O Que São?

Explicando de uma forma mais técnica, uma máquina virtual, também chamada de virtual machine, pode ser definida como um sistema operacional (normalmente chamado de convidado, ou guest), que é executado sobre outro sistema operacional (chamado de sistema host), de forma isolada. Ou seja, uma máquina virtual é um software no qual um sistema operacional pode ser instalado e executado dentro do mesmo. Assim surgindo a clássica expressão: “uma máquina virtual é um computador dentro do outro”.

Para ter uma VM, é necessário ter um Hypervisor (hipervisor em português), uma vez que o hipervisor é encarregado de criar e executar a máquina virtual. Dessa forma, o Hypervisor é responsável por fazer a ligação entre a VM e o sistema host, já que ele separa do hardware os recursos que serão utilizados pela maquina virtual, realocando-os entre os sistemas guest. Esse processo faz com que seja criado um ambiente virtual que atua separadamente do sistema host, ou seja, o sistema guest não interfere no sistema host e o mesmo para o inverso. A criação da máquina virtual é denominada virtualização.

Pelo fato das VM’s utilizarem parte do hardware que as foi “emprestado”, elas não são capazes de executarem programas que demandam mais hardware, como jogos ou softwares 3D.

Para que Servem?

Uma máquina virtual pode ter inúmeras funções dentro de um ambiente de trabalho, uma vez que por meio dela é possível realizar inúmeros testes de compatibilidade, já que caso você esteja criando um software é necessário verificar a compatibilidade dele com sistemas operacionais diferentes ou em versões anteriores de um mesmo sistema operacional.

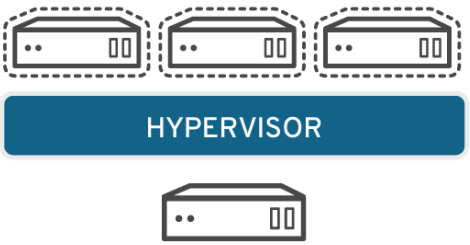
Além disso, muitas vezes o usuário utiliza em seu computador o sistema operacional Linux e precisa utilizar algum software do pacote office, para isso, ele pode instalar uma VM de sistema com o Windows, como o virtual box por exemplo.

As VM’s possuem outra função relacionadas a segurança, pois quando existe um arquivo que não é seguro, e o usuário possui o interesse de abrir esse arquivo, para que não ocorra nenhum dano ao computador, é recomendado que o usuário utilize de uma VM, já que elas atuam de maneira isolada ao sistema host, não podendo trazer danos a ele.

TIPOS DE MÁQUINAS VIRTUAIS (VM):

Existem 3 tipos de Máquinas Virtuais:

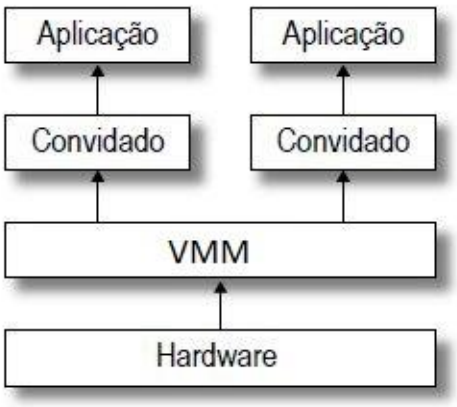
1. Máquinas Virtuais de Sistema:
   - Esse tipo de VM é adquirido por meio de um Hypervisor responsável por fazer o gerenciamento do acesso da memória e dados do Sistema Operacional Host para o Sistema Operacional Ghost de forma totalmente isolada.



Esquema de Máquinas Virtuais de Sistema

   - Características e objetivos das VM de Sistema:
     - O objetivo principal é executar um Sistema Operacional diferente do SO real do computador (host).
     - Por meio dos Hypervisor é possível mudar o local e o tamanho da memória utilizada para executar o Sistema Operacional ghost.
     - Um dos principais motivos para se utilizar esse tipo de VM é testar um certo Sistema Operacional ou usa-los para testar aplicativos não muito confiáveis.
     - As desvantagens desse tipo de VM é que o processamento das atividades executadas no Hypervisor são mais lentas. Além disso, é necessário ter um Hardware poderoso.

