Java como plataforma, não como linguagem

É fundamental conhecer com que objetivos a plataforma Java foi inicialmente projetada, para assim entender com profundidade os motivos que a levaram a ser fortemente adotada no lado do servidor.

O Java é uma plataforma completa de desenvolvimento criada pela Sun e que teve seu lançamento público em 1995. Esta plataforma vinha sendo desenvolvida desde 1991 sob o nome *Oak*, liderada por James Gosling. O foco inicial do projeto Oak era dispositivos eletrônicos como os *set-top box*. Por esse motivo, o foco da plataforma foi bastante concentrado na segurança: se a aplicação parasse, não deveria comprometer o restante do dispositivo [13].

Desde a concepção do Java, a ideia de uma plataforma de desenvolvimento e execução sempre esteve presente. O Java idealizado era uma solução que facilitasse o desenvolvimento de aplicativos portáveis utilizando, para isso, uma linguagem de programação simplificada, segura, orientada a objetos, com uma extensa API e um ambiente de execução portável.

Para atingir a questão da portabilidade, usa-se a ideia de uma **máquina virtual** que executa um código de máquina próprio. Essa máquina virtual traduz os comandos para a plataforma específica e, com isso, ganha-se independência de plataforma.

Logo após seu lançamento, o Java fez um enorme sucesso com seus **Applets**, pequenas aplicações embutidas em páginas Web e executadas no navegador do cliente. Nessa época que nasceu o famoso slogan **"Write once, run anywhere"**, pois qualquer navegador em qualquer sistema operacional rodaria a mesma aplicação sem necessidade de alterações com a ajuda da máquina virtual.

Atualmente, o Java está presente em vários ramos da tecnologia e é uma das plataformas mais importantes do mundo. O mercado de Applets acabou não se tornando um grande destaque e o Java firmou sua posição no mercado corporativo (Java EE), com soluções robustas para Web e aplicações distribuídas.

O Java é também uma das principais plataformas de desenvolvimento para dispositivos móveis (**Java ME**), com milhões de celulares, palms, set-top boxes e até nos aparelhos de blu-ray.

No Desktop, não tem sido diferente. Já em 2006, nos EUA, o Swing foi o toolkit gráfico mais usado em novas aplicações, com quase 50% do market share. Além disso, a Sun lançou o JavaFX para atacar o mercado das *Rich Internet Applications*, para combater o Adobe Flex e o Microsoft Silverlight. O JavaFX visa facilitar o desenvolvimento gráfico em Java e é uma grande aposta, apesar de seus concorrentes já estarem há um bom tempo no mercado.

É curioso perceber que o Java nasceu com o intuito de ser uma plataforma muito utilizada em dispositivos móveis, ganhou força através dos Applets, mas o grande mercado que tornou a plataforma disseminada foi o mercado corporativo. Dada sua portabilidade e segurança, a plataforma Java foi a escolha feita por grandes bancos e empresas que precisavam ter facilidade de mudar suas escolhas, evitando o *vendor lock-in*: essas empresas não querem ficar mais dependentes de um único fabricante, um único sistema operacional e assim por diante. Com a plataforma Java, essas empresas adquiriram essa liberdade.

Mas o que é exatamente o Java? **Mais que uma linguagem de programa-**

ção, Java é uma completa plataforma de desenvolvimento e execução. A plataforma Java é composta de três pilares:

- A máquina virtual Java (JVM);
- Um completo conjunto de APIs para as mais variadas tarefas;
- A linguagem Java.

De tudo isso, o menos importante certamente é a linguagem de programação. Hoje, evoluções na linguagem são muito complicadas. As características estáticas e sintáticas da linguagem Java, bem como todo o processo burocrático do JCP

, fazem com que seja bem complicado evoluir. Mas, impulsionada pela ideia da **C**ommon **L**anguage **R**untime (CLR) do .Net, a plataforma Java tem ido cada vez mais para o caminho de ser um ambiente de execução multi-linguagem através da máquina virtual.

O conceito de máquina virtual não foi inventado pelo Java. As *Hardware Virtual Machines*, como VMWare, Parallels ou VirtualBox, sempre foram muito famosas. São usadas para rodar vários sistemas operacionais ao mesmo tempo usando o recurso de virtualização que abstrai o hardware da máquina.

Já a JVM, **Java Virtual Machine** (Máquina Virtual Java) é uma **Application Virtual Machine**, que abstrai não só a camada de hardware, como a comunicação com o Sistema Operacional.

