Speare Code Editor
The Small IDE for
Java Development

Free, Lightweight, Open Source, Extendable Flexibility

Speare (http://www.sevenuc.com/en/speare.html) is a small and friendly code editor. It was originally developed to provide a native scripting language debugging environment that seamlessly integrated with C and C++. Speare designed to make programming feels light, simple and free. Speare not only has an efficient code navigation and call routines tracing ability but also has flexibility to extend it to support special developing requirements. Speare includes an ultra lightweight debugging environment for C, C++, Ruby, mruby, Lua, Python, PHP, Perl and Tcl, and give people complete freedom to control and customise the debugging environment for a new programming language.

Debug Java Projects

1. Show the debug toolbar
Click main menu "Navigate" → "Toggle Output".

2. Debug toolbar

From left to right: Start, Stop, Step Into, Step Out, Run To/Continue, Step Over/Step Next, Show Watches. The command "Run To" tell the debugger run to meet a breakpoint or an exception occurred or the program exited. On the rightmost there are three other icons, they are, search items in the stackview, siding stackview, and clean debug output.

Search in the debug output
Click in the output area to let it got focus and use the shortcut key "Command + F" to do the searching.

3. Socket Port
You can set the socket communication port number both used by Debug Server and the editor. Open the Preferences of Speare and select the "Extra" panel and then input your number.

The port number should be same as Java debug option:
-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=localhost:*7001*

Note: Please remember to empty the port number when you switched to debugging with the default built-in programming languages with default port number.

4. Watches
Watch List: click debug toolbar "Show Watches" to manage watched variables and expressions, batches of watched items can be removed by multiple selection.

a. Whenever values of watched variables changed, debug session will pause at that point.
b. The values of evaluated expressions will be sent as a *"virtual stack node"* that paired the expression and its value together and representing as JSON key and value.

When watched variables and expressions display in stackview, their nodes normally has a different icon and text colour, and always placed on the top of stackview.

Add Watches:
a. Click debug toolbar "Show Watches".
b. Right click stackview and select menu item "Watch Variable".

Remove Watches:
a. Click debug toolbar "Show Watches".
b. Right click stackview and select menu item "Remove Watch".

5. Run Extra Commands
a. Click main menu "Debug" → "Send Debug Command".

b. Right click stackview and select menu item "Send Command". When send an extra debug command, the input box will prompt in stackview, so please click stackview to let it got focus before select menu item.

6. Show Stackview Item Values

Right click stackview and select menu item "View Value AS", variables can be configured to display as "Hex", "Decimal", "Octal", "UTF-8" and "Unicode" sequences.

Caution:
Please ensure all source files have been dragged in the workspace before start a debug session, because macOS app can't be allowed to access files outside of its sandbox.

Startup Debugging Session

1. Launch a Terminal.app window or tab, and start Debug Server.
Note: The content directly printed by the debugging program will be displayed in the terminal instead of in the debug "output view".
2. Click ▷ start button on the "Debug toolbar" to start debugging session.
3. Continually click ⬚"Step Over" button several times on the "Debug toolbar" to ignore some initialising steps in debugging session until it reached the main entry point.

The Java Debugger

The Java debug environment of Speare support all kinds of JVM that implemented Java Debugger API, JDK all version from 6 and later, and all kinds of Java application, J2ME (Java ME) games, Applet on web page, standalone Java J2SE application for desktops, J2EE (Java EE) server side application for all kinds of application server, such as Tomcat, Spring, Weblogic,WebSphere, JBoss (Wildfly), Glassfish, and Jetty etc. You can enjoy debugging almost any type of Java application under the lightweight debugging environment of Speare code editor.
(Java debugger is newly added)

Setup Java development environment on macOS

1. Download JDK from Oracle website and install it (register required), the download file normally has a name like so:
    jdk-8u171-macosx-x64.dmg.

