Para utilizar bem uma linguagem de programação, é importante saber como ela opera, como é compilada, como seus módulos são carregados e executados. Conhecer um pouco de sua história também ajuda a entender as decisões arquitetônicas que nortearam sua evolução. Vamos apresentar uma visão de "dentro para fora" do Java neste capítulo. Entretanto tenho que concordar que muitas vezes este conhecimento não tem o charme e a praticidade da atividade de programação em si, e, portanto, você pode decidir ignorar este capítulo. Para aqueles que irão direto ao assunto, boa sorte e nos vemos alguns capítulos mais adiante. Para os que me seguirão nesta jornada exploratória da base da tecnologia, paciência por favor.

A linguagem Java foi criada em 1995, mas se popularizou em 1997. O nome original da linguagem era OAK, atribuído por um programador muito criativo, que provavelmente tinha na frente da janela do escritório na Sun um carvalho gigante (OAK = Carvalho em Inglês). Graças ao time de marketing da Sun, o nome foi alterado para Java continuando até os dias atuais. O pai da tecnologia, se assim podemos chamá-lo, foi James Gosling e seu time na Sun Microsystems.

Talvez não seja tão conhecido da maioria das pessoas, mas a linguagem foi criada com o primário objetivo de obter uma camada uniforme de hardware sobre objetos comuns do

quotidiano, como chaleiras, batedeiras, geladeiras, bem como objetos industriais como máquinas de CNC e controladores industriais. Já se previa naquela época que em um momento no futuro todos os dispositivos de uso quotidiano teriam microprocessadores de variadas características, e seria muito "esperto" ter uma linguagem genérica sobre todos eles (já falarei mais sobre isto adiante). É muito curioso que hoje o movimento que chamamos de IOT (Internet Of Things) foi o principal motivador da criação da plataforma Java.

A história mostrou que o mercado ainda não estava preparado para esta abordagem, e em torno de 1996/1997, com o "boom" da internet, a linguagem Java foi adotada como mecanismo de inteligência dos browsers. Naquela época, os browsers apresentavam apenas conteúdo estático, e a única forma de criar algum dinamismo em páginas era através de CGIs. Os CGIs são programas normais, criados em qualquer linguagem de programação, mas que "escrevem" o HTML na saída padrão. Os servidores Web capturam esta escrita e enviam ao browser. Não havia na época nenhum mecanismo consolidado para "dar vida" às páginas, como tomar uma atitude quando o mouse passasse sobre uma área na tela, criar animações e coisas deste tipo (ou ao menos não eram padronizados e, na época).

O Java, criado desde sua concepção para ser agnóstico

em relação ao processador ou sistema operacional, se encaixava como uma luva para ser a tecnologia que daria vida aos browsers, que já proliferava uma diversidade de sistemas operacionais na época. Esta tecnologia se chamava Applets. Alguns leitores talvez se lembrem que logo no começo da tecnologia, o principal exemplo que tínhamos era o Duke (mascote do Java), rolando pela tela dos vários browsers. Esta foi uma época de dominância do Netscape, o browser mais famoso de todos os tempos.

O princípio de execução era instalar um interpretador na máquina local com o browser (na época o nome era Java Plugin), e dentro do HTML se colocava uma "tag" que carregava os códigos Java da rede, em uma área isolada de ataques externos e restrita, ou a chamada "caixa de areia" (sandbox). Nesta área de execução o "Applet" tinha acesso limitado a arquivos locais e a rede também. O objetivo sempre foi garantir a segurança contra invasões.

Nos browsers o Java também não teve a receptividade esperada, creio que principalmente pelas limitações de acesso descritas anteriormente, e hoje a tecnologia chamada Applet praticamente desapareceu ou está em desuso.

