Embedded Java IDE

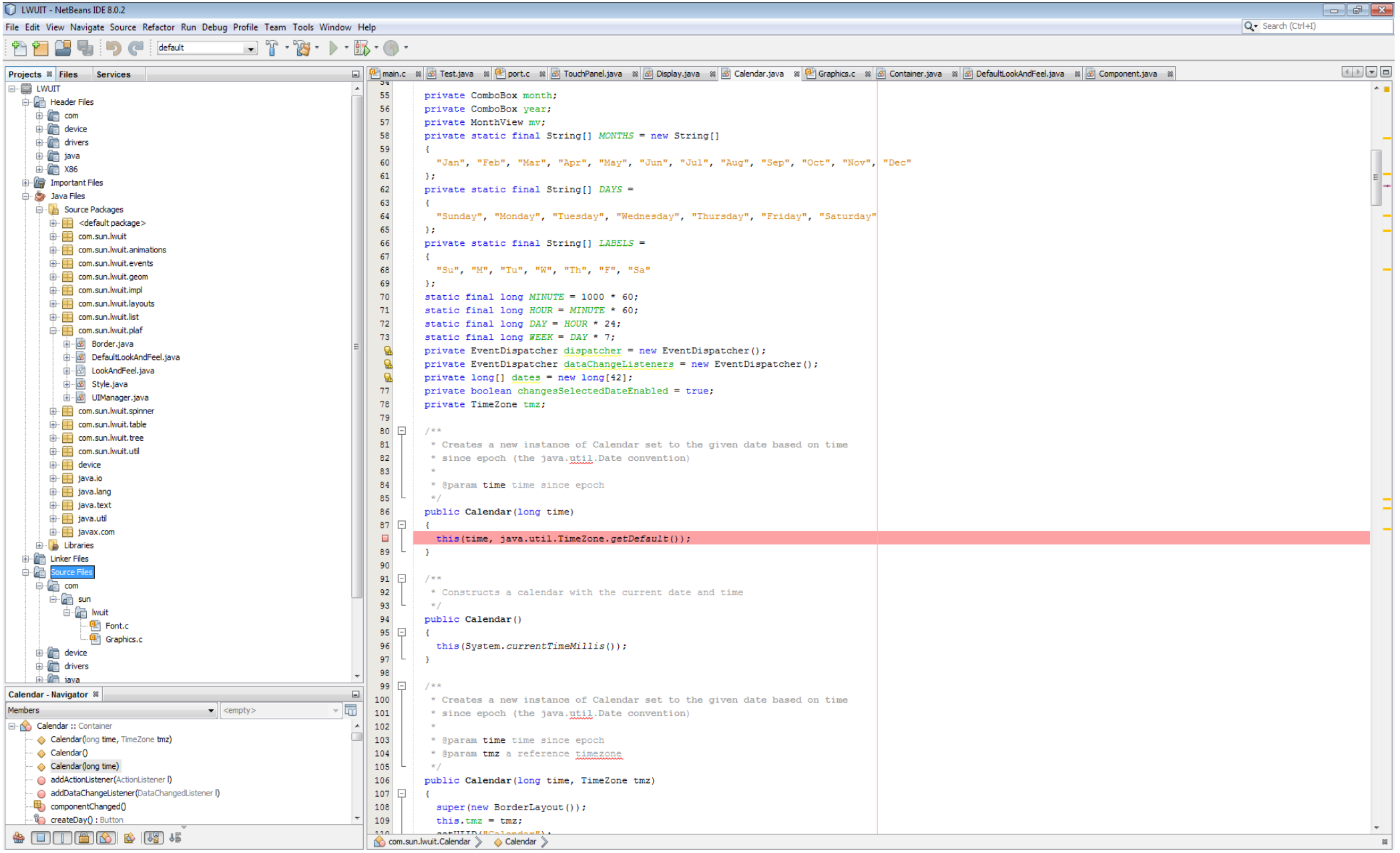
Embedded Java IDE in current version is a plugin for NetBeans 8 framework.

Embedded Java project

Embedded Java project is an extension of conventional NetBeans C language project, compatible upside down. We can see additional folder “Java Files”, which contains Java subproject with Java sources.

Also we can open Embedded Java projects in NetBeans IDE without plugin, in this case Java subfolder will be represented as ordinary file tree and during debugging all Java structures will be represented as their C counterparts.