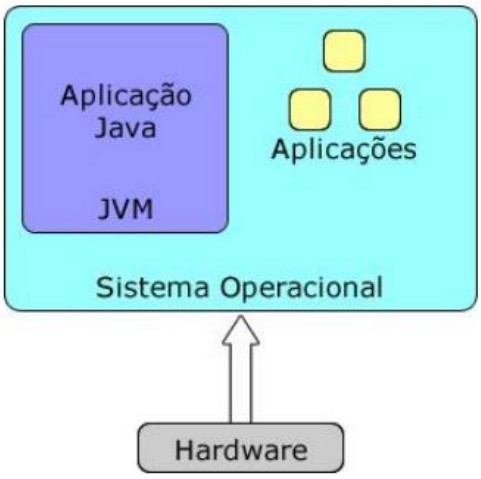
2. Máquinas Virtuais de Sistema Operacional:
   - Esse tipo de VM é adquirido por meio de um Hypervisor responsável por disponibilizar diversos espaços/ambientes isolados para diferentes usuários. Esses espaços são chamados de Zonas Virtuais.



Esquema de Máquinas Virtuais de Sistema Operacional

- Características e objetivos das VM de Sistema Operacional:
  - Esse tipo de virtualização não necessita de um Hardware muito potente, pois não exige muito recurso.
  - Esse tipo de VM é muito utilizado por empresas como uma alternativa de diminuir os custos, uma vez que permite fornecer vários ambientes digitais com apenas um Hardware.
  - Também é utilizada quando existem diversas máquinas físicas sendo utilizadas para rodar aplicações servidoras, como servidores de e-mail, web, entre outros, e a capacidade ociosa de processamento dessas máquinas é alta.

3. Máquinas Virtuais de Aplicação:
   - Essas VM servem especificamente para a execução de um software qualquer sobre um Sistema Operacional, por exemplo, um programa é escrito em uma linguagem de programação que será interpretada pela máquina virtual, que será responsável por ler o código binário do programa e executá-lo utilizando as instruções da máquina real.



Esquema de Máquina Virtual de Aplicação

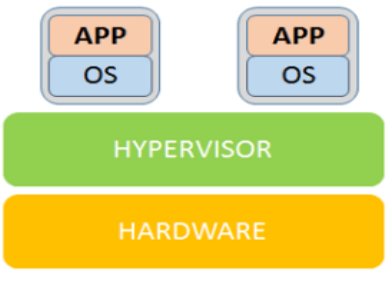
- Características e objetivos da Máquina Virtual de Aplicação:
    - As aplicações feitas utilizando VMs de Aplicação não precisam ser modificadas caso o Sistema Operacional da máquina real mude.
    - Permite padronizar aplicações.
    - O programa escrito para a máquina virtual poderá ser executado em qualquer sistema operacional, basta que exista uma máquina virtual desenvolvida para esse SO.
    - A execução de programas é mais lenta, pois as instruções precisam ser interpretadas pela máquina virtual.

TIPOS DE HYPERVISOR:

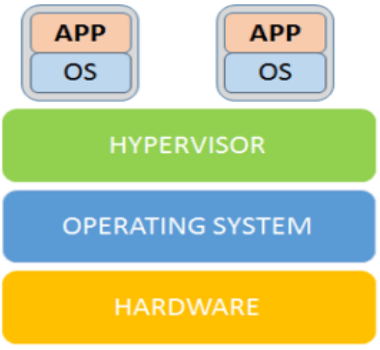
TIPO 1. Um hipervisor tipo 1 (nativo) é executado diretamente no hardware do host para gerenciar sistemas operacionais guest. Ele ocupa o lugar de um sistema operacional host, e os recursos da máquina virtual são programados diretamente no hardware pelo hipervisor. Este tipo é mais comum em datacenters corporativos ou outros ambientes baseados em servidor. Ou seja, Máquinas Virtuais de Sistema Operacional e/ou Máquinas Virtuais de Aplicação utilizam esse tipo de hipervisor para fazer a ligação com o hardware real. São exemplos de hipervisor tipo 1: VMware: ESXI/ESX, Microsoft Hyper-V, etc.