Entretanto, após estes dois fracassos, Java teve seu reconhecimento e consolidação no ambiente corporativo,

sendo utilizado para construção de servidores de aplicações e aplicações de infraestrutura. Havia uma brincadeira na época, claramente uma afronta à Microsoft, onde um pseudogrupo foi chamado de NOISE na internet (Netscape, Oracle, IBM, SUN and Everybody else). Isto porque embora Java tenha sido adotado massivamente por estas empresas e pelo mercado, estava fora do foco da Microsoft.

Desde o início, a plataforma Java foi criada para permitir a execução em diversos processadores, como dissemos anteriormente. Ou seja, uma grande sacada do time de James Gosling foi a criação de um assembly, ou conjunto de instruções de máquina para um processador hipotético, e um assembler, um conjunto de instruções binárias que poderiam ser executadas em qualquer outro processador através de um processo de interpretação (assembler também é o nome dado ao compilador assembly). Isto é o que conhecemos hoje como o bytecode Java. A máquina virtual Java (JVM) nada mais é do que um programa que interpreta este assembly para um processador nativo, e este interpretador deve ser criado para cada sistema operacional. Os bytecodes por outro lado são blocos comuns de código que podem executar em vários sistemas operacionais diferentes.

Há um consenso de que interpretar um conjunto de instruções em assembler tende a ser uma atividade mais efetiva

do que interpretar uma linguagem textual, como JavaScript.

Em teoria, basta de uma forma simplória converter cada instrução assembly do Java em um conjunto de instruções nativas do microprocessador hospedeiro. Isto é conhecido como JIT (Just In Time ou compilação em tempo de execução). Praticamente todas as JVMs existentes contém algum tipo de JIT para acelerar o processo de execução.

Além do assembly e do assembler, foi criada uma linguagem de programação com a sintaxe baseada no C++, e quando programamos em Java na verdade escrevemos um dialeto de C++ que é convertido para o bytecode. Nos dias atuais, existem outras linguagens de programação que compilam para o mesmo bytecode Java, como Groovy, Jython e Scala. Também foi criado um conjunto de APIs úteis para o programador, hoje parte do JSE (Java Standard Edition). Sobre o JSE foi criada uma camada corporativa chamada de JEE (Java Enterprise Edition).

Na prática, depois de todas estas evoluções, Java não é apenas uma linguagem, mas uma plataforma composta de um assembler ou set de instruções, uma linguagem de alto nível parecida com C++, um compilador, um interpretador e um conjunto de APIs para uso comum, além de um conjunto de APIs para o mercado corporativo. Por isto chamamos Java de plataforma, e não apenas uma linguagem de programação.

Mas como um programa Java executa? Bem, na prática escrevemos os códigos em um dialeto de C++, compilamos para o bytecode e executamos o programa com a JVM. O famoso programa "Hello Word", hoje praticamente o início de qualquer aprendizagem, e que foi apresentado pela primeira vez no livro de Bjarne Stroustroup (o criador do C++), seria criado assim em Java :

```
package meulivro;
public class MainClass {
        public static void main(String[] args) {
                System.out.println("Hello World !!");
        }
}
```

Bom, o código não faz nada além de imprimir uma mensagem de boas vindas na tela. Sobre este código, algumas considerações: Java é uma linguagem orientada para objetos, ou seja, os blocos construtivos são classes que são instanciadas e geram objetos (falaremos mais sobre isto em outro capítulo).

Mesmo o ponto de entrada do programa é uma classe, mas tem um método especial **main**, que pode ser executado quando ainda não temos objetos. Veja, tudo em Java são objetos, e objetos criam objetos. Mas se não temos um objeto ainda, é necessário algum mecanismo que permita criar um primeiro objeto, para que a partir dele todo o universo de objetos seja criado ao redor.

Este é o significado do **static** no código, ou seja, ele pode ser executado mesmo sem uma instância da classe (ou objeto) MainClass. Depois de compilado, ele vira um conjunto de bytecodes, ilustrado aqui de uma forma mais visível:

```
public class meulivro.MainClass {
  public meulivro.MainClass();
    Code:
      0: aload_0
      1: invokespecial #1
      4: return
  public static void main(java.lang.String[]);
    Code:
      0: return
}
```

Por exemplo, **aload_0** é uma instrução assembler que diz para pegar a primeira variável declarada dentro da classe e coloca-la na pilha. Felizmente para ser um bom programador Java você não precisa conhecer cada um de seus bytecodes.

