Use a Cabeça Java™

Tradução da 3ª Edição

Kathy Sierra
Bert Bates
Trisha Gee

ALTA BOOKS
GRUPO EDITORIAL
Rio de Janeiro, 2024

Sumário

	Intro	xxi
1	Águas Javônicas mergulhe: *um breve tchibum*	1
2	Um Passeio a Objetópolis: *classes e objetos*	27
3	Conheça Suas Variáveis: *variáveis primitivas e de referência*	49
4	Como os Objetos Se Comportam: *métodos usam variáveis de instância*	71
5	Métodos Extrafortes: *escrevendo um programa*	95
6	Usando a biblioteca Java: *conheça a API Java*	125
7	Vivendo Melhor em Objetópolis: *herança e poliformismo*	167
8	Polimorfismo Levado a Sério: *interfaces e classes abstratas*	199
9	Vida e Morte de Um Objeto: *construtores e garbage collection*	237
10	Os Números Importam: *números e elementos estáticos*	273
11	Estruturas de Dados: *coleções e generics*	307
12	Lambdas e Streams: *O Que, Não Como: lambdas e streams*	369
13	Comportamento Arriscado: *tratando exceções*	421
14	Uma História Muito Gráfica: *criando uma GUI*	461
15	Trabalhando seu Swing: *usando o swing*	509
16	Salvando Objetos e Texto: *serialização e E/S de arquivo*	539
17	Faça uma Conexão: *redes e threads*	587
18	Lidando Com Problemas de Concorrência: *race conditions e dados imutáveis*	639
A	Apêndice A: *Receita Final de Código*	673
B	Apêndice B: A volta dos que não foram: *os Onze Principais Tópicos que não apareceram no livro…*	683
	Índice	701

Conteúdo (o papo sério)

Introdução

Seu cerébro no modo Java. Aí está *você*, tentando *aprender* algo novo, mas seu *cérebro* parece ter outras ideias, evitando que você *absorva* o conteúdo. Ele tá tipo: "Melhor guardar espaço para coisas mais urgentes, tipo quais animais selvagens evitar ou se fazer snowboard peladão é má ideia". Então, como *fazer* seu cérebro acreditar que sua vida depende de dominar o Java?"

Para quem é este livro?	xxiv
Sabemos o que está pensando	xxv
Metacognição: pensando sobre pensar	xxvii
Veja o que NÓS fizemos:	xxviii
Veja o que VOCÊ pode fazer para que seu cérebro lhe obedeça	xxix
O que você precisa para este livro:	xxx
Coisas de última hora que você precisa saber:	xxxi

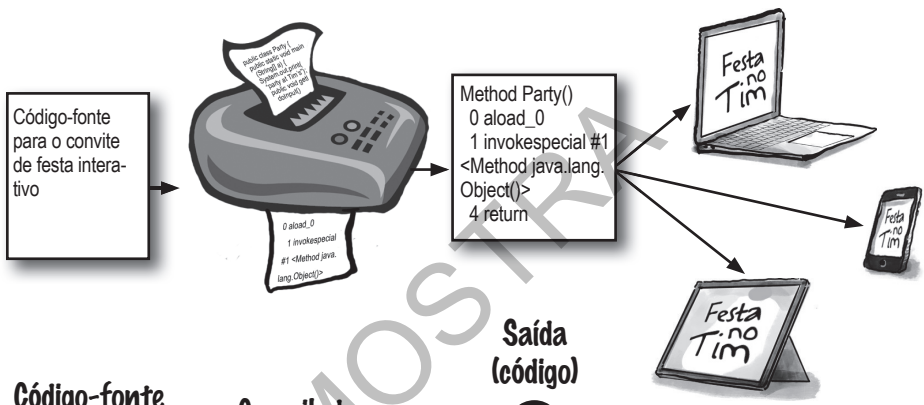
Águas Javônicas

O Java eleva você a novos patamares. Desde o singelo lançamento da (frágil) versão 1.02 ao público, o Java enfeitiçou os programadores com sua sintaxe amigável, recursos orientados a objetos, gerenciamento de memória e, o melhor de tudo — a promessa de portabilidade. O atrativo de **escrever uma vez/executar em qualquer lugar** é forte demais. Uma comunidade devotada deslanchou, enquanto os programadores se debatiam contra bugs, limitações e, ah sim, o fato de que a linguagem era lentíssima. Mas isso foi há muito tempo. Se estiver começando a programar em Java, **você tem sorte**.