Uma aplicação tradicional em C, por exemplo, é escrita em uma linguagem de alto nível que abstrai as operações de hardware e é compilada para linguagem de máquina. Nesse processo de compilação, é gerado um executável com instruções de máquina específicas para o sistema operacional e o hardware em questão.

Um executável do C não é portável: não podemos rodar um executável Windows no Linux e vice-versa. Mas o problema de portabilidade não se restringe ao executável. As APIs são também específicas de Sistema Operacional. Assim, não basta recompilar para outra plataforma, é preciso reescrever boa parte do código.

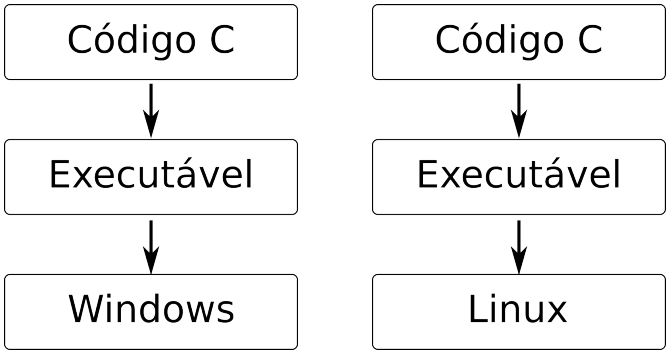

Figura 4.1: Programa em C sendo executado em 2 plataformas diferentes

Ao usar o conceito de máquina virtual, o Java elimina o problema da portabilidade do código executável. Ao invés de se gerar um executável específico, como *"Linux PPC"*

ou *"Windows i386"*, o compilador Java gera um executável para uma máquina genérica, uma máquina **virtual**, não física, a JVM.

A JVM é uma máquina completa que roda em cima da máquina real. Ela possui suas próprias instruções de máquina (assembly) e suas APIs próprias. O papel da máquina virtual é executar as instruções de máquina genéricas no Sistema Operacional e no hardware específico sob o qual está rodando.

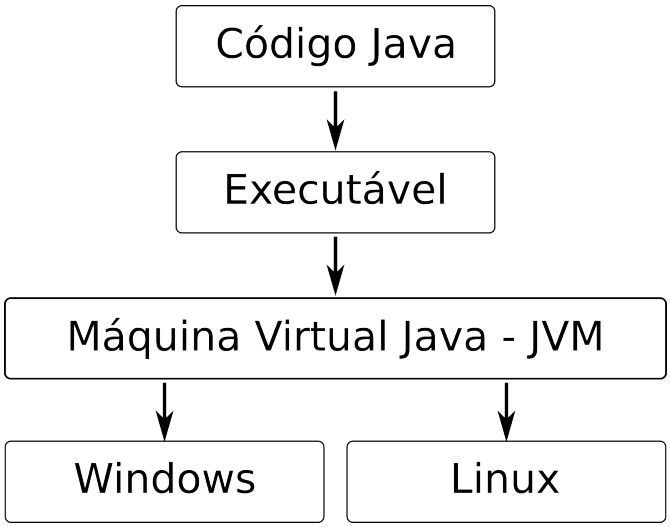

Figura 4.2: Programa em Java sendo executado em 2 plataformas diferentes

No fim das contas, o problema da portabilidade continua existindo, mas o Java puxa essa responsabilidade para a JVM ao invés do desenvolvedor comum. É preciso instalar uma JVM específica para o sistema operacional e o hardware que se vai usar. Mas, feito isso, o desenvolvimento do aplicativo é totalmente portável.

É importante perceber que, por causa desse conceito da JVM, o usuário final que deseja apenas rodar o programa (e não desenvolver) não pode simplesmente executá-lo. Tanto os desenvolvedores quanto os usuários precisam da JVM. Mas os desenvolvedores, além da JVM, precisam do compilador e outras ferramentas.

Por isso que o Java possui duas distribuições diferentes: o **JDK (Java Development Kit)**, é focado no desenvolvedor e traz, além da VM, compilador e outras ferramentas úteis; e o **JRE (Java Runtime Environment)**, que traz apenas o necessário para se executar um aplicativo Java (a VM e as APIs), voltada ao usuário final.

