# Dynamic Java Class Loader - Documentation

Prepared by: Mohammad Jawad Zein El Deen.

Presented to: Dr. Hamid Mcheik.

Academic Year: 2018-2019.

Date Submitted: 30 Dec 2018.

## Contents

## Client Side

In order to establish a connection with the server we need a socket, we will use BufferedReader to read from screen client input and another to get server response, PrintWriter to send request to server.

```
public class Client {
        private Socket socket;
        private BufferedReader inputStream;
        private BufferedReader bufferedReader;
        private PrintWriter printWriter;
        public Client (Socket s) throws IOException {
                socket = s;
                inputStream = new BufferedReader(new InputStreamReader(System.in));
                bufferedReader = new BufferedReader(new InputStreamReader(socket.getInputStream()));
                printWriter = new PrintWriter(socket.getOutputStream(),true);
        }
```

### Sending File over socket

containing the java code is carried out over the following steps

1. Client ask program to send file
2. Program ask client for path of file
3. Client enter the file path
4. Program convert the file to byte array and send the file over `DataOutputStream`
5. Client must then send the Class name
6. Final step server compiles the code (for code explanation look in next chapter)

```
private void sendFileOverSocket() throws IOException {
    printWriter.println("a");//send to server 1
    String response = bufferedReader.readLine();
    if(! response.equals("OK For Sending File")) {
        System.out.println("Server Sent : " + response );
        return;
    }
    System.out.println("Enter the directory path of file to send, example D:/UPDIR/Calculator.java");
    String FILE_PATH = inputStream.readLine();
    byte [] bytes = simulateFileByteArr(FILE_PATH);
    //byte [] bytes = simulateFileByteArr("D:/UPDIR/Calculator.java");
    //System.out.println("length of bytes " + bytes.length);
    printWriter.println(bytes.length+"");
    DataOutputStream out = new DataOutputStream(new BufferedOutputStream(socket.getOutputStream()));
    out.write(bytes);
    out.flush();
    response = bufferedReader.readLine();
    if(response.equals("Sent File  Recieved")) {
        System.out.println("File Sent and saved, Please enter class name as in file, example Calculator");
        printWriter.println(inputStream.readLine());
        //printWriter.println("Calculator");
    }
    System.out.println(bufferedReader.readLine());
}
```

### Invoking method from the client side

After sending the code and compiling it, client can send a request for invoking a method in this class.

```
private void callMethodOverSocket() throws IOException {
    printWriter.println("callMethodOverSocket");
    System.out.println("enter your input, use the folloing format MethodName;parm1;parm2;...");
    String request = inputStream.readLine();
    printWriter.println(request);
    System.out.println("Request : " + request);
    System.out.println("From SERVER : " + bufferedReader.readLine());
}
```

Invoking the method must be of the following format: Method Name then “;” followed by as suitable number of parameters separated by “;”.

Server will then return the response of the invoking process.

## Code for reading file into Byte Array on client side

```
public static byte[] simulateFileByteArr(String FILE_PATH) {
    File file = new File(FILE_PATH);
    FileInputStream fis = null;
    try {
        fis = new FileInputStream(file);
        byte fileByte[] = new byte[(int)file.length()];
        fis.read(fileByte);
        fis.close();
        return fileByte;
    } catch (Exception e) {
        e.printStackTrace();
    }
    return null;
}
```

---

## Sample code used in testing

```
public class Calculator{
    public String getFuncionNames() {return "add,sub,mul,div,pow";}
    public double add(double a,double b) {return a+b;}
    public double sub(double a,double b) {return a-b;}
    public double mul(double a,double b) {return a*b;}
    public double div(double a,double b) {return a/b;}
    public double pow(double a,int b) {
        if (b ==0) return 1;
        if (b > 1) return a*pow(a,b-1);
        if (b < 0) return 1/a * pow(a,b+1);
        return a;
    }
}
```

---

## Server Side

### Server main program

```
public static void main(String [] args) {
    try {
        System.out.println("Server Started");
        @SuppressWarnings("resource")
        ServerSocket serverSocket = new ServerSocket(8080);
        while(true) {
            try {
                Socket client = serverSocket.accept();
                (new ServerThread(client)).start();
            }
            catch(Exception e) {
                break;
            }
        }
    }
    catch(Exception e) {
        e.printStackTrace();
    }
}
```

Will establish server socket and create a thread server for every client.

