* **Java** **Activities**:
* **Abstract**: UFT’s Java framework works with JNI, the flow in high-level is as follows:



* **Code Structure**:
  + Basically, we have 2 technologies that work with Java:
    - **JMS**: all the JMS activities are Java based. See details about JMS below
    - **Java Call activity**: This activity allows customer to call JAR or CLASS files. See details about it below.

Each one of these 2 activities has its own C# projects and C++ Adaptor

* + **JVMLoader** (in SOA2TechRTFWK): C# project that is responsible for loading and unloading JVM (by calling **JavaInfra**). JVM is loaded in every UFT script that contains Java based activity.  
    This code is invoked from Main.cs of UFT script, the invoking code is generated by Start activity’s translator.
  + **JavaInfra** (in SOA2TechRTFWKCpp): C++ project that is responsible for loading and unloading JVM with proper settings by JNI calls. the major class there is ‘**jvm\_env.cpp**’,
* **How to debug Java code:** if you need to debug the Java code, here are the steps you need to follow:
  + Open the project in eclipse (or other Java IDE)
  + Add environment variable: ‘**JNI\_DEBUG\_PORT**’ with value of any available port number (usually we use 8000). You can add also the variable ‘**JNI\_DEBUG\_SUSPEND**’ with value ‘y’ (for ‘yes’), this will suspend the JVM until debugger connects to it.
  + Run the script, if you didn’t add ‘**JNI\_DEBUG\_SUSPEND**’, then add a breakpoint in C# code after loading the JVM and before calling the needed java code.
  + Go to eclipse, add breakpoints and connect to JVM with port number that you specified in **JNI\_DEBUG\_PORT**’ value
  + Now eclipse debugger is connected to JVM in, just continue running the script, it will break on Java code (if it is in its path)
* **JMS**
* **Abstract:** see here (although some of information are out-of-date, the explanation is good enough): <https://rndwiki.corp.hpecorp.net/confluence/display/soa/Application+Servers+with+JMS+Environments>
* **Code Structure**:
  + **Activities code**: in **SOA2TechImplEXT** solution, we have the standard 2 projects
    - JMSTransport – it contains runtime classes
    - JMSTransportDT – it contains design time data
  + **JMSTransportAdaptor** projectin solution **TechImplExtCpp**, it is the JMS C++ adaptor, i.e. it is the C++ code that calls Java JMS code. Major class in this project is ‘**JMSTransport**.**cpp’**, and the entry point of it is the method CJMSTransport::init\_jms().
  + **JavaJMSTransport** folder (in the folder of **SOA2TechImplEXT** solution). It is a **Java** project (can be opened in eclipse). Major classes here are:
    - ‘**WorkflowManagerImpl’**,this is the class that calls JMS in real, e.g. ‘sendMessage’ sends a message to queue, and ‘receiveMessage’ receive message from queue.
    - **‘JMSSupportImpl’**, this class prepare the JMS calls (e.g. creates Connection, session and Queue object for each call), and then calls ‘**WorkflowManagerImpl’**
    - **‘JMSBridge’**, this class is the entry point of Java code, i.e. C++ code calls this class’s methods (at first we call ‘init\_jms’ after that we call the methods according to what activities we have in the script).  
      So, the flow here goes like that:

**Activity RT code 🡪 JMSTransportAdaptor 🡪 JMSBridge 🡪 JMSSupportImpl 🡪 WorkflowManagerImpl 🡪 JMS server**

* **Debugging** java code goes exactly as explained above (in Java Activities)
* **Java Call**
* **Abstract:** this method allows customer to call Java code: jar file or classes files. The called jar file must implement the interface ‘ServiceTestCall’. The jar file must be compiled with Java 7 or less (we don’t support Java 8 – UFT12.53).  
  This feature has 2 phases:
  + **Design** **time**: we uncompile the Jar file to temp directory, customer selects the class that implements the interface. After that, we read the input and output properties of the call (by calling the methods: ‘*getInputProperties’* and ‘*getOutputProperties’*, and this is why we need a class to implement the interface), and add them to the activity on the canvas.
  + **Run time**: here we call the method ‘*execute’* from the class that implements the interface.
* **Code Structure**:
  + **Runtime**: it is very similar to JMS flow, the C++ adaptor here is ‘JavaCallAdaptor’, and we don’t have Java code because it is provided by customer.  
    The RT project is ‘JavaCallRT’ in ‘SOA2TechImplEXT’ solution
  + **Design time**: in design time we start a WCF service, because a limitation that we can’t load JVM from UFT process (because of collisions), so, we start the WCF host, we call its API to load JVM so JVM will be loaded by the service process, the same for calling ‘*getInputProperties’* and ‘*getOutputProperties’*. The design time projects are:
    - JavaCallDT: which contains the standard DT data (signature…)
    - JavaCallService: the WCF service.
    - JavaCallServiceHost: the WCF service host
    - JavaCallServiceProxy: the client that works against the WCF service