Alguns de nós arrancaram todos os fios de cabelo até fazer com que uma simples aplicação rodasse. Mas *você*, veja só, *você* pode usar o Java atual: **elegantíssimo, rápido e fácil de ler e escrever**.

Como o Java funciona

O objetivo é escrever uma aplicação (neste exemplo, um convite de festa interativo) e fazê-la rodar em qualquer dispositivo que seus amigos tenham.



Código-fonte

1

Crie um documento para o código-fonte. Use um protocolo estabelecido (nesse caso, a linguagem Java).

Compilador

2

Rode seu documento por meio de um compilador de código-fonte. O compilador verifica se há erros e não permite que você compile nada até que esteja convencido de que tudo executará corretamente.

Saída (código)

3

O compilador cria um documento novo, codificado em ***bytecode*** Java. Qualquer dispositivo capaz de rodar Java poderá interpretar/converter esse arquivo em algo que possa ser processado. O bytecode compilado independe de plataforma.

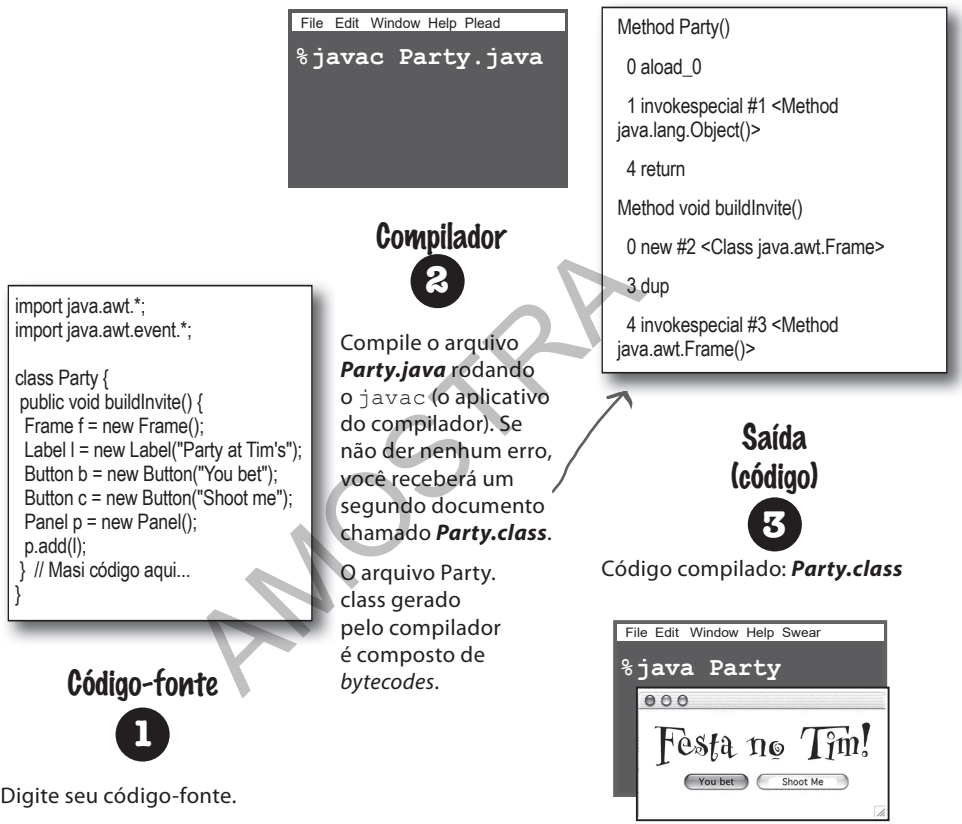
Máquinas virtuais

4

Todos os seus amigos têm uma máquina ***virtual*** Java (JVM), implementada em software, rodando dentro de seus dispositivos eletrônicos. Quando seus amigos executam seu programa, a máquina virtual lê e executa o bytecode.

