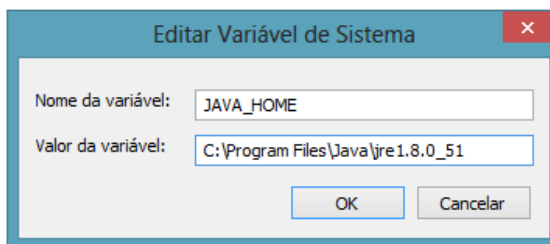
Instalação do JDK em ambiente Windows

2. Escolha a aba "Configurações Avançadas de Sistema" e depois clique no botão "Variáveis de Ambiente"



3. Nesta tela, você verá, na parte de cima, as variáveis de ambiente do usuário corrente e, embaixo, as variáveis de ambiente do computador (servem para todos os usuários). Clique no botão *Novo...* da parte de baixo.

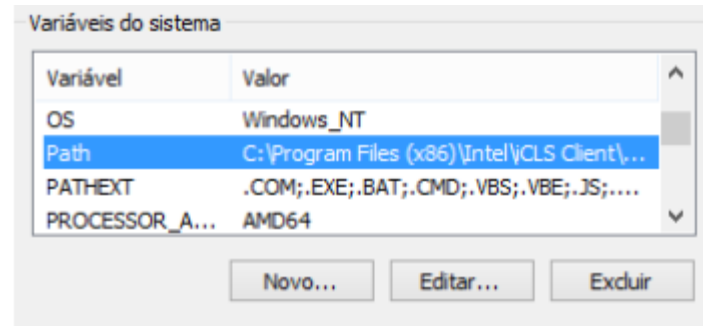
4. Em *Nome da Variável* digite JAVA_HOME e, em valor da variável, digite o caminho que você utilizou na instalação do Java. Provavelmente será algo como: C:\Program Files\Java\jdk1.8.0_51\:



(Clique em *Ok*)

Instalação do JDK em ambiente Windows

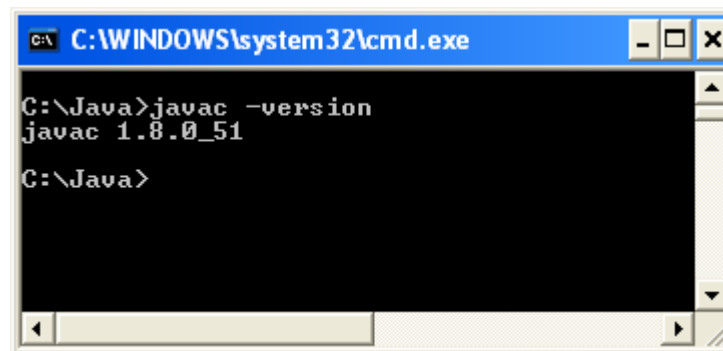
5. Não vamos criar outra variável, mas sim *alterar*. Para isso, procure a variável PATH, ou Path (dá no mesmo), e clique no botão de baixo "Editar".



6. Não altere o nome da variável! Deixe como está e adicione no final do valor;%JAVA_HOME%\bin, não esqueça do ponto-e-vírgula - assim, você está adicionando mais um caminho à sua variável Path.

7. Abra o prompt, indo em *Iniciar, Executar* e digite cmd

8. No console, digite javac -version. O comando deve mostrar a versão do Java Compiler e algumas opções.



Obs.: O comando apresentado não pode ser executado no Windows 8.