Sem se preocupar com os detalhes, um conjunto de bytecodes é gerado para cada classe compilada e salvo em um arquivo com o mesmo nome do código fonte, mas com a extensão .class.

Por exemplo, o código fonte **MainClass.java** irá gerar um código compilado **MainClass.class**.

Ainda há mais um detalhe aqui: existe o conceito de package, que é uma forma de organizar classes relacionadas, e evitar colisões de nomes. Nas APIs do JSE temos **packages** como **java.io**, **java.net**, **java.lang**, etc. Cada um com sua finalidade específica.

Em termos de como isto fica fisicamente nos diretórios, a classe precisa estar dentro de diretórios para cada palavra contida no nome do package, e separada por pontos.

Por exemplo, uma classe no pacote "**com.ibm.livro**" com o nome **MeuLivro** deve obrigatoriamente estar no diretório :

```
<diretório de busca>
        com
            ibm
                livro
                    MeuLivro.class
```

Podem existir diretórios anteriores, mas a partir do **"diretório de busca"** é mandatório que a classe esteja na mesma estrutura. Se o diretório de busca for por exemplo **c:/classes**, a estrutura completa deve ser

```
c:/classes/com/ibm/livro/MeuLivro.class
```

O diretório de busca recebe um nome especial, "**classpath**". Como existe o "**path**" dos sistemas operacionais, que é a partir de onde se encontram os executáveis, os classpaths são a partir de onde se localizam as classes. Quando a máquina virtual inicia a execução, ela tem acesso a um conjunto de classpaths, que são os caminhos ou diretórios a partir dos quais se devem iniciar as buscas pelas classes. Vocês verão mais adiante que estes detalhes são de extrema importância, e até arriscaria dizer que a a maioria dos problemas com a linguagem Java estão relacionados a este processo de carga de classes.

Ocorre que nem todo sistema operacional tem uma estrutura de diretórios como descrito acima. Lembre-se de que Java foi criado para executar em uma torradeira. A solução fornecida pelos arquitetos da linguagem foi colocar toda esta

estrutura de diretórios dentro de um arquivo que permita representar esta hierarquia, mesmo que o sistema operacional hospedeiro não tenha diretórios. O formato padrão e open source para isto é o TAR, adotado inicialmente pelos sistemas Unix.

É bem verdade que hoje o Java também suporta formato ZIP e até tem um formato próprio chamado JAR. Dentro de um JAR existe uma estrutura de diretórios que representam o caminho dos pacotes, ou seja, a estrutura de diretórios, como explicado anteriormente.

No nosso exemplo anterior, poderíamos ter dentro de um arquivo pacote.jar, algo como:

```
pacote.jar
        com
            ibm
                livro
                    MeuLivro.class
```

E no classpath teríamos agora algo como

```
c:/classes/pacote.jar
```

Ou seja, temos dois modelos de armazenamento de classes em Java, o que clamamos comumente de **"loosed classes"** (classes soltas em diretórios do sistema operacional), e outro conhecido como JARs ou pacotes (grupos de classes dentro de arquivos com a extensão JAR).

Pode-se ter um classpath com vários destes valores encadeados como por exemplo:

```
c:/classes/pacote.jar; c:/classes
```

O separador varia por sistema operacional, sendo que no Windows é ; (ponto e vírgula) e nos Unix e Mac : (dois pontos).

O classpath é uma variável de ambiente, e deve estar disponível para que a JVM (interpretador de bytecodes) funcione. Todo o cenário está mais ou menos delineado. Executamos uma classe, e ela localiza outras classes através de uma busca no classpath por uma estrutura de diretórios que corresponda ao package somado ao nome da Classe. Ainda temos uma outra peça aqui.