2. Properly setup *JAVA_HOME* environment variable if there's multiple JDK installed in your system, run the following commands:

```
$ export JAVA_HOME=/Library/Java/JavaVirtualMachines/jdk1.8.0_171.jdk/Contents/Home
$ sudo update-alternatives --config javac
$ sudo update-alternatives --config java
```

Test location of Java executable:

```
$ which java
$ java -version
```

3. Append the following four lines into ~/.zshrc (or ~/.bash_profile):

```
export JAVA_HOME=$(/usr/libexec/java_home)
export JAVACMD=$JAVA_HOME/bin/java
export CLASSPATH=.:$JAVA_HOME/lib:$JAVA_HOME/jre/lib/
export PATH=$JAVA_HOME/bin:$PATH
```

4. Run command "source ~/.zshrc", quit Terminal.app and restart to test the above settings.

```
$ java -version
$ which java
```

The printed path of java command should *not be*:
/System/Library/Frameworks/JavaVM.framework/Versions/Current/Commands/java

If the path has prefix "/Library/Java/JavaVirtualMachines/jdkxx.xx.xx.jdk", then you can run more test commands:

```
$ java -version
$ javac -version
```

The printed version should have the same value of your installed JDK, e.g. "java version "1.8.0_171"".

Start Debugging Steps:
1. Download Speare debug server:

→ http://www.sevenuc.com/debuggers/java_debugger.zip.

2. Uncompress the tarball to your local directory, e.g. ~/Desktop and take a look at the readme.txt file in it.

3. Compile Java project with -g, the compiler debug option.

a. for Ant:

```
<target name="compile" depends="clean, makedir">
  <javac srcdir="${src.dir}" destdir="${build.dir}"
 classpathref="build.classpath" includeantruntime="false"
encoding="UTF-8" debug="on">
  </javac>
</target>
```

b. for Maven:

```
<configuration>
    <source>1.8</source>
    <target>1.8</target>
    <encoding>UTF-8</encoding>
    <compilerArgs>
    <arg>-g</arg>
    </compilerArgs>
</configuration>
```

c. for Gradle:
Set Java compile options for JavaCompile tasks:

```
tasks.withType(JavaCompile) {
    options.compilerArgs += ['-g', '-Xdoclint:none', '-Xlint:none', '-nowarn']
}
```

Caution: stack tracking can't properly work if class file compiled without debug option.

4. Lanuch JVM and connect with debug server:
Launch approach A:
directly start JVM with debug server (suitable for standalone non-GUI library or applications).

```
$ java -jar sds.jar -sourcepath "/Users/username/full/path/of/project/src" -cp ... com/z/y/x/mainClass
```
(using class files: sds.jar and folder "com/z/y/x/..." both should be under the same directory)
```
$ java -jar sds.jar -sourcepath "/Users/username/full/path/of/project/src" -cp ... -jar libx.jar
```
(using jar file directly)

Launch approach B:
start JVM and attach debug server to it (suitable for server, Android, and GUI applications).

```
$ java -agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=localhost:7001 -cp ... com/z/y/x/mainClass
$ java -agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=localhost:7001 -cp ... -jar libx.jar
$ java -jar sds.jar -sourcepath "/Users/username/full/path/of/project/src" -attach localhost:7001
```

5. Debug session start: click "Start" button on the debug toolbar of Speare.

Add breakpoint, step in, step out, step next, watch stack trace ...

6. Run extra commands:

a. Show version

. version: fetch version info of the debugger.

b. Show help

.help: fetch command list of the debugger.

c. Show source code directory

.sourcepath: which used to look for source files, source code directory of the application.

d. Show class path

.classpath: list directories in which to look for classes, the libraries used by the application.

e. Add function breakpoints

.stop in classname.method: add a breakpoint, e.g. "stop in com.x.y.classname.method(int, String)".

f. Clear breakpoints

.clear classname.method: clear a breakpoint, e.g. "clear com.x.y.classname.method(int, String)".

g. Run application

.run: start JVM and start execution of the main class of debugged application.

h. List threads

.threads: show threads that are currently running, their name and current status.

i. Switch thread

.thread `thread name`: select a thread to be the current thread.

j. Dump thread

.where [`threadindex`] : dumps the stack of the current thread.

k. Display object value

.print [MyClass.myStaticField|myObj.myInstanceField] : find out the value of an object.

l. Set variable value

.set `set = `: assign value to field/variable/array element when application is running.

m. Evaluate expression

.eval `expression`: (like print).

n. Move stack frame

.up [n frames]: move up a thread's stack one or n frames.