Compilando o primeiro programa

Vamos para o nosso primeiro código! O programa que imprime uma linha simples. Para mostrar uma linha, podemos fazer:

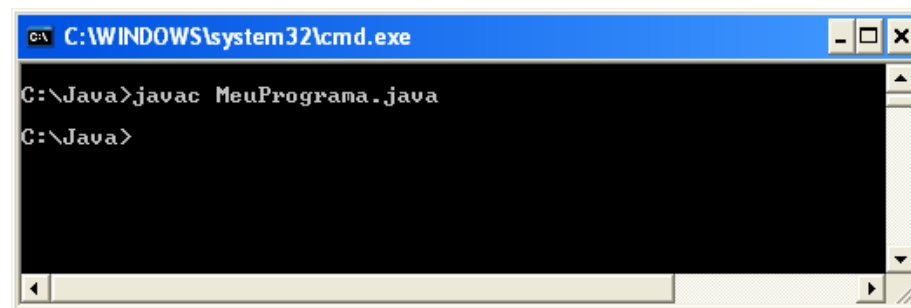
```
System.out.println("Minha primeira aplicação Java!");
```

Mas esse código não será aceito pelo compilador java. O Java é uma linguagem bastante burocrática e, precisa de mais do que isso para iniciar uma execução. Veremos os detalhes e os porquês durante as próximas aulas. O mínimo que precisaríamos escrever é algo como:

```
class MeuPrograma {  
    public static void main (String[] args {  
        System.out.println("Minha primeira aplicação Java!");  
    }  
}
```

Obs.: O Java é case sensitive, tome cuidado com maiúsculas e minúsculas.

Após digitar o código acima, grave-o como **MeuPrograma.java** em algum diretório. Para compilar, você deve pedir para que o compilador de Java da Oracle, chamado javac, gere o bytecode correspondente ao seu código Java.



Compilando o primeiro programa

Depois de compilar, o **bytecode** foi gerado. Quando o sistema operacional listar os arquivos contidos no diretório atual, você poderá ver que um arquivo **.class** foi gerado, com o mesmo nome da sua classe Java.



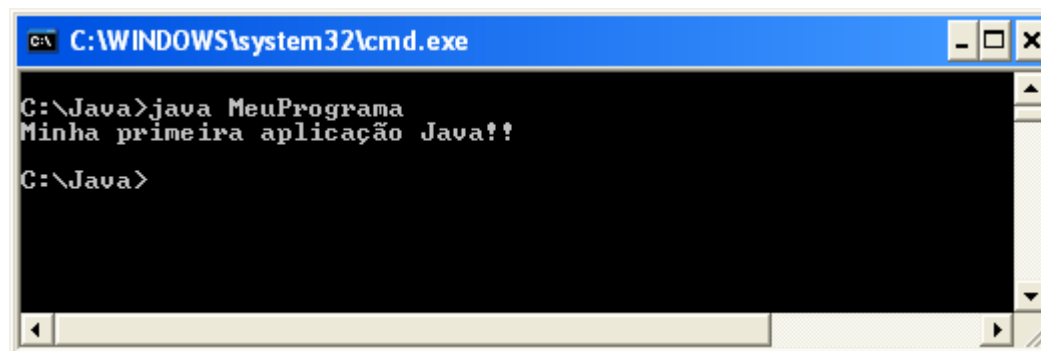
```
C:\WINDOWS\system32\cmd.exe

Pasta de C:\Java
25/07/2015  15:08    <DIR>          .
25/07/2015  15:08    <DIR>          ..
25/07/2015  15:08                449 MeuPrograma.class
25/07/2015  15:07                126 MeuPrograma.java
                2 arquivo(s)                575 bytes
                2 pasta(s) 84.265.050.112 bytes disponíveis

C:\Java>
```

EXECUTANDO SEU PRIMEIRO PROGRAMA

Os procedimentos para executar seu programa são muito simples. O **javac** é o compilador Java, e o **java** é o responsável por invocar a máquina virtual para interpretar o seu programa.



```
C:\WINDOWS\system32\cmd.exe

C:\Java>java MeuPrograma
Minha primeira aplicação Java!!

C:\Java>
```

O QUE ACONTECEU ???

```
1  class MeuPrograma {
2      public static void main(String[] args) {
3
4          // miolo do programa começa aqui!
5          System.out.println("Minha primeira aplicação Java!!");
6          // fim do miolo do programa
7
8      }
9  }
```

O miolo do programa é o que será executado quando chamamos a máquina virtual. Por enquanto, todas as linhas anteriores, onde há a declaração de uma classe e a de um método, não importam para nós nesse momento. Mas devemos saber que toda aplicação Java começa por um ponto de entrada, e este ponto de entrada é o método **main**.