### Handling the sent file over socket

```
private void handleSentFile() throws IOException {
    printWriter.println("OK For Sending File");
    DataInputStream in = new DataInputStream( new BufferedInputStream(clientSocket.getInputStream()) );
    int byteCount = Integer.parseInt(bufferedReader.readLine());
    byte [] inputBytes = new byte[byteCount];
    in.read(inputBytes);
    printWriter.println("Sent File  Recieved");
    String FileName = bufferedReader.readLine();
    File JavaFile = new File("ClientJavaFile/"+FileName+".java");
    if (JavaFile.getParentFile().exists() || JavaFile.getParentFile().mkdirs()) {
        try (FileOutputStream stream = new FileOutputStream("ClientJavaFile/"+FileName+".java")) {
            stream.write(inputBytes);
        }
    }
    try {
        obj = compileFile(JavaFile , FileName);
        if(obj != null) {
            printWriter.println("file compiled");
        }
        else {
            printWriter.println("file not compiled");
        }
    }
    catch(Exception e) {
        e.printStackTrace();
    }

}
```

This process is composed of three main phases:

1- Reading sized of bytes to create byte array of same size

2- Acquiring the bytes sent to server object
3- Save and compile the code sent.

## Compiling Code

```
public Object compileFile(File JavaFile , String className) throws Exception {
    DiagnosticCollector<JavaFileObject> diagnostics = new DiagnosticCollector<JavaFileObject>();
    JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
    StandardJavaFileManager fileManager = compiler.getStandardFileManager(diagnostics, null, null);
    List<String> optionList = new ArrayList<String>();
    optionList.add("-classpath");
    optionList.add(System.getProperty("java.class.path") + ";dist/InlineCompiler.jar");
    Iterable<? extends JavaFileObject> compilationUnit
    = fileManager.getJavaFileObjectsFromFiles(Arrays.asList(JavaFile));
    JavaCompiler.CompilationTask task = compiler.getTask(
        null,
        fileManager,
        diagnostics,
        optionList,
        null,
        compilationUnit);
    if (task.call()) {
         URLClassLoader classLoader = new URLClassLoader(new URL[]{new File("./").toURI().toURL()});
         Class<?> loadedClass = classLoader.loadClass("ClientJavaFile." + className);
         Object obj = loadedClass.newInstance();
         classLoader.close();
         fileManager.close();
         compiledClass = loadedClass;
         return obj;
    }
    else {
        for (Diagnostic<? extends JavaFileObject> diagnostic : diagnostics.getDiagnostics()) {
            System.out.format("Error on line %d in %s%n",
                    diagnostic.getLineNumber(),
                    diagnostic.getSource().toUri());
        }
    }
    fileManager.close();
    return null;
}
```

Above code is a generic function that will take as input the File and the class name, and will attempt to compile code existing in file.

Using tools.jar, and methods within, compiling code and creating new instance object of the code.

We save the class and object in server ram for usage by client afterwards for calling the methods.

## Server invoking methods dynamically

Client will send the request method name; parameters…;

If no parameters are defined there is no need for any interaction, however, acquiring the input parameters and embedded in string format was tricky. Therefore, the solution was to create a method caster that will handle un cast-able classes, such as float, double and integer, and other that may be appended to the method since calling is dynamic.

```
private void runMethodOverSocket() throws IOException {
    try {
        String request = bufferedReader.readLine();
        StringTokenizer st = new StringTokenizer(request,";");
        String methodName = st.nextToken();
        Object InvokeResult = null;
        Method [] methods = compiledClass.getDeclaredMethods();
        for(Method m : methods) {
            if(m.getName().equals(methodName)) {
                Map<Integer,Class<?>> mapParameterClass = new HashMap<Integer,Class<?>>();
                int i = 0;
                for(Class<?> cls : m.getParameterTypes()) {
                    mapParameterClass.put(i++, cls);
                }
                List<Object> objects = new ArrayList<Object>();
                i = 0;
                while(st.hasMoreTokens()) {
                    Class<?> cls = mapParameterClass.get(i++);
                    Object o = CastObject(st.nextToken(),cls);
                    objects.add(o);
                }
                InvokeResult = m.invoke(obj,objects.toArray());
                break;
            }
        }
        printWriter.println("Result from invoking " + InvokeResult);
    }
    catch( Exception e) {
        e.printStackTrace();
```

