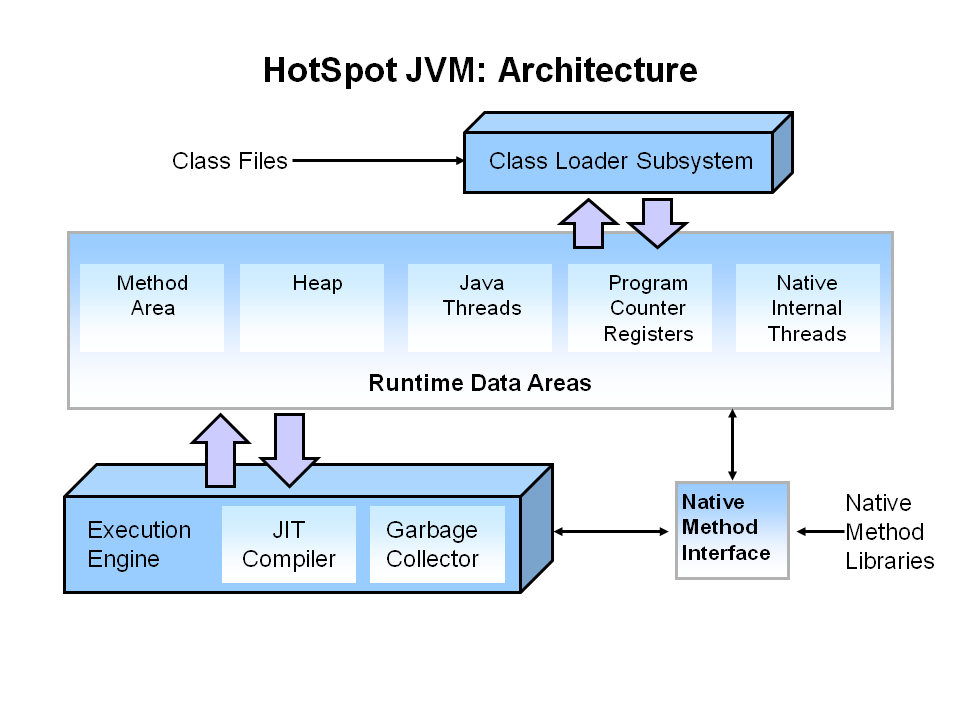
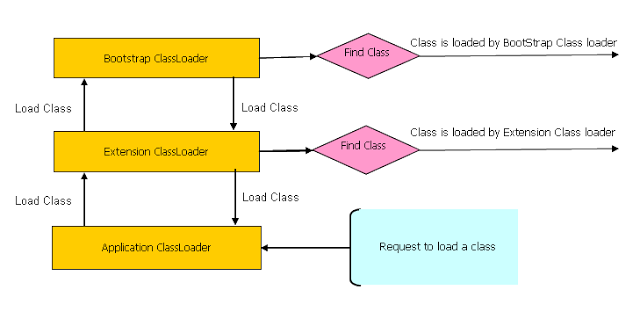
# JVM



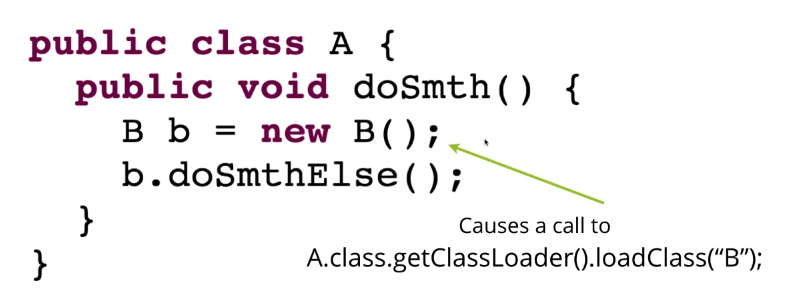
* JRE consists of JVM and JAVA runtime API
* The main components of the JVM include the classloader, the runtime data areas, and the execution engine.
* Java command create an instance of JVM

### Class Loader

* Class Loader responsible for bringing class file into the JVM. It is loading class files from file system, network or any other source.
* There are 3 types of class loader in java-  
    
    
  Whenever a new JVM is started by typing java MyMainClass, the **"bootstrap class loader"** is responsible for loading key Java classes like java.lang.Object and other runtime code into memory first. The runtime classes are packaged inside of the JRE\lib\rt.jar file.

Next come the **Java extension class loader**. We can store extension libraries, those that provide features that go beyond the core Java runtime code, in the path given by the **java.ext.dirs** property. The **ExtClassLoader** is responsible for loading all .jar files kept in the java.ext.dirs path. A developer can add his or her own application .jar files or whatever libraries he or she might need to add to the classpath to this extension directory so that they will be loaded by the extension class loader.