Um ponto fundamental a respeito da JVM é que ela é uma **especificação**, diferente de muitos outros produtos e linguagens. Quando você comprava o Microsoft Visual Basic 6.0, só havia uma opção: o da Microsoft. A JVM, assim como a linguagem, possui especificações muito bem definidas e abertas: a *Java Virtual Machine Specificaton* e a *Java Language Specification* [61, 30]. Em outras palavras, você pode tomar a liberdade de escrever um compilador e uma máquina virtual para o Java. E é isso que acontece: não existe apenas a JVM da Sun (Sun JVM HotSpot), mas também a J9 da IBM, a

JRockit da Oracle/BEA, a Apple JVM, entre outras. Você fica independente até de fabricante, muito importante para empresas grandes que já sofreram muito com a falta de alternativas para tecnologias adotadas antigamente.

Essa variedade de JVMs também dá competitividade ao mercado, pois cada uma das empresas fabricantes tenta melhorar a sua implementação, seja melhorando o JIT compiler, fazendo *tweaks* ou mudando o gerenciamento de memória.

Isso traz também segurança para as empresas que usam a tecnologia. Se algum dia o preço da sua fornecedora estiver muito alto, ou ela não mais atingir os seus requisitos mínimos de performance, é possível trocar de implementação. Temos a garantia de que isso vai funcionar, pois uma JVM, para ganhar esse nome, tem de passar por uma bateria de testes da Sun, garantindo compatibilidade com as especificações.

A plataforma Microsoft .NET também é uma especificação e, por esse motivo, o grupo Mono pode implementar uma versão para o Linux.

Os bytecodes Java

A JVM é então o ponto chave para a portabilidade e a performance da plataforma Java. Como vimos, ela executa instruções genéricas compiladas a partir do nosso código, traduzindo-as para as instruções específicas do sistema operacional e do hardware utilizados.

Essas instruções de máquina virtual são os famosos **bytecodes**. São equivalentes aos *mnemônicos* (comandos) do assembly, com a diferença que seu alvo é uma máquina virtual. O nome *bytecode* vem do fato que cada *opcode* (cada instrução) tem um byte de tamanho e, portanto, a JVM tem a capacidade de rodar até 256 bytecodes diferentes (embora o Java 6 tenha apenas 204). Isso faz da JVM uma máquina teoricamente simples de se implementar e de rodar até em dispositivos com pouca capacidade.

Podemos inclusive visualizar esses bytecodes. O download do JDK traz uma ferramenta, o **javap**, capaz de nos mostrar o bytecode contido em um arquivo *.class* compilado de maneira mais legível para nós. O bytecode será mostrado através dos nomes das instruções e não através de bytes puros.

O código a seguir ilustra um exemplo de classe Java:

```
public class Ano {
    private final int valor;
    public Ano(int valor) {
        this.valor = valor;
    }
    public int getValor() {
        return valor;
    }
}
```

E a execução da linha do comando `javap -c Ano` exibe o conjunto de instruções que formam a classe `Ano`:

```
public class Ano extends java.lang.Object{
public Ano(int);
```

```
  Code:
   0:    aload_0
   1:    invokespecial    #10; //Method java/lang/Object."<init>":()V
   4:    aload_0
   5:    iload_1
   6:    putfield    #13; //Field valor:I
   9:    return

public int getValor();
  Code:
   0:    aload_0
   1:    getfield    #13; //Field valor:I
   4:    ireturn

}
```

Outro exemplo da força do bytecode para a plataforma Java está em exemplos de outras linguagens, como Scala, que é compilada para bytecode Java. No fundo, para a JVM, o que interessa são os bytecodes e não de qual linguagem eles foram compilados. Em Scala, por exemplo, teríamos o seguinte código fonte, que possui a mesma funcionalidade que o código Java mostrado anteriormente:

```
class Ano(private val valor:Int) {
  def getValor = valor
}
```

Ao compilarmos a classe Scala acima com o *scalac* - o compilador Scala - os bytecodes gerados serão os mesmos que vimos na classe Java. Podemos observar isso rodando o comando `javap -c Ano` novamente. Ainda veremos mais sobre outras linguagens rodando em cima da JVM neste capítulo.

Além disso, muitos frameworks e servidores de aplicação geram bytecode durante a execução da aplicação (dinamicamente) e veremos com detalhes no capítulo 5.

A JVM é então um poderoso executor de bytecodes, sem interessar de onde esses mesmos vêm, independente de onde está rodando. É por esse motivo que James Gosling já afirmou que a linguagem Java não é o mais importante da plataforma, e sim a JVM.