Using reflect library, we can obtain declared methods of the class. Case where method accepting different types for different declarations is not handled. We consider only one method exist for a given name.

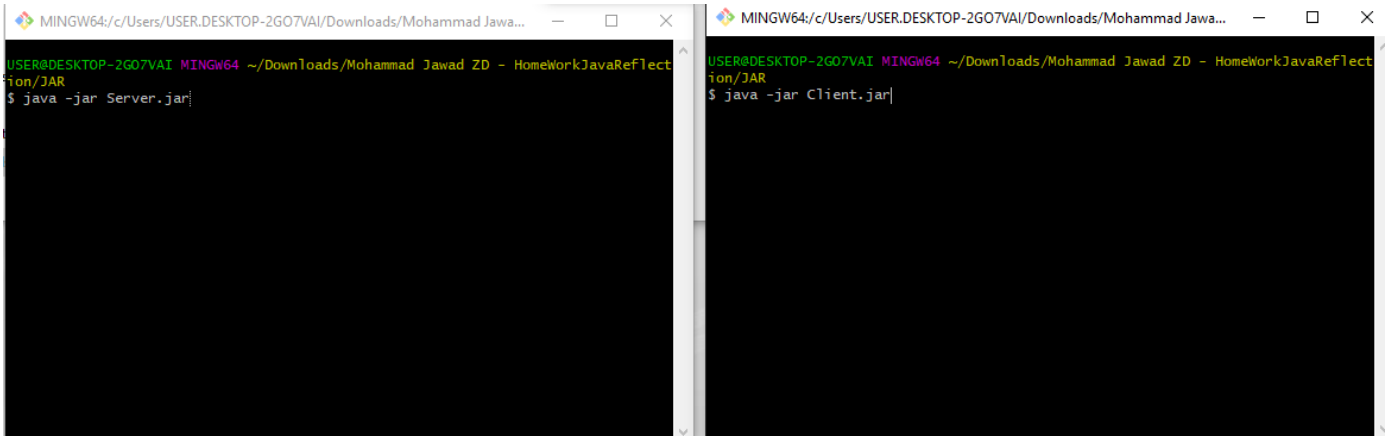
We handled the dynamicity of parameter types and casting for known non-cast-able classes like integer and double.