No caso do nosso código, a linha do `System.out.println` faz com que o conteúdo entre aspas seja colocado na tela.

// (comentário de única linha)

- indica que o restante da linha é um comentário

/* (comentário de múltiplas linhas) */

- pode ser dividido em várias linhas

/** (comentário de documentação) */

- programa utilitário javadoc - lê esses comentários e usa os mesmos para preparar a documentação do programa

Documentar programas e melhorar a legibilidade

Dica: todo programa deve iniciar com um comentário indicando o propósito do programa.

3. Variáveis

Dentro de um bloco, podemos declarar variáveis e usá-las. Em Java, toda variável tem um tipo que não pode ser mudado, uma vez que declarado:

tipoDaVariavel nomeDaVariavel;

Por exemplo, é possível ter uma idade que guarda um número inteiro:

int idade;

Com isso, você declara a variável **idade**, que passa a existir a partir daquela linha. Ela é do tipo **int**, que guarda um número inteiro. A partir daí, você pode usá-la, primeiramente atribuindo valores.

A linha a seguir é a tradução de: "**idade deve valer quinze**".

idade = 15;

Você pode usar os operadores +, -, / e * para operar com números, sendo eles responsáveis pela adição, subtração, divisão e multiplicação, respectivamente. Além desses operadores básicos, há o operador % (módulo) que nada mais é que o **resto de uma divisão inteira**. Veja alguns exemplos:

Tipo de operadores

- Aritméticos

- +
- -
- *
- /
- % (módulo) “resto da divisão”

- Igualdade

- ==
- !=

- Relacionais

- > (Maior que)
- < (Menor que)
- >= (Maior ou igual)
- <= (Menor ou igual)

- Lógicos

- && (E lógico)
- || (OU lógico)
- ! (NÃO lógico)

- Bitwise (bit a bit)

- & (E bit a bit)
- | (OU bit a bit)

Tipo de operadores

| Operadores | Associatividade | Tipo |
|---------------------|--------------------------|----------------|
| () | Da esquerda para direita | parênteses |
| * / % | Da esquerda para direita | multiplicativo |
| + - | Da esquerda para direita | aditivo |
| < <= > >= | Da esquerda para direita | relacional |
| = = ! = | Da esquerda para direita | igualdade |
| && | Da esquerda para direita | E lógico |
| | Da esquerda para direita | Ou lógico |
| (teste)? se V: se F | Da direita para esquerda | condicional |
| = += -= *= /= %= | Da direita para esquerda | de atribuição |

Operadores de incremento e decremento

| Operador | Chamado de | Expressão de exemplo | Explicação |
|----------|----------------|----------------------|--|
| ++ | Pré-incremento | ++a | Incrementa a por 1, depois utiliza o novo valor de a na expressão em que a reside. |
| ++ | Pós-incremento | a++ | Utiliza o valor atual de a na expressão em que a reside. |
| -- | Pré-decremento | --b | Decrementa b por 1, depois utiliza o novo valor de b na expressão em que b reside. |
| -- | Pós-decremento | b-- | Utiliza o valor atual de b na expressão em que b reside. |

```
int quatro = 2 + 2;  
int tres = 5 - 2;  
int oito = 4 * 2;  
int dezesseis = 64 / 4;  
int um = 5 % 2;    // 5 dividido por 2 dá 2 e tem resto 1;  
                  // o operador % pega o resto da divisão inteira
```

Comentários em Java

Para fazer um comentário em java, você pode usar o `//` para comentar até o final da linha, ou então usar o `/* */` para comentar o que estiver entre eles.

```
/* comentário daqui, ate aqui */
```

```
// uma linha de comentário sobre a idade
```

```
int idade;
```

Declarando e usando variáveis

Representar números inteiros é fácil, mas como guardar valores reais, tais como frações de números inteiros e outros? Outro tipo de variável muito utilizado é o `double`, que armazena um número com ponto flutuante (e que também pode armazenar um número inteiro).

```
double pi = 3.14;  
double x = 5 * 10;
```