Embedded Java project

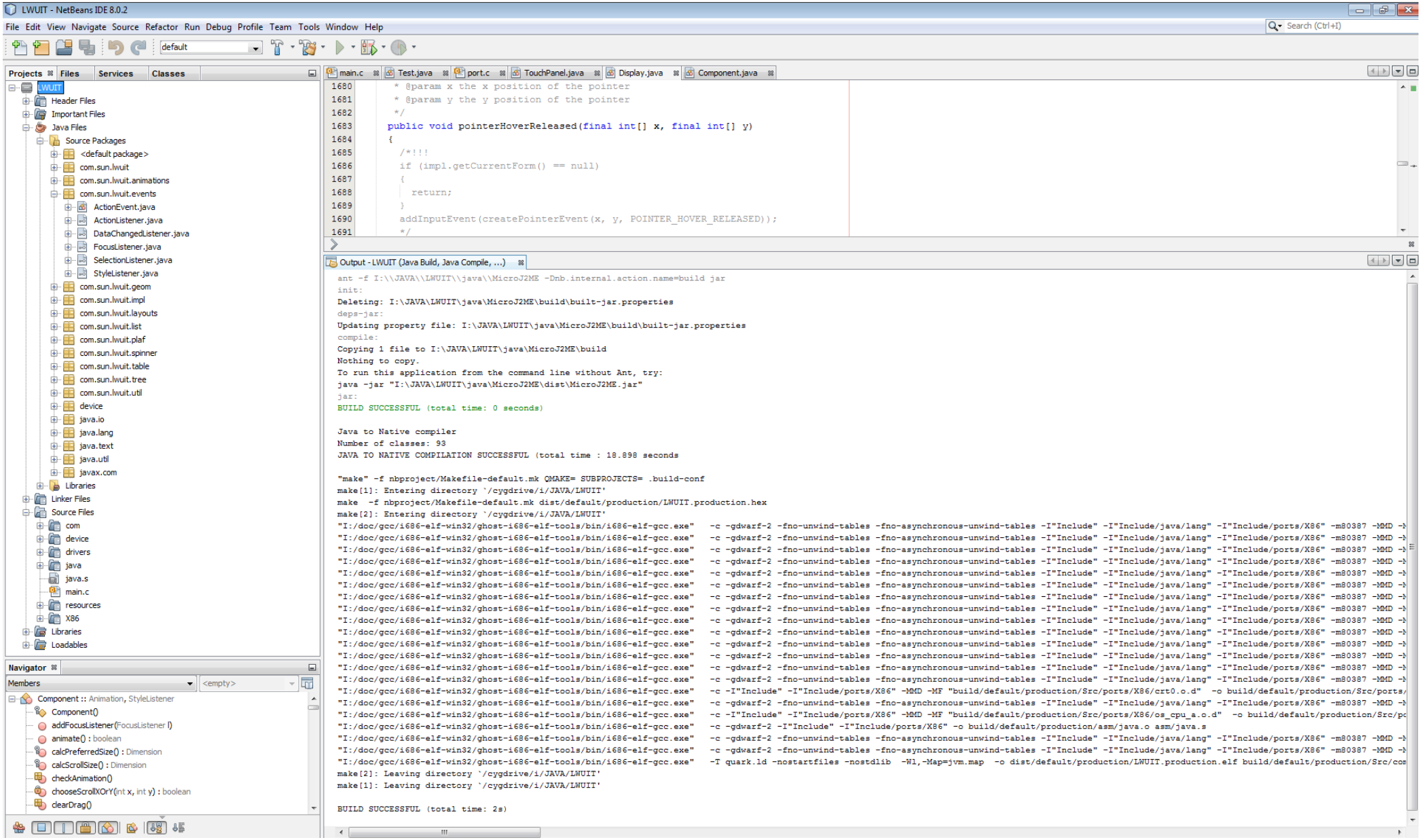


Project Build

Then we can build entire project. In this figure we can see all stages of building process:

- Build Java subproject
- Java to Native (X86 assembler) compilation
- Build Native (C & asm language subproject)