/

TIPO 2. O hipervisor tipo 2 (hosted) é executado em um sistema operacional convencional como uma camada de software ou aplicação. Ele funciona abstraindo sistemas operacionais guest do sistema operacional host. Os recursos de máquina virtual são operados em um sistema operacional host, que é executado no hardware. Esse tipo de hipervisor é mais adequado para usuários individuais que desejam executar vários sistemas operacionais em um computador pessoal a fim de testar um Sistema Operacional novo, por exemplo. São exemplos de hipervisor tipo 2: Oracle VirtualBox, VMware Workstation, etc.



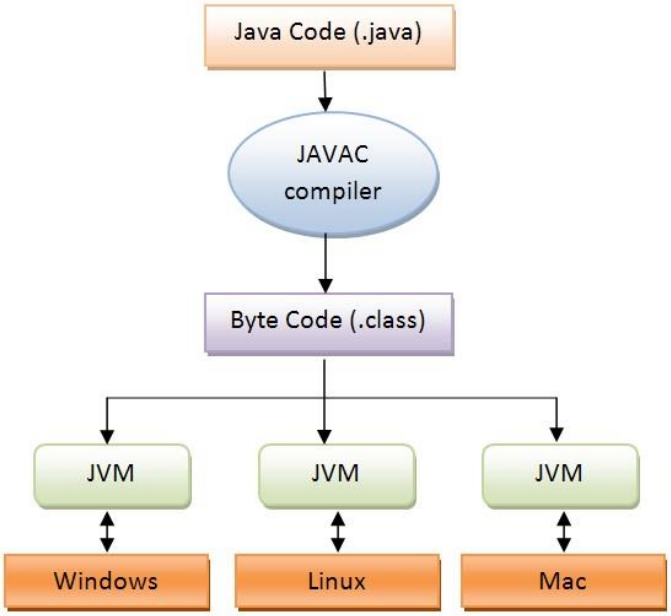
Type 1 Hypervisor



Type 2 Hypervisor

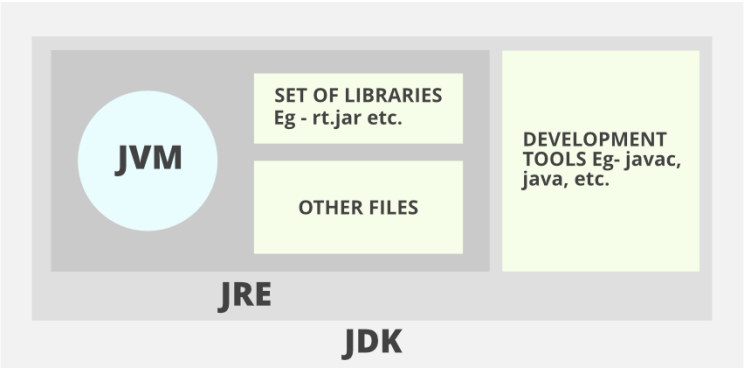
Arquitetura do JVM

O Java Virtual Machine (JVM) é a ferramenta responsável por carregar, verificar e executar programas Java, permitindo carregar os mesmos arquivos em qualquer sistema operacional, sem necessidade de adaptação desses arquivos (apenas da própria JVM, já que existe uma específica para cada sistema operacional). Ele consegue fazer isso convertendo o bytecode Java para o código de máquina necessário no SO que está executando-o.



Arquivos .java são compilados e convertidos em byte code (.class), para daí serem interpretados pela JVM