### Method for handling casting

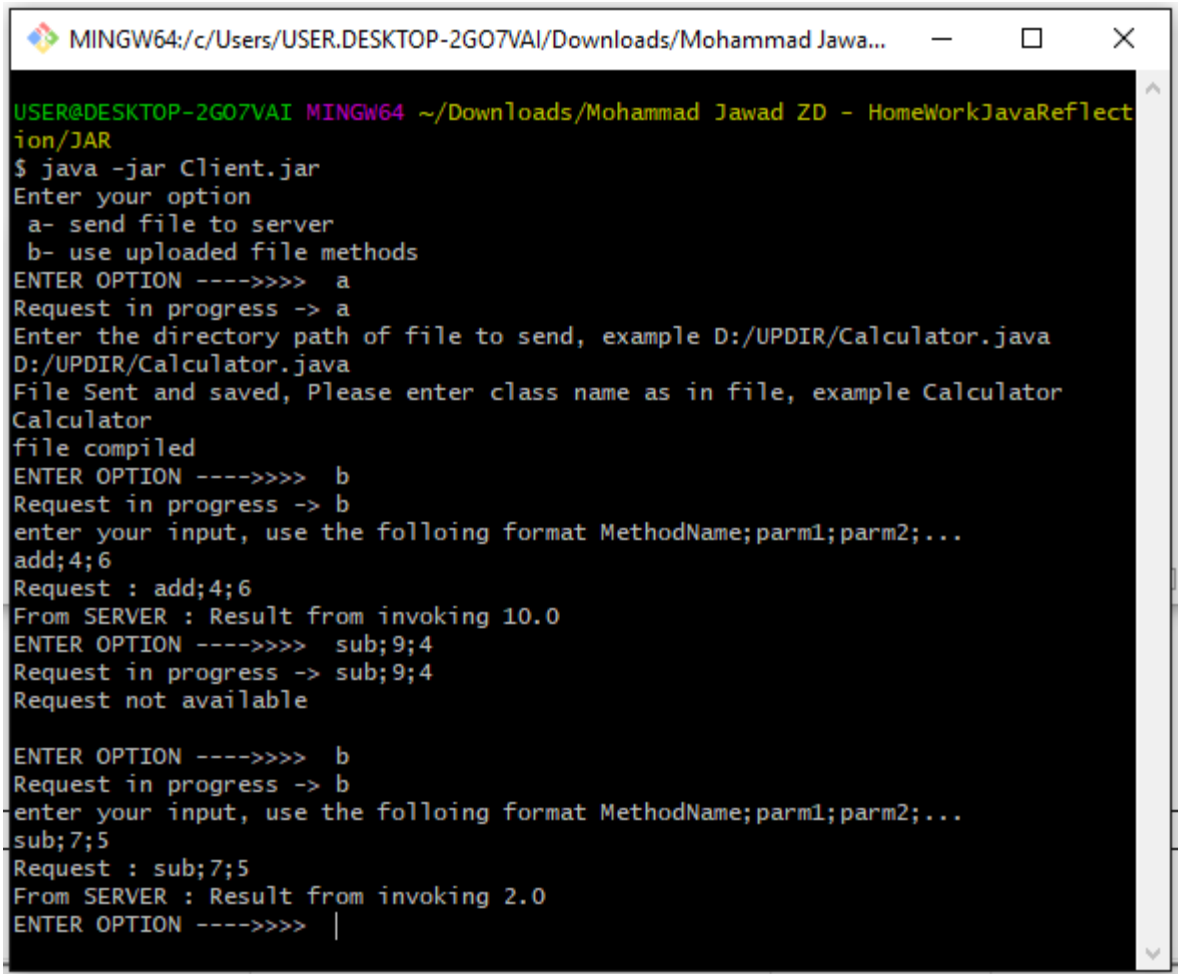
```
private Object CastObject(Object o,Class<?> cls) {
    Object ob = o;
    String className = cls.getName();
    switch(className) {
    case "double" : return Double.parseDouble(o.toString());
    case "int" : return Integer.parseInt(o.toString());
    default : try{
                    return cls.cast(ob);
                }
                catch(Exception e) {
                    return ob;
                }
    }
}
```