O que você fará no Java

Você criará um arquivo de código-fonte, o compilará com o compilador javac e, em seguida, executará o bytecode compilado em uma máquina virtual Java.



(Observação: isso **NÃO** é um tutorial... Logo mais, você escreverá código real, mas por ora, queremos apenas que tenha uma ideia de como tudo se encaixa.

Em outras palavras, o código dessa página não é real; nada de tentar compilá-lo.)

Uma breve história do Java

A princípio, o Java foi lançado (boatos diriam que "escapou"), em 23 de janeiro de 1996. A linguagem tem mais de 25 anos! Nos primeiros 25 anos, o Java evoluiu, e a API Java progrediu bastante. A melhor estimativa que temos é que, nos últimos 25 anos, mais de 17 zilhões de linhas de código Java foram escritas. À medida que você dedica tempo programando em Java, com certeza vai se deparar com códigos Java antiquíssimos e com outros mais recentes. A linguagem Java é famosa por sua compatibilidade com versões anteriores, ou seja, códigos antigos podem ser executados tranquilamente com as novas JVMs.

Neste livro, começaremos usando estilos mais antigos de programação (lembre-se, é provável que você se depare com códigos assim no "mundo real") e, em seguida, apresentaremos um estilo de código mais novo. Do mesmo modo, às vezes mostraremos classes mais antigas na API Java e, depois, alternativas mais recentes.

Velocidade e uso de memória

Da primeira vez que foi lançado, o Java era lento. Mas logo depois, a VM do HotSpot foi criada, assim como outras melhorias de desempenho. Mesmo sendo verdade que o Java não é lá a linguagem mais rápida que existe, é considerada bastante rápida — quase tanto quanto C e Rust, e ***bem*** mais rápida do que a maioria das linguagens existentes.

O Java tem um superpoder mágico — a JVM. A Java Virtual Machine consegue otimizar seu código *enquanto está em execução*, assim é possível criar aplicações muito rápidas sem ter que escrever código de alto desempenho especializado.

Mas — sinceridade total — em comparação com C e Rust, o Java usa bastante memória.

Aponte o seu lápis

Respostas na página 6.

Veja como é fácil programar em Java

Tente adivinhar o que cada linha de código faz… (as respostas estão na próxima página).

```
int size = 27;
String name = "Fido";
Dog myDog = new Dog(name, size);
x = size - 5;
if (x < 15) myDog.bark(8);

while (x > 3) {
  myDog.play();
}

int[] numList = {2, 4, 6, 8};
System.out.print("Hello");
System.out.print("Dog: " + name);
String num = "8";
int z = Integer.parseInt(num);

try {
  readTheFile("myFile.txt");
}
catch (FileNotFoundException ex) {
  System.out.print("File not
found.");
}
```

declara uma variável com um inteiro chamada 'size' e lhe atribui o valor 27

se x (valor de 22) for menor que 15, diz a dog para latir 8 vezes

exibe "Hello"... provavelmente na linha de comando

P: As convenções de nomenclatura para as versões do Java são confusas. Existem JDK 1.0, 1.2, 1.3 e 1.4. Daí, do nada, temos o J2SE 5.0, que mudou para Java 6, Java 7, e da última vez que verifiquei, existia versão do Java até o Java 18. O que está rolando?