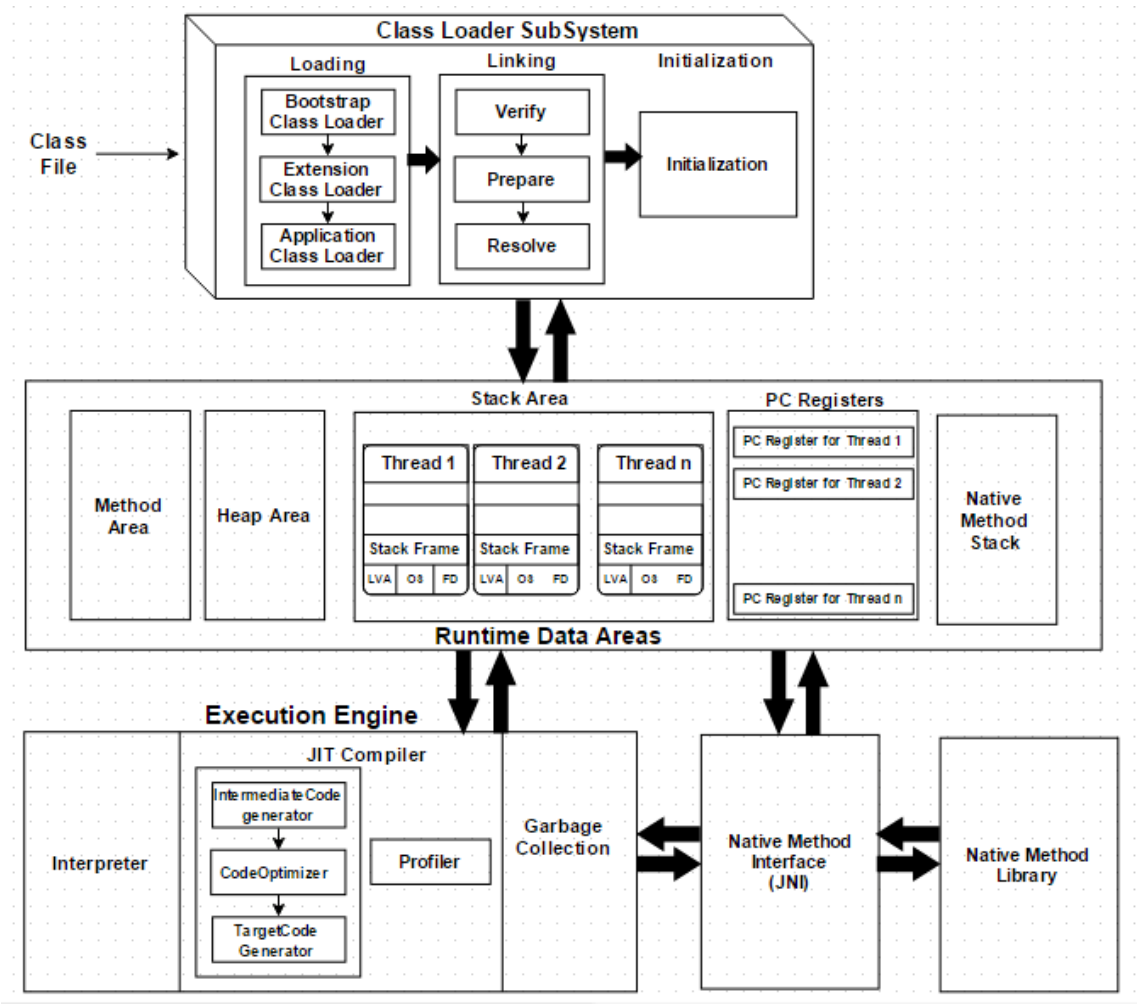
O JVM não funciona de maneira independente. Para usá-lo, deve-se baixar o JRE (Java Runtime Enviroment), que consiste em um pacote de bibliotecas e APIs da linguagem Java. Além dele, pode-se utilizar o JDK (Java Development Kit), que é o kit de desenvolvedor, um programa que engloba ambos os itens citados anteriormente (assim como o **javac**, principal compilador da linguagem Java) e tem como finalidade ser a plataforma base para um programador que deseja criar softwares em Java. A seguir uma imagem explicita essa relação entre esses três componentes:



Relação entre JDK, JRE e JVM

O JVM pode ser subdividido em 3 principais segmentos:

1. ClassLoader Subsystem
2. Runtime Data Area
3. Execution Engine



1. ClassLoader Subsystem

O Classloader é responsável por carregar, conectar e inicializar um arquivo .class, também conhecido como carregamento dinâmico de classes.