Project Build



Simulator or Board connection

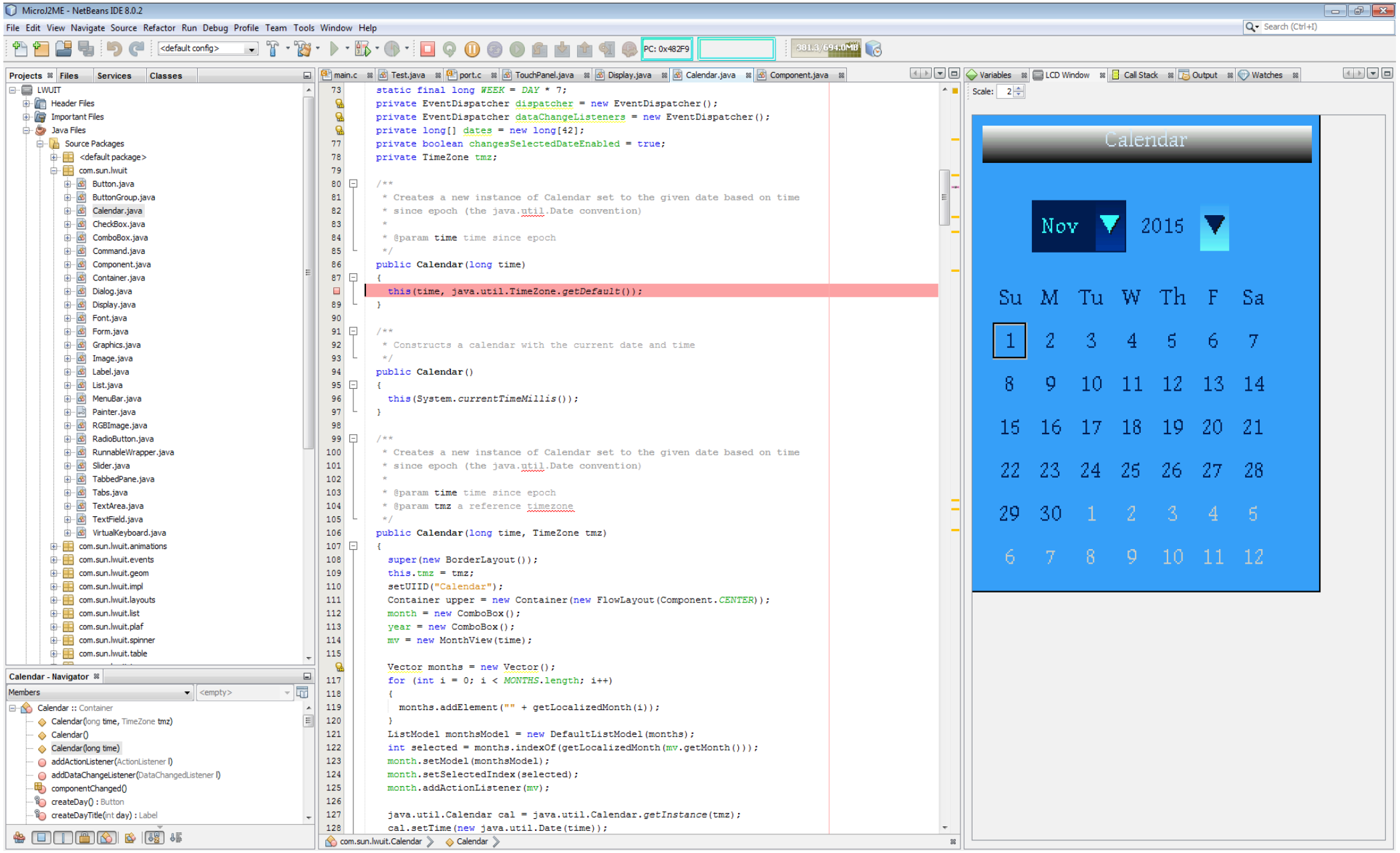
After build, simulator of X86 (or, later, real board with Curie) can be connected. *Current version supports simulator, but has all interfaces to JTAG debugging.*

Program loaded to simulator and running example.

We can see a form with Calendar widget from LWUIT, which contains different internal widgets, such as labels, buttons and combo boxes.

LCD simulates real display 320 to 240 pixels, which is driven by ILI9325 controller, connected to processor I/O bus.