R: Os números das versões variaram bastante nos últimos 25 anos! Podemos ignorar as letras (J2SE/SE), já que atualmente não são mais usadas. Quanto aos números, as coisas são um pouco mais complicadas. Tecnicamente, o Java SE 5.0 era na verdade o Java **1.**5. O mesmo vale para o Java 6 (1.6), Java 7 (1.7) e Java 8 (1.8). Em teoria, o Java ainda está na versão 1.x, já que as versões novas são compatíveis com as versões anteriores, todas remontam à versão 1.0. Contudo, era um tanto confuso ter um número de versão diferente do nome que todos usavam. Logo, o número de versão oficial do Java 9 em diante é apenas o número, sem o prefixo "1"; ou seja, o Java 9 realmente é a versão 9, não a versão 1.9. Neste livro, usaremos a convenção comum de 1.0–1.4 e, a partir de 5, descartaremos o prefixo "1". Além disso, desde que o Java 9 foi lançado em setembro de 2017, a cada seis meses um Java era lançado, cada um com um novo número de versão "principal". Por isso, passamos bem rápido do 9 para o 18!

Aponte o seu lápis respostas

Veja como é fácil programar em Java

Não se preocupe se ainda não entendeu nada com coisa alguma! Neste livro, explicamos tudo nos mínimos detalhes (principalmente nas 40 primeiras páginas). Caso o Java seja parecido com alguma linguagem que você já tenha usado, será mais fácil. Caso contrário, não se preocupe. *Chegaremos lá…*

`int size = 27;`	declara a variável 'size' com um inteiro e lhe atribui o valor 27
`String name = "Fido";`	declara a variável 'name' com uma string de caracteres e lhe atribui o valor "Fido"
`Dog myDog = new Dog(name, size);`	declara uma variável nova 'myDog' e a faz usar 'name' and 'size'
`x = size - 5;`	subtrai 5 de 27 (valor de 'size') e a atribui a uma variável chamada 'x'
`if (x < 15) myDog.bark(8);`	se x (valor de 22) for menor que 15, diz a dog para latir 8 vezes
`while (x > 3) {`	o loop continua enquanto x for maior que 3...
`  myDog.play();`	diz ao dog para brincar (seja lá o QUE isso signifique para um dog...)
`}`	isso se parece com o final do loop -- tudo em {} é feito no loop
`int[] numList = {2, 4, 6, 8};`	declara uma lista de variáveis de inteiros 'numList' e insere 2,4,6,8 na lista.
`System.out.print("Hello");`	exibe "Hello"... provavelmente na linha de comando
`System.out.print("Dog: " + name);`	exibe "Dog: Fido" (o valor de 'name' é "Fido") na linha de comando
`String num = "8";`	declara variável 'num' com uma string de caracteres e lhe atribui o valor de "8"
`int z = Integer.parseInt(num);`	converte a string de caracteres "8" em um valor numérico real 8
`try {`	o try tenta fazer algo... talvez o que estamos tentando não funcione...
`  readTheFile("myFile.txt");`	lê o arquivo de texto chamado "myFile.txt" (pelo menos ,TENTA ler o arquivo...)
`}`	talvez seja o fim do "tentar fazer algo", acho que você pode tentar outra coisa...
`catch (FileNotFoundException ex) {`	talvez seja aqui que você descubra se o que tentou não funcionou...
`  System.out.print("File not found.");`	se o que tentamos falhou, exibe "File not found"na linha de comando
`}`	parece que tudo dentro das { } é o que temos que fazer se o "try" não funcionar...

Estrutura do código Java

O que tem dentro de um arquivo-FONTE?

Um arquivo de código-fonte (com a extensão *.java*) contém uma definição de ***classe***. A classe representa uma *parte* do seu programa, mesmo que uma aplicação pequena precise de uma única classe. A classe deve ficar dentro de um par de chaves.

```
public class Dog {



}
```

classe

No arquivo-fonte, insira uma classe.

Na classe, insira os métodos.

No método, insira as instruções.

O que tem dentro de uma classe?

Uma classe tem um ou mais ***métodos***. Na classe Dog, o método ***bark*** conterá instruções sobre como um cachorro deve latir. Seus métodos devem ser declarados *dentro* de uma classe (ou seja, dentro de um par de chaves).

```
public class Dog {
   void bark() {



   }
}
```

método

O que tem dentro de um método?