O tipo `boolean` armazena um valor verdadeiro ou falso, e só: nada de números, palavras ou endereços, como em algumas outras linguagens.

```
boolean verdade = true;
```

`true` e `false` são palavras reservadas do Java. É comum que um `boolean` seja determinado através de uma **expressão booleana**, isto é, um trecho de código que retorna um booleano, como o exemplo:

```
int idade = 30;  
boolean menorDedade = idade < 18;
```

O tipo `char` guarda um, e apenas um, caractere. Esse caractere deve estar entre aspas simples. Não se esqueça dessas duas características de uma variável do tipo `char`! Por exemplo, ela não pode guardar um código como `"` pois o vazio não é um caractere!

```
char letra = 'a';
```

Outros tipos primitivos

Vimos aqui os tipos primitivos que mais aparecem. O Java tem outros, que são o **byte**, **short**, **long** e **float**.

Cada tipo possui características especiais que, para um programador avançado, podem fazer muita diferença, pois cada tipo tem seus tamanhos variável. Observe a tabela abaixo:

| Tipos | Tamanho | Característica |
|---------|---------|------------------|
| Boolean | 1 bit | Verdadeiro/Falso |
| Byte | 1 byte | Numérico |
| Short | 2 bytes | Numérico |
| Char | 2 bytes | Alfanumérico |
| Int | 4 bytes | Numérico |
| Float | 4 bytes | Ponto Flutuante |
| Long | 8 bytes | Numérico |
| Double | 8 bytes | Ponto Flutuante |

4. CASTING

Casting e Promoção

Alguns valores são incompatíveis se você tentar fazer uma atribuição direta. Enquanto um número real costuma ser representado em uma variável do tipo `double`, tentar atribuir ele a uma variável `int` não funciona porque é um código que diz: "**i deve valer d**", mas não se sabe se `d` realmente é um número inteiro ou não.

```
double d = 3.1415;  
int i = d; // não compila
```

O mesmo ocorre no seguinte trecho:

```
int i = 3.14;
```

O mais interessante, é que nem mesmo o seguinte código compila:

```
double d = 5; // ok, o double pode conter um número inteiro  
int i = d; // não compila
```

O código acima compila sem problemas, já que um `double` pode guardar um número com ou sem ponto flutuante. Todos os inteiros representados por uma variável do tipo `int` podem ser guardados em uma variável `double`, então não existem problemas no código acima.

Casting e Promoção

Às vezes, precisamos que um número quebrado seja arredondado e armazenado num número inteiro. Para fazer isso sem que haja o erro de compilação, é preciso ordenar que o número quebrado seja **moldado (casted)** como um número inteiro. Esse processo recebe o nome de **casting**.

```
double d3 = 3.14;
```

```
int i = (int) d3;
```

O casting foi feito para moldar a variável d3 como um int. O valor de i agora é 3.

O mesmo ocorre entre valores int e long.

```
long x = 10000;
```

```
int i = x; // não compila, pois pode estar perdendo informação
```

E, se quisermos realmente fazer isso, fazemos o casting:

```
long x = 10000;
```

```
int i = (int) x;
```

Casting e Promoção

CASOS NÃO TÃO COMUNS DE CASTING E ATRIBUIÇÃO

Alguns **castings** aparecem também:

```
float x = 0.0;
```

O código acima não compila pois todos os literais com ponto flutuante são considerados **double** pelo Java. E **float** não pode receber um **double** sem perda de informação, para fazer isso funcionar podemos escrever o seguinte:

Outro caso, que é mais comum:

```
double d = 5;  
float f = 3;  
float x = f + (float) d;
```

Você precisa do casting porque o Java faz as contas e vai armazenando sempre no maior tipo que apareceu durante as operações, no caso o **double**.

E, uma observação: no mínimo, o Java armazena o resultado em um **int**, na hora de fazer as contas.