The third and most important class loader from the developer perspective is the **AppClassLoader**. The application class loader is responsible for loading all of the classes kept in the path corresponding to the classpath environment variable.

* 
* How to load a Class?

Java provides API to explicitly load a class by Class.forName(classname) and Class.forName(classname, initialized, classloader), remember JDBC code which is used to load JDBC drives.  
**Example** - com.src.java.classloader – ClassLoaderTest.java

* Java’s “**SystemClassLoader**” can use to pint out the current project classpath , indirectly display the library dependency

**Example** - com.src.java.classloader – SystemClassLoaderTest.java

* Custom Class Loader [**discuss later after Nested class**]

### Garbage Collector

* Garbage Collector is used to remove objects which go out of scope.
* Automatic garbage collection is the process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects. An in use object, or a referenced object, means that some part of your program still maintains a pointer to that object. An unused object, or unreferenced object, is no longer referenced by any part of your program. So the memory used by an unreferenced object can be reclaimed.
* JVM following 3 steps to do Garbage collection based on “**Mark & Sweep**” algorithm

**Step1: Marking**

Garbage collector identifies which pieces of memory are in use and which are not.

**Step2: Normal Deletion**

Normal deletion removes unreferenced objects leaving referenced objects and pointers to free space.

**Step3: Deletion with Compacting**

To further improve performance, in addition to deleting unreferenced objects, JVM can also compact the remaining referenced objects. By moving referenced object together, this makes new memory allocation much easier and faster.

* Heap related switches

|  |  |
| --- | --- |
| **Switch** | **Description** |
| -Xms | Sets the initial heap size for when the JVM starts. |
| -Xmx | Sets the maximum heap size. |

* System.GC() command

We can explicitly invoke the GC (System.gc()) but it is not guaranteed.

### Java Runtime Memory

* Java Runtime Memory Area consist **Method Area, Heap, Java Stack, and Native Stack**
* Java objects reside in an area called *the* ***heap***. The heap is created when the JVM starts up and may increase or decrease in size while the application runs. When the heap becomes full, *garbage* is *collected.*During the garbage collection objects that are no longer used are cleared, thus making space for new objects.
* **Instance variables** and object lives on Heap. Each object into the heap has a pointer to the Method Table related to its method.
* Prints JVM memory utilization statistics

**Example** - com.src.java.classloader – TestMemory.java

* Regardless of no. of Object, there should be only **one copy of method is loaded into Method Area**. Method Area consist Method Table which get reference of location where methods are loaded into Method Area.
* All static variables get the spaces into the Method Tables which is a global area.
* For each thread, there is a separate Java Stack / Native Stack**:**  
  For each method inside a Java Stack there is a Java Stack Frame. Stack frame consists of any local variables which are defined inside a method, any arguments which are passing to a method, next instructions to be executed inside method, and also the status of the return type.
* Gargabe Collection will invoke as and when required to collect all object which go out of scope (Heap & Method Area)
* When stack memory is full, Java runtime throws **java.lang.StackOverFlowError** whereas if heap memory is full, it throws **java.lang.OutOfMemoryError**: Java Heap Space error.

# Reflection

Java Reflection makes it possible to inspect classes, interfaces, fields and methods at runtime, without knowing the names of the classes, methods etc. at compile time. It is also possible to instantiate new objects, invoke methods and get/set field values using reflection.

### Why do we need reflection?

Reflection enables us to:

* Examine an object's class at runtime
* Construct an object for a class at runtime
* Examine a class's field and method at runtime
* Invoke any method of an object at runtime
* Change accessibility flag of Constructor, Method and Field

### Class Summary

The java.lang.Class class provides many methods that can be used to get metadata examine and change the run time behavior of a class.

Class  
String getName()  
Class getSuperClass()  
Class[] getInterfaces()  
Methods[] getMethods()  
Constructor[] getConstructors()  
Field[] getFields()  
Method[] getDeclaredMethods()  
Constructor[] getDeclaredConstructors()  
Field[] getDeclaredFieds()  
Object newInstance()  
Class forName(String)  
Class forName(String, boolean initialize, ClassLoader)

### Examples

1. Create Object dynamically & Invoke a method of a class dynamically

**Example** - com.src.java.reflection – InvokeMethod.java

1. Debuggers use reflection to inspect dynamically the code that is being executed.
2. Test tools like Junit or Mockito use reflection in order to invoke desired methods containing specific syntax or to mock specific classes, interfaces and methods.
3. Code analysis tools like PMD or Findbugs or SONAR use reflection in order to analyze the code against the list of code violations that are currently configured.
4. Class loader uses reflection to load a class.
5. Access Private members

**Example** - com.src.java.reflection – Person.java, AccessPrivateMembers.java