Lembrando novamente o fato de que Java foi criado para ser executado em uma batedeira, e posteriormente em um browser, mas todos eles conectados em uma rede. As classes podem vir pela internet, estar em diretórios ou mesmo serem carregadas a partir de uma base de dados relacional. Como podemos ter esta diversidade de opções? Existe um componente fundamental em Java chamado ClassLoader. O ClassLoader é uma classe Java (!) responsável por carregar outras classes, de qualquer lugar. Se estivermos rodando dentro de um browser, o ClassLoader carrega as classes da rede. Se estivermos em uma batedeira, o ClassLoader as carrega a partir de um firmware ou alguma área de memória do eletrodoméstico. Basta criar ClassLoaders específicos para cada finalidade que se deseja executar. Esta classe ainda tem uma inteligência, que funciona da seguinte forma: uma vez carregada a classe, ela fica armazenada no ClassLoader como um cache, e na segunda vez em que for solicitada o valor cacheado é retornado a quem solicitou.

Para entendermos este processo, vamos aplica-lo ao nosso HelloWord. A execução se inicia com a chamada da linha de comando:

```
java meulivro.MainClass
```

O primeiro passo é pedir ao Classloader para carregar MainClass. Este então deve localizar algum diretório no classpath com um diretório **meulivro** (o package). Ele pode estar no formato loosed ou JAR, não importa. Uma vez localizado, tenta-se encontrar um arquivo **MainClass.class** (precisa ser exatamente este nome com este case) e o carrega na memória.

Quem faz isto é o ClassLoader fundamental (um classloader onipresente, para simplificar). Quando a classe está sendo resolvida, ou seja, seu bytecode verificado, dependências resolvidas e espaço de memória alocado, o ClassLoader encontra a linha

```
System.out.println("Hello World !!");
```

e inicia uma nova busca pelo diretório java/lang, que é onde esta a classe System se encontra (este pacote é tão fundamental que não precisa ser digitado). Quando encontra um arquivo **System.class** o carrega e continua o processo. Como existe uma String (**java.lang.String**) , representada pelo **"Hello World !!"**, a JVM faz o mesmo processo até encontrar e cachear a classe **String**. Este processo continua, de uma forma dinâmica, ou seja, as classes são carregadas quando são necessárias, durante o processo de execução do programa.

Este processo é diferente de programas nativos no Windows ou Linux, por exemplo. Neste caso, tudo se resolve antes da execução (bem, de uma forma simplificada, pois pode-se ter carga dinâmica em Windows ou Linux também). Mas no Java as dependências são resolvidas sob demanda. Por isto as vezes ela é chamada de uma linguagem dinâmica.

É fácil perceber que o início de execução de um programa Java é como um "Big Bang", com um turbilhão de atividades sendo executadas, onde centenas e talvez milhares de classes serão carregadas em um tempo muito curto, antes de ficar cacheadas. Por esta razão, um programa Java é melhor quanto mais tempo ele estiver executando. No início da execução ele gasta um tempo imenso para carregar classes e mantê-las na memória. Após algum tempo em execução ele estabiliza na carga destas classes e passa a utilizar as que já estão na memória. Um programa em Batch é uma péssima utilização da linguagem, e um servidor de aplicações como o WebSphere, por exemplo, é um excelente exemplo de utilização da tecnologia.

Lá atrás eu comentei sobre o "ClassLoader fundamental", o que sugere que temos mais de um deles a cada momento. Na realidade, em Java temos hoje vários ClassLoaders para a JVM. A razão disto é que cada dispositivo onde Java executa poderia ter características diferentes, e diversas opções de carga poderiam ser necessárias. Uma parte das classes poderia ser carregada localmente e uma outra parte a partir da rede, por exemplo. Em uma JVM típica, temos hoje os seguintes ClassLoaders:

1) **Bootstrap ClassLoader** – Carrega as classes

principais do Java e que são necessárias já no início da execução. Normalmente elas estão em um arquivo rt.jar dentro de algum diretório da JVM (normalmente jre/lib)