Board or Simulator connection

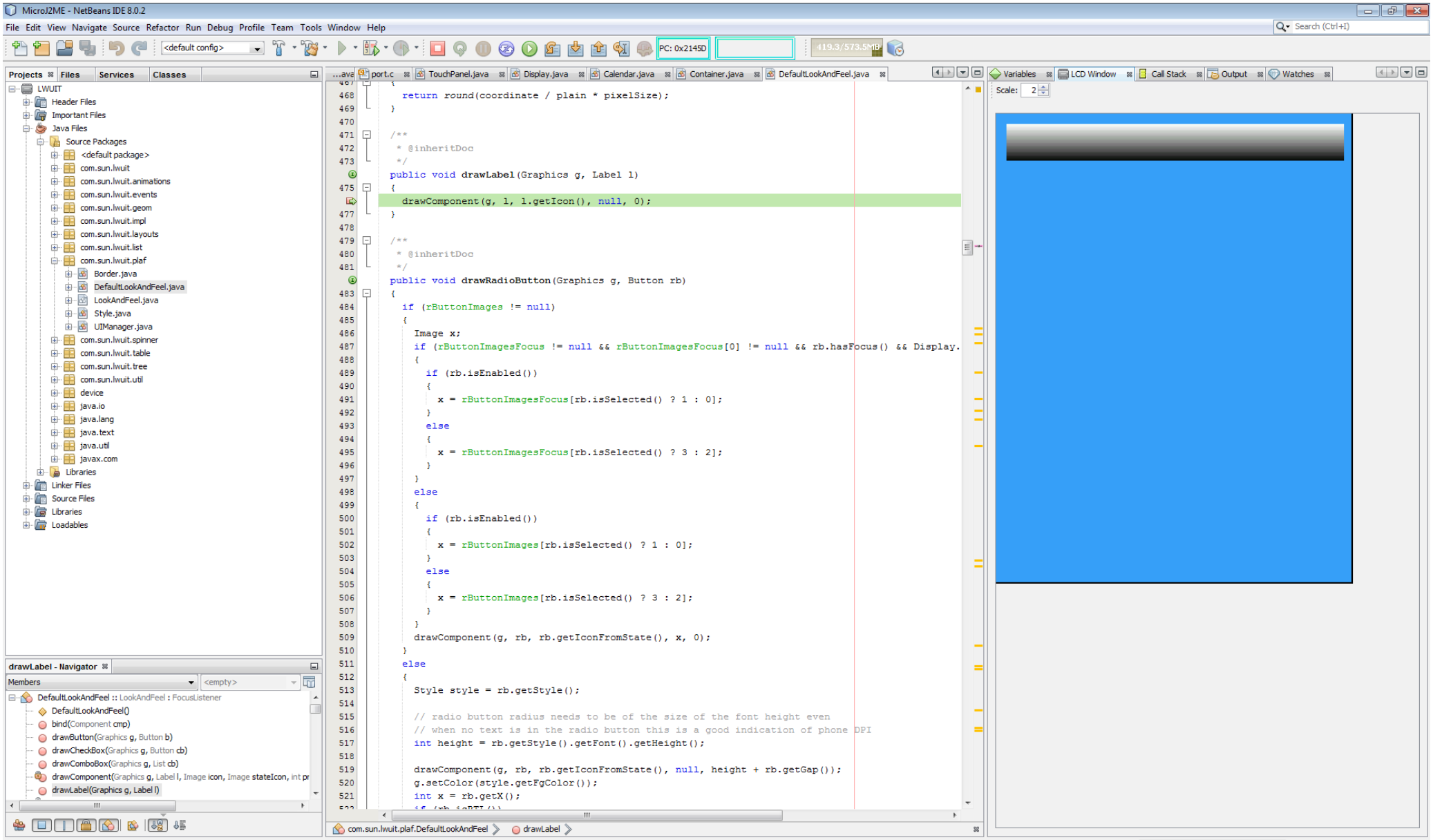


Stop on breakpoint

Now we can set a breakpoint and restart application.

Program stops on breakpoint during display form painting.