Dentro das chaves de um método, escreva as instruções sobre como esse método deve ser executado. O *código* do método é basicamente um conjunto de instruções, e, por ora, considere um método como uma função ou procedimento.

```
public class Dog {
   void bark() {
     instrução1;
     instrução2;
   }
}
```

instruções

Estrutura anatômica de uma classe

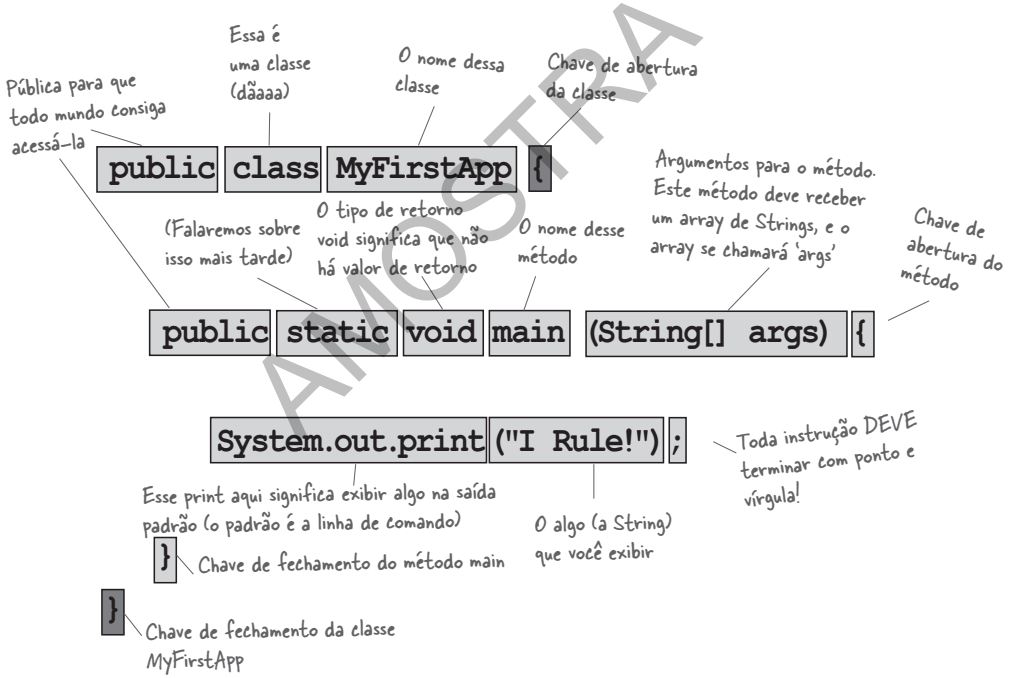
Assim que começa a rodar, a JVM procura a classe que você imputou na linha de comando. Em seguida, começa a procurar um método expressamente escrito que se pareça exatamente com:

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

   // Seu código entra aqui

}
```

Depois, a JVM executa tudo que estiver entre as chaves { } do seu método principal. Toda aplicação Java deve ter, pelo menos, uma **classe** e um método **main** (não um método main por *classe*; somente um por *aplicação*).



Não se preocupe em memorizar tudo de imediato... Você está apenas no começo.

Escrevendo uma classe com o método main()

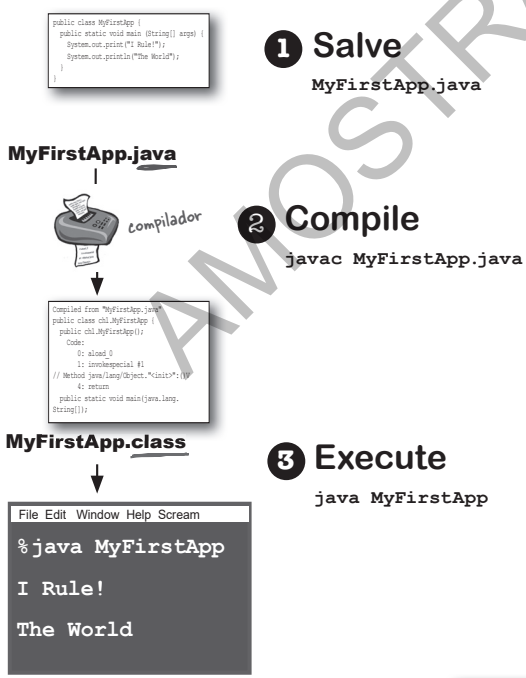
No Java, inserimos tudo em uma **classe**. Você cria um arquivo de código-fonte (com a extensão *.java*) e, depois, o compila dentro de um novo arquivo de classe (com a extensão *.class*). Ao executar seu programa, na realidade está executando uma classe.

Executar um programa é o mesmo que dizer à Java Virtual Machine (JVM): "Carregue a classe **MyFirstApp** e, em seguida execute seu método **main()**. Não pare de executar até que todo o código main termine."

No Capítulo 2, *Um Passeio a Objetópolis*, nos aprofundaremos no conceito de *classe*, mas por ora, a única pergunta que precisa fazer é ***como escrevo um código Java que execute sem problemas?*** Tudo começa com o método **main**().

O método **main**() é onde seu programa começa a ser executado.

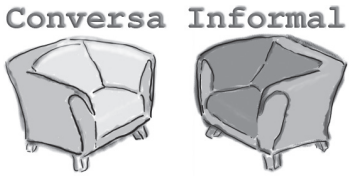
Independentemente do tamanho de seu programa (dito de outro modo, não importa quantas *classes* seu programa use), é necessário um método **main**() para dar o pontapé inicial.




```
public class MyFirstApp {

   public static void main (String[] args) {
     System.out.println("I Rule!");
     System.out.println("The World");
   }