2) **Extension ClassLoader** – Carrega as classes de extensão, normalmente definidas pela variável java.ext.dir que é passada para a JVM
3) **CLASSPATH ClassLoader** – Carrega as classes definidas no Classpath como explicado anteriormente

Elas seguem uma hierarquia, e no caso da JVM se iniciam da mais fundamental para baixo. No processo de carga de classes da JVM, primeiro o Bootstrap é acionado para carregar a classe. Se ele não encontrar a classe, passa o controle para Extension, e assim até chegar ao Classloader CLASSPATH. Ele sempre segue esta ordem e nunca retorna um nível na busca.

Quando os servidores de aplicação foram criados, eles criaram outros níveis com mais ClassLoaders, e hoje temos carregadores para cada bloco de um servidor de aplicação (EARs, WARs), e ainda podemos modificar a ordem de busca, começando de cima para baixo (parentFirst) ou debaixo para cima (parentLast).

Toda esta diversidade foi implementada em nome da flexibilidade, mas também costuma ser uma fonte de problemas, pois caso uma classe não seja encontrada, provavelmente uma Exceção, que também é uma classe em Java (!), será lançada e pode comprometer a execução do programa. Também existe o problema de colisão de classes, que é um programa tentar carregar uma classe, mas já existe uma outra classe com o mesmo nome carregada. Você pode

argumentar que uma classe teoricamente nunca deve ter o mesmo nome de outra, mas lembre-se de que existem versões.

Uma classe de um módulo pode estar utilizando a release 1.0 de uma API, e outro módulo (carregado pelo mesmo ClassLoader) pode estar tentando utilizar a 2.0. Como a versão do módulo não é parte do nome da classe, os carregadores Java não têm como distinguir classes de versões diferentes. Neste caso, ele irá utilizar a primeira carregada, e certamente algum problema de execução irá acontecer no futuro, caso a classe de versão errada seja carregada.

Para resolver estes tipos de problemas foi criado o padrão OSGI, que é adotado por soluções como Liberty profile, Websphere e Eclipse. No OSGI, os ClassLoaders são isolados por módulo, e um ClassLoader não interfere na carga dos outros. Ainda existe um descritor que indica qual versão do pacote ou API é necessária para execução, e esta validação é feita na carga do módulo. Por isto, uso de versões erradas de classes (as colisões) são mais difíceis de acontecer no OSGI, o que gera maior estabilidade na execução de programa Java.

Uma boa prática para evitar problemas com cargas de classes é validar cada uma das bibliotecas ou JARs sendo utilizados, e garantir que não há conflitos entre classes, ou diferentes versões sendo utilizadas por um mesmo empacotamento. Isto pode ser muito complexo, já que em um servidor de aplicações típico podemos ter centenas de bibliotecas e milhares de classes.

Existem ferramentas que auxiliam a encontrar estes conflitos e a corrigir problemas de execução em Java.

Pois bem, como isto me afeta? Veja, existem ferramentas

como Maven ou outras ferramentas de automação que fazem o trabalho via força bruta, de identificar tudo que um aplicativo precisa em termos de JEE e coloca-los como dependências de bibliotecas na aplicação. Mas como é força bruta, não muita há inteligência no processo (sem crítica ao Maven). Cabe ao programador ou arquiteto verificar se as bibliotecas incluídas na aplicação não estão já instaladas no servidor JEE, verificar conflitos entre classes ou bibliotecas, e selecionar a melhor estratégia de carga de classes (parentFirst, parentLast). Esta é uma área nebulosa da linguagem, porque normalmente o programador não se preocupa com ela, e os administradores muitas vezes a deixam passar despercebido também.

Como um Micro Serviço é um bloco muito estanque que executa uma atividade específica, este tipo de problema tende a ser minimizado nesta arquitetura.

Pois bem, temos um entendimento básico do funcionamento de um programa Java, mas precisamos entender um pouco de orientação para objetos para programar em Java, o que veremos a seguir.