## Case Study – Testing the code

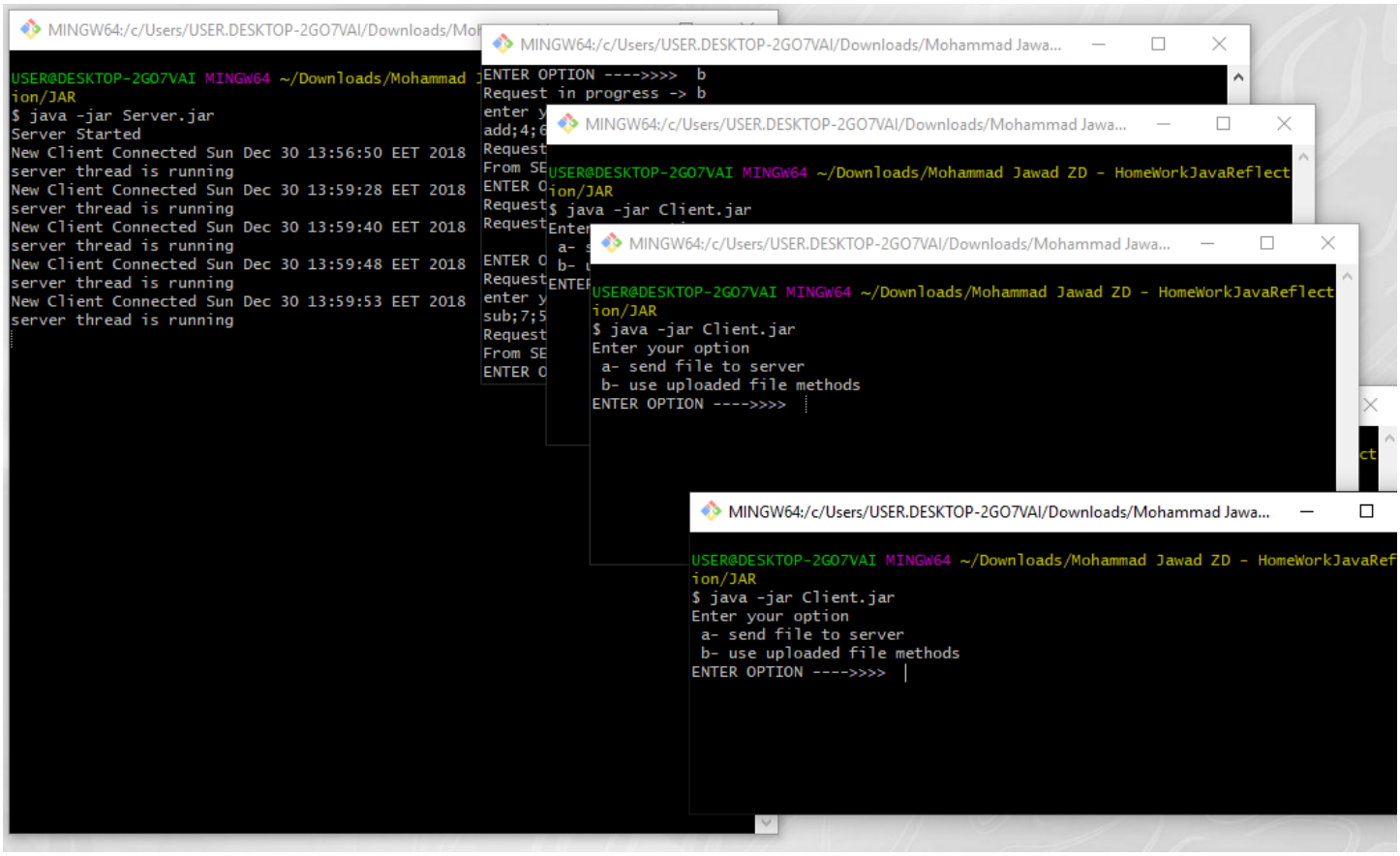
Compiled versions of jar files can be invoked using command java –jar filename.jar




After following instructions, we are able to call any method existing in the java file sent to server.




Server program will show status of connected clients


MINGW64:/c/Users/USER.DESKTOP-2GO7VAI/Downloads/Mohammad Jawa...
USER@DESKTOP-2GO7VAI MINGW64 ~/Downloads/Mohammad Jawad ZD - HomeWorkJavaReflection/JAR
$ java -jar Server.jar
Server Started
New Client Connected Sun Dec 30 13:56:50 EET 2018
server thread is running
New Client Connected Sun Dec 30 13:59:28 EET 2018
server thread is running
New Client Connected Sun Dec 30 13:59:40 EET 2018
server thread is running
New Client Connected Sun Dec 30 13:59:48 EET 2018
server thread is running
New Client Connected Sun Dec 30 13:59:53 EET 2018
server thread is running
ENTER OPTION ---->>>> b
Request in progress -> b
$ java -jar Client.jar
Enter your option
a- send file to server
b- use uploaded file methods
ENTER OPTION ---->>>>