Stop on breakpoint

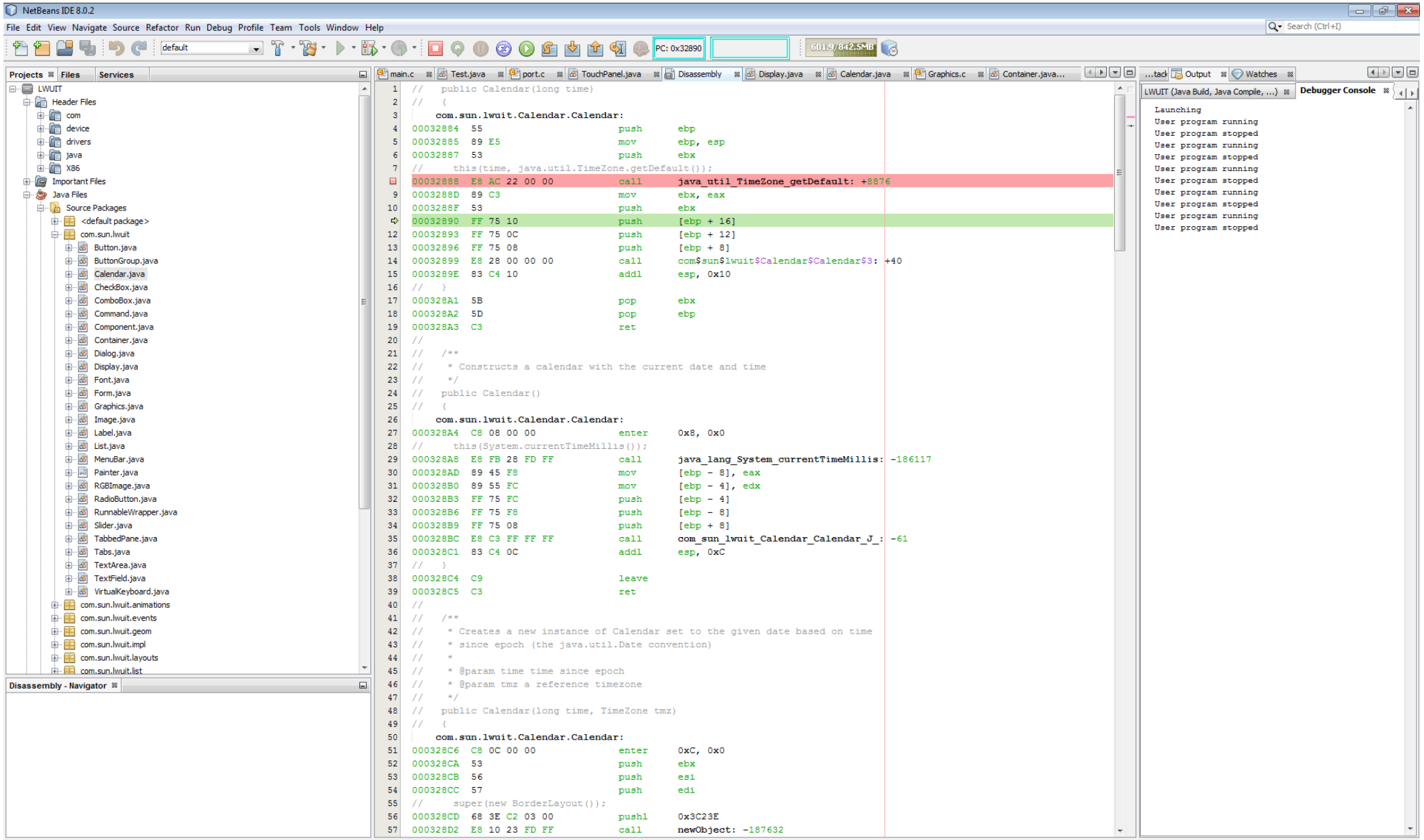


Disassembler Window

Now we can switch to Disassembler Window and see X86 assembler code *(this is code with maximum debugging capabilities, not for evaluation of compiler optimizations).*

In Disassembler Window we can toggle breakpoints of two different types – Java/C Source line breakpoints and Address breakpoints, system automatically choose which type of breakpoint will be toggled. If disassembly line has mapping from Java/C source, then Source line breakpoint will be added/removed, otherwise – Address breakpoint, which can be seen only in Disassembler of Breakpoints window

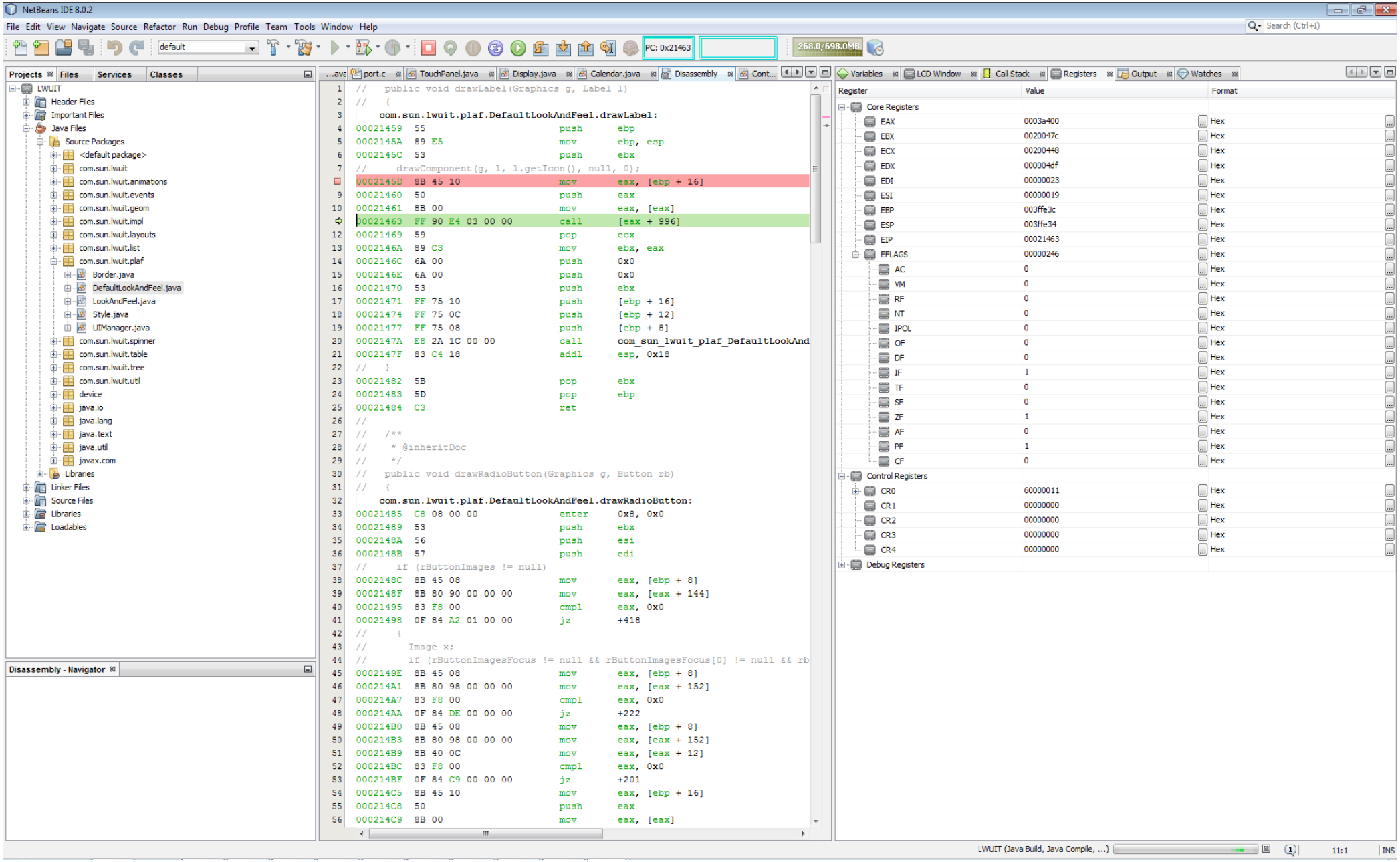
Disassembler Window



Processor registers window

Processor registers window contains nothing special. For conviniece is has a tree representation of different register groups (processor peripherals will be also included).