}
```



Conversa da noite: **O compilador e a JVM debatem a questão: "Quem é mais importante?"**

A Máquina Virtual Java

Pera, você tá de brincadeira? ***Fala sério!*** Eu sou a linguagem Java. Sou eu quem executa um programa. O compilador apenas fornece um arquivo e acabou. Só um arquivo. Podemos imprimi-lo para usá-lo como papel de parede, fazer uma fogueira, forrar uma gaiola, o que for. O arquivo não faz nada, a menos que eu esteja lá para rodá-lo.

E tem mais isso, o compilador não tem senso de humor. Mas também, né? Se eu tivesse que passar o dia todo catando violaçõezinhas insignificantes de sintaxe…

Não estou dizendo que você é um inútil *completo*. Mas, sério, o que você faz? Sério mesmo. Não faço ideia. Se um programador escrever bytecode à mão eu consigo rodar. Talvez você fique desempregado em breve, parça.

(Depois dessa, não tenho nem mais o que falar sobre teu senso de humor.) Mas você ainda não respondeu minha pergunta: o que você faz *de verdade*?

O compilador

Não estou gostando nada desse tom.

Como é? Sem *mim*, você executaria o quê? Pois fique sabendo que existe um *motivo* para o Java ter sido arquitetado com um compilador de bytecode. Se fosse uma linguagem puramente interpretativa, em que — no tempo de execução — a máquina virtual tivesse que converter diretamente o código-fonte de um editor de texto, um programa Java seria executado em uma velocidade absurdamente lenta.

Isso é uma visão bem ignorante (para não dizer *arrogante*). Mesmo que *seja* verdade que — *teoricamente* — possamos executar qualquer bytecode formatado de modo adequado, ainda que não venha de um compilador Java, na prática, isso é absurdo. Escrever bytecode manualmente é como pintar as fotos de suas férias em vez de fotografá-las com seu celular — mesmo se tratando de arte, a maioria das pessoas prefere dedicar seu tempo a outras coisas. Inclusive, eu agradeceria se você *não* se dirigisse a mim como "parça".

A Máquina Virtual Java

Mas algumas ainda chegam! Posso lançar umas exceções ClassCastExceptions, fora que às vezes as pessoas tentam inserir o tipo errado de coisa em um array declarado para armazenar uma coisa diferente, e…

Aham, tá bom. Mas como fica a *segurança*? Eu faço uma penca de bagulhos de segurança, e você fica, tipo, verificando *ponto e vírgula*? Noooossa, que perigo, heim! Ainda bem que você tá de olho nisso!

Tanto faz. Eu também tenho que fazer as *mesmas coisas*, só pra ter certeza de que ninguém passou escondido por você e mudou o bytecode antes de executar.

Pode contar com isso, *parça*.

O Compilador

Não se esqueça de que o Java é uma linguagem fortemente tipificada. Ou seja, não posso permitir que variáveis armazenem dados do tipo errado. É um recurso de segurança fundamental, e posso barrar a grande maioria das violações antes mesmo de chegarem até você. Além do mais…

Com licença, ainda não terminei. E, sim, *existem* algumas exceções de tipo de dados que aparecem no tempo de execução, mas algumas precisam de permissão para suportar a vinculação dinâmica [dynamic binding]. No tempo de execução, um programa Java pode incluir novos objetos que nem eram *conhecidos* pelo programador, assim, tenho que permitir certa flexibilidade. Minha função é barrar qualquer coisa que nunca — nunca *mesmo* — poderia funcionar no tempo de execução. Em geral, consigo saber quando algo não vai rodar. Por exemplo, se um programador tentar usar sem querer um objeto Button como uma conexão socket, eu detectaria isso, evitando danos no tempo de execução.

Sou a primeira linha de defesa, como dizem. As violações do tipo de dados que falei anteriormente poderiam instaurar o caos em um programa se pudessem se revelar. Também sou eu que paro violações de acesso, como um código tentando invocar um método privado ou alterar um método que, por motivos de segurança, nunca deve ser alterado. Impeço as pessoas de mexerem em códigos que não deveriam ver, códigos que tentam acessar dados críticos de outra classe. Levaria horas, talvez até dias, para descrever a importância do meu trabalho.

Como eu disse antes, se eu não barrasse o que equivale a talvez 99% dos problemas em potencial, você nem funcionaria. E parece que não temos mais tempo, teremos que debater isso em um bate-papo posterior.

O que podemos inserir no método main?

Uma vez dentro do main (ou de *qualquer* método), a diversão começa. Podemos inserir tudo que é usado na maioria das linguagens de programação ***a fim de que o computador faça alguma coisa.***

Seu código pode instruir a JVM a:

1 fazer alguma coisa

Instruções: declarações, atribuições, chamar métodos etc.

```
int x = 3;
String name = "Dirk";
x = x * 17;
System.out.print("x is " + x);
double d = Math.random();
// Este é um comentário
```

2 fazer alguma coisa reiteradamente

Loops: *for* e *while*

```
while (x > 12) {
  x = x - 1;
}

for (int i = 0; i < 10; i = i + 1) {
  System.out.print("i is now " + i);
}
```

3 fazer alguma coisa em determinadas condições

Branching: testes *if/else*

```
if (x == 10) {
  System.out.print("x must be 10");
} else {
  System.out.print("x isn't 10");
}
if ((x < 3) && (name.equals("Dirk"))) {
  System.out.println("Gently");
}
System.out.print("this line runs no matter what");
```

Sintaxe Divertida

★ Cada instrução deve terminar com ponto e vírgula.

```
x = x + 1;
```

★ Um comentário com uma linha começa com duas barras.

```
x = 22;

// Esta linha me preocupa
```

★ A maioria dos espaços em branco não importa.

```
x      =      3  ;
```

★ Variáveis são declaradas com um **nome** e um **tipo** (no Capítulo 3, você aprenderá sobre todos os tipos Java).

```
int weight;

// Tipo: int, name: weight
```

★ Classes e métodos devem ser definidos dentro de um par de chaves.

```
public void go() {
   // Código espetacular
aqui
}
```