Até casting com variáveis do tipo **char** podem ocorrer. O único tipo primitivo que não pode ser atribuído a nenhum outro tipo é o **boolean**.

Casting e Promoção

CASTINGS POSSÍVEIS

Abaixo estão relacionados todos os casts possíveis na linguagem Java, mostrando a conversão **de** um valor **para** outro. A indicação **Impl.** quer dizer que aquele cast é implícito e automático, ou seja, você não precisa indicar o cast explicitamente (lembrando que o tipo boolean não pode ser convertido para nenhum outro tipo).

| PARA: | byte | short | char | int | long | float | double |
|---------------|--------|--------------|--------|--------------|--------------|--------------|--------------|
| DE: | byte | short | char | int | long | float | double |
| 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) | ---- |

5. Estruturas de Decisão

O IF e o ELSE

A sintaxe do **if** no Java é a seguinte:

```
if (condicaoBooleana) {  
    codigo;  
}
```

Uma **condição booleana** é qualquer expressão que retorne **true** ou **false**. Para isso, você pode usar os operadores **<**, **>**, **<=**, **>=** e outros. Um exemplo:

```
int idade = 15;  
if (idade < 18) {  
    System.out.println("Não pode entrar");  
}
```

Além disso, você pode usar a cláusula **else** para indicar o comportamento que deve ser executado no caso da expressão booleana ser falsa:

```
int idade = 15;  
if (idade < 18) {  
    System.out.println("Não pode entrar");  
} else {  
    System.out.println("Pode entrar");  
}
```

O IF e o ELSE

Você pode concatenar expressões booleanas através dos operadores lógicos "E" e "OU". O "E" é representado pelo **&&** e o "OU" é representado pelo **||**.

Um exemplo seria verificar se ele tem menos de 18 anos e se ele não é amigo do dono:

```
int idade = 15;
boolean amigoDoDono = true;
if (idade < 18 && amigoDoDono == false) {
    System.out.println("Não pode entrar");
} else {
    System.out.println("Pode entrar");
}
```

Esse código poderia ficar ainda mais legível, utilizando-se o operador de negação, o **!**. Esse operador transforma o resultado de uma expressão booleana de **false** para **true** e vice versa.

```
int idade = 15;
boolean amigoDoDono = true;
if (idade < 18 && !amigoDoDono) {
    System.out.println("Não pode entrar");
} else {
    System.out.println("Pode entrar");
}
```


O IF e o ELSE

Repare na linha 3 que o trecho **amigoDoDono == false** virou **!amigoDoDono**. Eles têm o mesmo valor.

Para comparar se uma variável tem o **mesmo valor** que outra variável ou valor, utilizamos o operador **==**. Repare que utilizar o operador **=** dentro de um **if** vai retornar um erro de compilação, já que o operador **=** é o de atribuição.

```
int mes = 1;
if (mes == 1) {
    System.out.println("Você deveria estar de férias");
}
```

5. Estruturas de Repetição

O while é um comando usado para fazer um **laço (loop)**, isto é, repetir um trecho de código algumas vezes. A ideia é que esse trecho de código seja repetido enquanto uma determinada condição permanecer verdadeira.

```
int idade = 15;
while (idade < 18) {
    System.out.println(idade);
    idade = idade + 1;
}
```

O trecho dentro do bloco do **while** será executado até o momento em que a condição **idade < 18** passe a ser falsa. E isso ocorrerá exatamente no momento em que **idade == 18**, o que não o fará imprimir **18**.

```
int i = 0;
while (i < 10) {
    System.out.println(i);
    i = i + 1;
}
```

Já o **while** acima imprime de 0 a 9.