Processor registers window



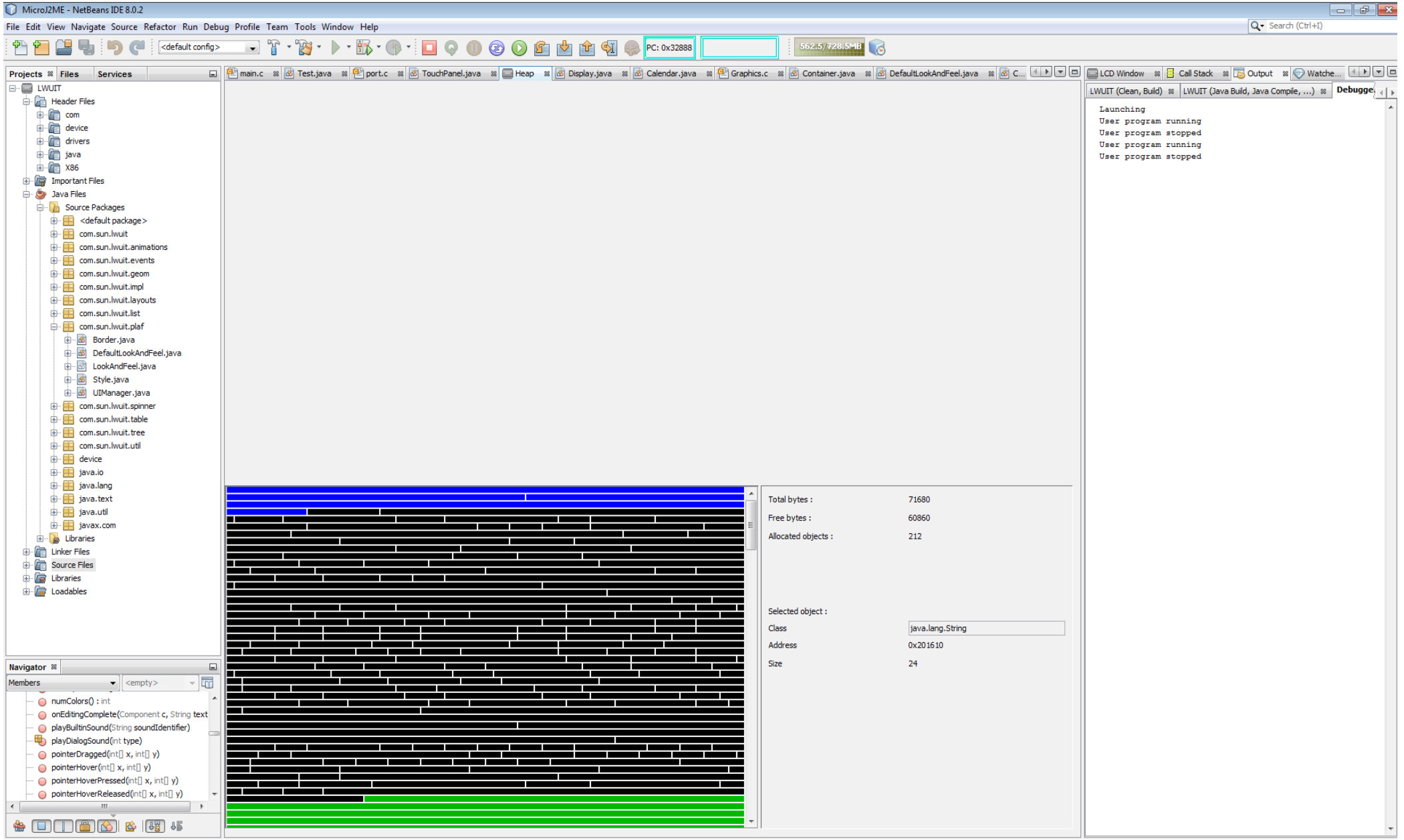
Runtime Heap Window

Some useful window, which represent runtime Heap.

- Green color – free memory blocks.
- Black color – dynamically allocated Java objects
- Blue color – special allocated regions (for example, processor thread stacks).