1.1 Loading

Os arquivos são carregados nesse componente. BootStrap ClassLoader, Extension ClassLoader, e Application ClassLoader são os três que auxiliam nessa tarefa.

1. **BootStrap ClassLoader** – Responsável por carregar arquivos do bootstrap classpath, que nada mais são que os arquivos rt.jar. É uma tarefa de alta prioridade entre as 3 citadas.
2. **Extension ClassLoader** – Responsável por carregar arquivos que estão dentro da pasta de extensão (jre/lib).

3. **Application ClassLoader** – Carrega os arquivos de classe do classpath. Por padrão, o caminho da classe é definido como o diretório atual. Pode-se alterar o caminho do arquivo usando a opção *-cp* ou *-classpath*. Também é conhecido como carregador de arquivos de aplicativos.

1.2 Linking

1. **Verify** – Verifica se o bytecode executado está correto ou não. Caso não esteja, mostrará mensagem de erro e interromperá a operação.
2. **Prepare** – Todas as variáveis estáticas de memória são alocadas e atribuídas a valores padrões.
3. **Resolve** – All symbolic memory references are replaced with the original references from Method Area.

1.3 Initialization

Fase final do ClassLoading. Aqui, as variáveis estáticas voltam para seus valores originais, e o bloco estático é executado.

2. Runtime Data Area (ou JVM Memory)

A Runtime Data Area pode ser dividida entre 5 principais componentes:

1. **Method Area** – Nessa etapa, toda informação de classe (arquivo .class) como nome da classe, métodos e variáveis etc. é armazenada, incluindo as variáveis estáticas. Existe um único method area para cada JVM, e essa é uma etapa compartilhada.
2. **Heap Area** – as informações de todos os objetos são armazenadas na área de heap. Há também uma área de heap por JVM. Também é um recurso compartilhado.
3. **Stack Area** – para cada thread, a JVM cria uma pilha de tempo de execução que é armazenada aqui. Cada bloco dessa pilha é chamado de registro de ativação / quadro de pilha, que armazena chamadas de métodos. Todas as variáveis locais desse método são armazenadas em seus quadros correspondentes. Após o término de um encadeamento, sua pilha de tempo de execução será destruída pela JVM. Não é um recurso compartilhado.
4. **PC Registers** – Armazena o endereço da instrução de execução atual de uma thread. Cada thread possui registros de PC separados.
5. **Native Method stacks** – para cada thread, uma pilha nativa separada é criada. Ele armazena informações do método nativo.

3. Execution Engine

O mecanismo de execução executa o **.class** (bytecode). Ele lê o código de byte linha por linha, usa dados e informações presentes em várias áreas da memória e executa instruções. Pode ser classificado em três partes:

- **Interpreter** : interpreta o bytecode linha por linha e depois executa. A desvantagem aqui é que, quando um método é executado várias vezes, sempre é necessária uma interpretação.

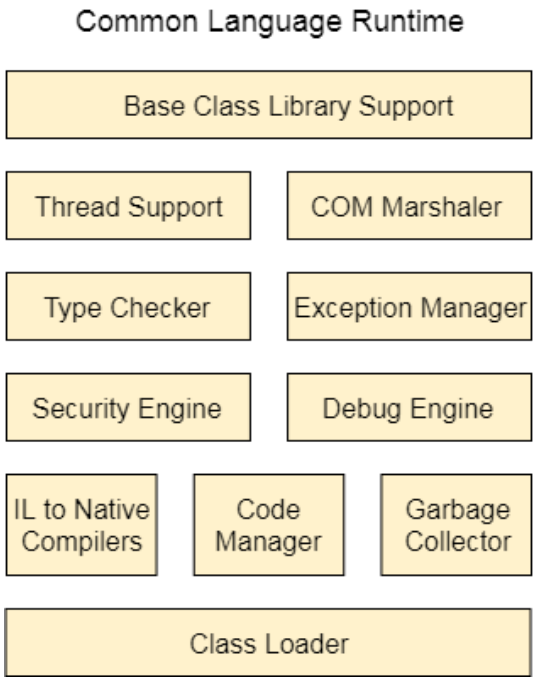
- **Compilador Just-In-Time (JIT)** : É usado para aumentar a eficiência de um interpretador. Ele compila todo o bytecode e o altera para o código nativo, de modo que sempre que o interpretador vê chamadas de método repetidas, o JIT fornece código nativo direto para essa parte, de modo que a reinterpretação não é necessária e, portanto, a eficiência é aprimorada.
- **Coletor de lixo** : destrói objetos não referenciados.