.down [n frames]: move down a thread's stack one or n frames.

o. Continues execution

.cont: continues execution after a breakpoint, exception, or step.

p. Show class details

.classes: list currently loaded classes in JVM.

.class `<class id>`: show details of named class.

.methods `<class id>`: show methods list of a class.

.fields `<class id>`: show fields list of a class.

q. Hot code replace

.redefine `<class id> <class file name>`: edit code and replace class while debugging.

r. Run command from file
. Load file sds.ini from launch directory, and execute each line as a command, line
begins with "#" is comment.
s. Ignore execution paused at specified package
.exclude `<class pattern>`: e.g. "exclude com.z.y.x.*;com.z.y.v.*", each pattern separated by a ";".
...

Debugging Android Application

1. launch emulator → start "Dev Tools" app inside the emulator →
"Development Settings", select your application as the "Debug
App" → check on "Wait for debugger".

2. start the Dalvik Debug Monitor ("ddms" tool in the Android SDK),
this will allow debugger to connect to running apps inside the
emulator or on a connected device.

3. Run your app inside the emulator, a message will prompt and
say:

"is waiting for the debugger to attach".

4. $ java -jar sds.jar -sourcepath "/Users/username/path/to/project/src" -attach localhost:8700
INSTALL_FAILED_INSUFFICIENT_STORAGE
To increase the size available to app, the emulator must be started with the
"-partition-size" parameter. This command reserves 100MB for apps:
$ emulator -partition-size 100 -avd <virtual-device-name>

Details Explained

In order to attach debugger to an Android application, which is running
inside the Dalvik VM, we have to use adb bridges the gap between an application
and a development/debugging environment. The Dalvik VM creates a JDWP
thread for every application to allow debuggers to attach to it on certain
ports/process IDs. In order to find out the JDWP port of a debuggable
application, run the command:

  $ adb jdwp

This will return a list of currently active JDWP processes, the very last
number corresponds to the last debuggable application launched. To attach
debugger to the remote VM we have to have adb forward the remote JDWP port
to a local port. This is done with the forward command, like so:

```
$ adb forward tcp:7001 jdwp:JDWP_PORT
```

adb will open a local TCP socket that you can connect to, and will forward all data sent to the local TCP socket to the JDWP process running on the device/emulator. After forward data, attach debugger like normal:

```
$ java -jar sds.jar -sourcepath "/Users/username/full/path/of/project/src" -attach localhost:7001
```

Debugging Server Applications

1. Settings to launch Tomcat applications

The startup script for Tomcat is named catalina.sh, to start a server with debug arguments:

```
$ catalina.sh jpda start
```

The default listening on port is 8000 and suspend=n, them can be changed by environment variables:

JPDA_TRANSPORT, JPDA_ADDRESS, and JPDA_SUSPEND.

2. Settings to launch Spring applications

a. launch application from the command line with debug arguments:

```
$ java -jar myapp.jar -Dagentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=7001
```

b. Maven:

```
$ mvn spring-boot:run -Dagentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=7001
```

c. Gradle:

* ensure Gradle passes command line arguments to the JVM:

```
bootRun {
    systemProperties = System.properties
}
$ gradle bootRun -Dagentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=7001
```

3. Weblogic

The startup script for Weblogic is startWeblogic.sh, to start a server with debug enabled just set the environment variable debugFlag to true. The default listener on port is 8453 and suspend=n, can override by setting the DEBUG_PORT environment variable.