If we click on some black region, we can see some information of allocated object (address, size and Java Class of this object)

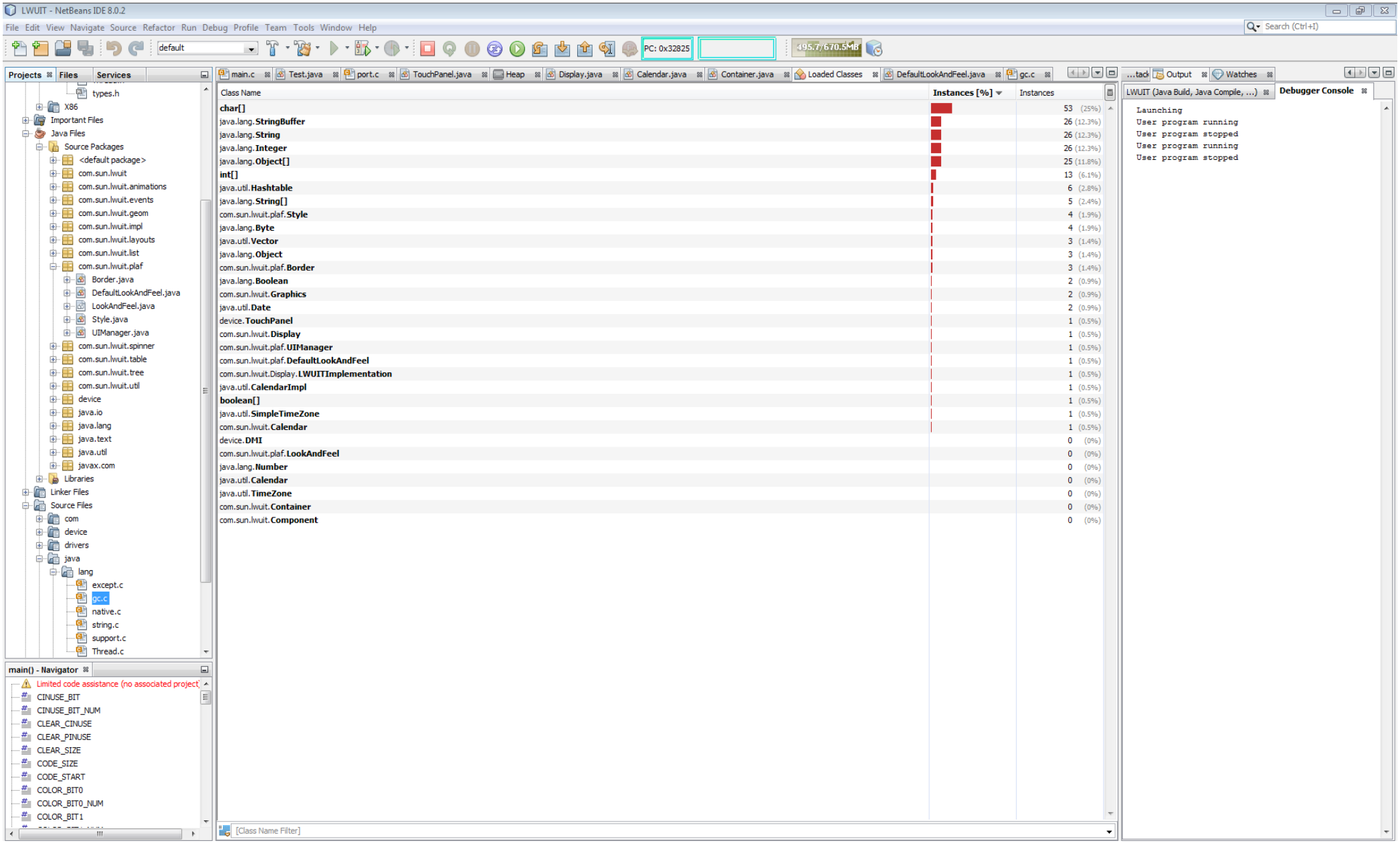
Runtime Heap Window



Runtime Status

Another useful information about runtime status. We can see how many object of which Java class are allocated in this execution point.