Java Native Interface (JNI):
É uma interface que interage com as Bibliotecas de Método Nativo e fornece as bibliotecas nativas (C, C ++) necessárias para a execução. Ele permite que a JVM execute bibliotecas C / C ++ e seja executado por bibliotecas C / C ++ que podem ser específicas do hardware.

Native Method Libraries:
É uma coleção de bibliotecas nativas (C, C ++) que são exigidas pelo mecanismo de execução.

Arquitetura do CLR (.NET)

A máquina virtual responsável pela execução de programas na plataforma .NET Framework é o Common Language Runtime (CLR), suportando várias linguagens de programação diferentes como C#, F#, Visual Basic .NET, etc.. O Common Language Runtime implementa o VES (Virtual Execution System), que é um sistema de tempo de execução que fornece um ambiente de execução de código gerenciado. O VES é definido na implementação da CLI (Common Language Infrastructure) da Microsoft. O seguinte diagrama representa a sua arquitetura:



- **Base Class Library Support :** O Common Language Runtime fornece suporte para a biblioteca de classes básicas. O BCL contém várias bibliotecas que fornecem vários recursos, como *coleções* , *E / S* , *XML* , *definições de DataType* , etc. para as várias linguagens de programação *.NET* .
- **Thread Support:** O CLR fornece suporte de thread para gerenciar a execução paralela de vários threads. A *classe System.Threading* é usada como classe base para isso.
- **COM Marshaller:** A comunicação com o componente COM (Component Object Model) no aplicativo .NET é fornecida usando o COM Marshaller. Isso fornece o suporte de interoperabilidade COM.
- **Type Checker:** a segurança de tipo é fornecida pelo verificador de tipo usando o Common Type System (CTS) e a Common Language Specification (CLS) que são fornecidos no CLR para verificar os tipos que são usados em um aplicativo.
- **Exception Manager:** o gerenciador de exceções no CLR trata as exceções independentemente da *linguagem .NET* que as criou. Para um determinado aplicativo, o bloco catch das exceções são executados caso ocorram e se não houver bloco catch, o aplicativo é encerrado.
- **Security Engine:** O mecanismo de segurança no CLR lida com as permissões de segurança em vários níveis, como nível de código, nível de pasta e nível de máquina. Isso é feito usando as várias ferramentas fornecidas na estrutura *.NET* .
- **Debug Engine:** um aplicativo pode ser depurado durante o tempo de execução usando o mecanismo de depuração. Existem várias interfaces ICorDebug que são usadas para rastrear o código gerenciado do aplicativo que está sendo depurado.
- **JIT Compiler:** O compilador JIT no CLR converte o Microsoft Intermediate Language (MSIL) no código de máquina que é específico para o ambiente do computador em que o compilador JIT é executado. O MSIL compilado é armazenado de forma que esteja disponível para execuções subsequentes, se necessário.
- **Code Manager:** O gerenciador de código no CLR gerencia o código desenvolvido na estrutura .NET, ou seja, o código gerenciado. O código gerenciado é convertido em linguagem intermediária por um compilador específico da linguagem e, em seguida, a linguagem intermediária é convertida em código de máquina pelo compilador Just-In-Time (JIT).
- **Garbage Collector:** o gerenciamento automático de memória é possível usando o coletor de lixo no CLR. O coletor de lixo libera automaticamente o espaço de memória depois que ele não é mais necessário para que possa ser realocado.
- **CLR Loader:** Vários módulos, recursos, assemblies, etc. são carregados pelo carregador CLR. Além disso, este carregador carrega os módulos sob demanda se eles forem realmente necessários para que o tempo de inicialização do programa seja mais rápido e os recursos consumidos sejam menores.