4. WebSphere

Check the startup script for WebSphere and find debug option and env variables to setup JVM debug enabled,

```
WAS_DEBUG
-J <java_option>
-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=7001
```

Check points:

   0. don't configure debug options on web page.

   a. /WebSphere/AppServer/bin/startServer.sh

   b. /WebSphere/AppServer/profiles/WRSProfile/config/cells/WRSNodeCell/ nodes/WRSNode/servers/server1/server.xml

   c. associated service for the profile with 'manual' mode.

   d. stop and start the server.

5. JBoss (Wildfly)

   The startup script for JBoss is stand-alone.sh, to start a server with debug enabled just add –debug. The default listener on port is 8787 and suspend=n, can be override by specifying it after –debug argument.

6. Glassfish

   The startup script for Glassfish is asadmin, to start a server with debug enabled just add –debug:

   $ asadmin start-domain --debug

The default listener on port is 9009 and suspend=n.

7. Jetty

   To start a Jetty application server just add debug arguments to java command:

   -agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=7001

Self-defined port number

$ java -jar sds.jar -port <number> -sourcepath ...

You can setup a different port number between the debugger (sds.jar) and Speare code editor from command line, but please keep the same with the port number configured in Speare.

Offline debug Java application

transport error 202: gethostbyname: unknown host

$ cp /etc/hosts ~/Desktop/

$ echo >> 127.0.0.1 username.local >> hosts

$ sudo mv hosts /etc/hosts

Note: "127.0.0.1" and "username.local" should be separated by a character '\t'.

Tools used to diagnose Java application

Out Of Memory!!

Configure JVM lanuch parameters

The following three JVM options specify initial and max heap size and thread stack size while running Java programs:

-Xms - set initial Java heap size
-Xmx - set maximum Java heap size
-Xss - set java thread stack size

Garbage Collection algorithms are used to attain better stability of Java application, Java provides 4 ways to implement garbage collection namely below:

-XX:+UseSerialGC
-XX:+UseParallelGC
-XX:+USeParNewGC
-XX:+UseG1GC

Using the following parameters to log the GC activity:

-XX:+UseGCLogFileRotation
-XX:NumberOfGCLogFiles=< number of log files >
-XX:GCLogFileSize=< file size >[unit]
-Xloggc:/path/to/gc.log

Tool Utilities

jmap - obtain a heap dump at runtime.
jhat - parses a heap dump file, show how many object allocated and the heap size.
jstack - stack dump a Java process.
jstat - display performance statistics for a JVM.
hprof - Heap and CPU profiling tool.

Switch Java Virtual Machine

You can directly switch the JVM and setup JAVA_HOME and CLASSPATH to fit your project.

Pretty Debug Output Printing

Speare allows keywords highlighting in console output, the colour definition file located in Speare.app/Contents/Resources/console.plist, you can directly add some keywords to support your special debugging output colour scheme.

Note: *valid colour value must a hex string that has seven characters, e.g, right click the editor → select "Colour Picker" then you can fetch such a HTML colour value conveniently.*

Extend Speare Code Editor for Java Development

Speare code editor support almost any type of Java project development, from Java Card, IoT, and cloud-connected devices, traditional classic J2ME (Java ME) embedded systems to J2SE desktop applications, and sever side big J2EE (Java EE) projects. You can directly run unit test, Git commands, and deploy commands for CI servers without leave editor. Speare code editor can be easily extended to run any dev tasks such as document generation, code style check (e.g. Checkstyle), code coverage tool (such as JaCoCo), Java static code analysis tools (e.g. PMD) and various CI servers such as CruiseControl, Jenkins, Hudson, Buildbot, Wercker, CircleCI, Codeship, etc. And you can integrate JNI native C and C++ code very conveniently in the same ultra lightweight dev environment with cross-programming-language references.

To add shell scripts to better debugging Java code in Speare code editor, please download the guide document and following the description in it.