Outro comando de **loop** extremamente utilizado é o **FOR**. A ideia é a mesma do **While**: fazer um trecho de código ser repetido enquanto uma condição continuar verdadeira. Mas além disso, o **FOR** isola também um espaço para inicialização de variáveis e o modificador dessas variáveis. Isso faz com que fiquem mais legíveis, as variáveis que são relacionadas ao loop:

```
for (inicializacao; condicao; incremento) {  
    codigo;  
}
```

Um exemplo é o a seguir:

```
for (int i = 0; i < 10; i = i + 1) {  
    System.out.println("olá!");  
}
```

Repare que esse for poderia ser trocado por:

```
int i = 0;  
while (i < 10) {  
    System.out.println("olá!");  
    i = i + 1;  
}
```

Pós incremento ++

`i = i + 1` pode realmente ser substituído por `i++` quando isolado, porém, em alguns casos, temos essa instrução envolvida em, por exemplo, uma atribuição:

```
int i = 5;  
int x = i++;
```

Qual é o valor de `x`? O de `i`, após essa linha, é 6.

O operador `++`, quando vem após a variável, retorna o valor antigo, e incrementa (pós incremento), fazendo `x` valer 5.

Se você tivesse usado o `++` antes da variável (pré incremento), o resultado seria 6:

```
int i = 5;  
int x = ++i; // aqui x valera 6
```

Controles de Loops

Apesar de termos condições booleanas nos nossos laços, em algum momento, podemos decidir parar o loop por algum motivo especial sem que o resto do laço seja executado.

```
for (int i = x; i < y; i++) {  
    if (i % 19 == 0) {  
        System.out.println("Achei um número divisível por 19 entre x e y");  
        break;  
    }  
}
```

O código acima vai percorrer os números de x a y e parar quando encontrar um número divisível por 19, uma vez que foi utilizada a palavra chave **break**.

Da mesma maneira, é possível obrigar o loop a executar o próximo laço. Para isso usamos a palavra chave **continue**.

```
for (int i = 0; i < 100; i++) {  
    if (i > 50 && i < 60) {  
        continue;  
    }  
    System.out.println(i);  
}
```

O código acima não vai imprimir alguns números. (**Quais exatamente?**)

Escopo das Variáveis

No Java, podemos declarar variáveis a qualquer momento. Porém, dependendo de onde você as declarou, ela vai valer de um determinado ponto a outro.

```
// aqui a variável i não existe  
int i = 5;  
// a partir daqui ela existe
```

O **escopo da variável** é o nome dado ao trecho de código em que aquela variável existe e onde é possível acessá-la.

Quando abrimos um novo bloco com as chaves, as variáveis declaradas ali dentro **só valem até o fim daquele bloco**.

```
// aqui a variável i não existe  
int i = 5;  
// a partir daqui ela existe  
while (condicao) {  
    // o i ainda vale aqui  
    int j = 7;  
    // o j passa a existir  
}  
// aqui o j não existe mais, mas o i continua dentro do escopo
```

No bloco acima, a variável *j* pára de existir quando termina o bloco onde ela foi declarada. Se você tentar acessar uma variável fora de seu escopo, ocorrerá um erro de compilação.

Escopo das Variáveis

```
EscopoDeVariavel.java:8: cannot find symbol
symbol : variable j
location: class EscopoDeVariavel
    System.out.println(j);
                        ^
1 error
```

O mesmo vale para um if:

```
if (algumBooleano) {
    int i = 5;
} else {
    int i = 10;
} System.out.println( i ); // cuidado!
```

Aqui a variável **i** não existe fora do **if** e do **else**! Se você declarar a variável antes do **if**, vai haver outro erro de compilação: dentro do **if** e do **else** a variável está sendo redeclarada! Então o código para compilar e fazer sentido fica:

```
int i;
if (algumBooleano) {
    i = 5;
} else {
    i = 10;
} System.out.println( i );
```


Escopo das Variáveis

Uma situação parecida pode ocorrer com o for:

```
for (int i = 0; i < 10; i++) {  
    System.out.println("olá!");  
}
```

```
System.out.println(i); // cuidado!
```

Neste for, a variável **i** morre ao seu término, não podendo ser acessada de fora do for, gerando um erro de compilação. Se você realmente quer acessar o contador depois do loop terminar, precisa de algo como:

```
int i;  
for (i = 0; i < 10; i++) {  
    System.out.println("olá!");  
}  
System.out.println(i);
```