Runtime Status

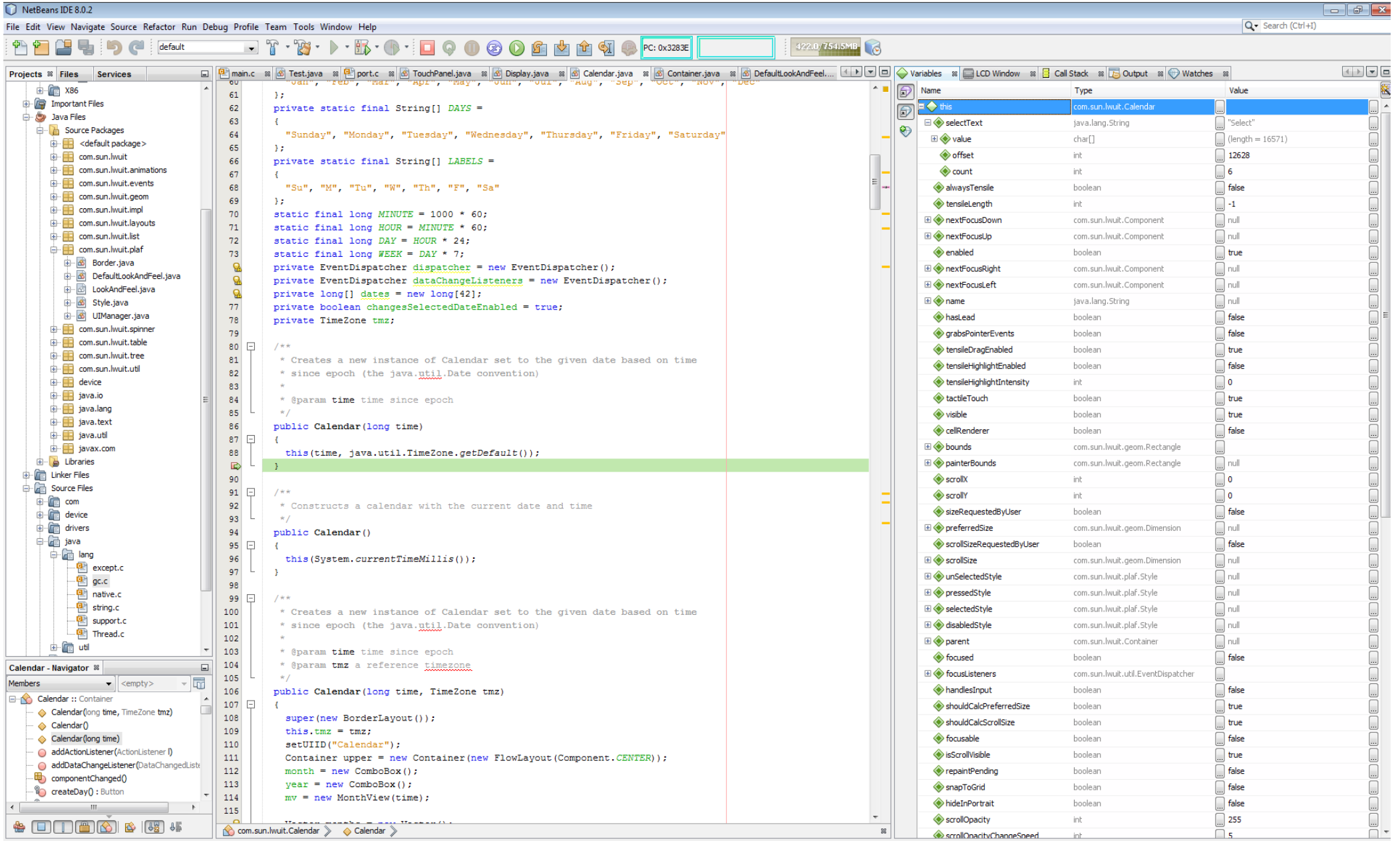


Class Name	Instances [%]	Instances
char[]		53 (25%)
java.lang.StringBuffer		26 (12.3%)
java.lang.String		26 (12.3%)
java.lang.Integer		26 (12.3%)
java.lang.Object[]		25 (11.8%)
int[]		13 (6.1%)
java.util.Hashtable		6 (2.8%)
java.lang.String[]		5 (2.4%)
com.sun.lwuit.plaf.Style		4 (1.9%)
java.lang.Byte		4 (1.9%)
java.util.Vector		3 (1.4%)
java.lang.Object		3 (1.4%)
com.sun.lwuit.plaf.Border		3 (1.4%)
java.lang.Boolean		2 (0.9%)
com.sun.lwuit.Graphics		2 (0.9%)
java.util.Date		2 (0.9%)
device.TouchPanel		1 (0.5%)
com.sun.lwuit.Display		1 (0.5%)
com.sun.lwuit.plaf.UIManager		1 (0.5%)
com.sun.lwuit.plaf.DefaultLookAndFeel		1 (0.5%)
com.sun.lwuit.Display.LWUITImplementation		1 (0.5%)
java.util.CalendarImpl		1 (0.5%)
boolean[]		1 (0.5%)
java.util.SimpleTimeZone		1 (0.5%)
com.sun.lwuit.Calendar		1 (0.5%)
device.DMI		0 (0%)
com.sun.lwuit.plaf.LookAndFeel		0 (0%)
java.lang.Number		0 (0%)
java.util.Calendar		0 (0%)
java.util.TimeZone		0 (0%)
com.sun.lwuit.Container		0 (0%)
com.sun.lwuit.Component		0 (0%)

Variables and Watches Windows

Variables and Watches Windows contain nothing special. System will show variable according with current execution point. If we stopped on Java source, all variables and watches will be Java types and classes. If we stopped on C source, there will be native C structures or native counterparts to Java classes.

Variables and Watches Windows

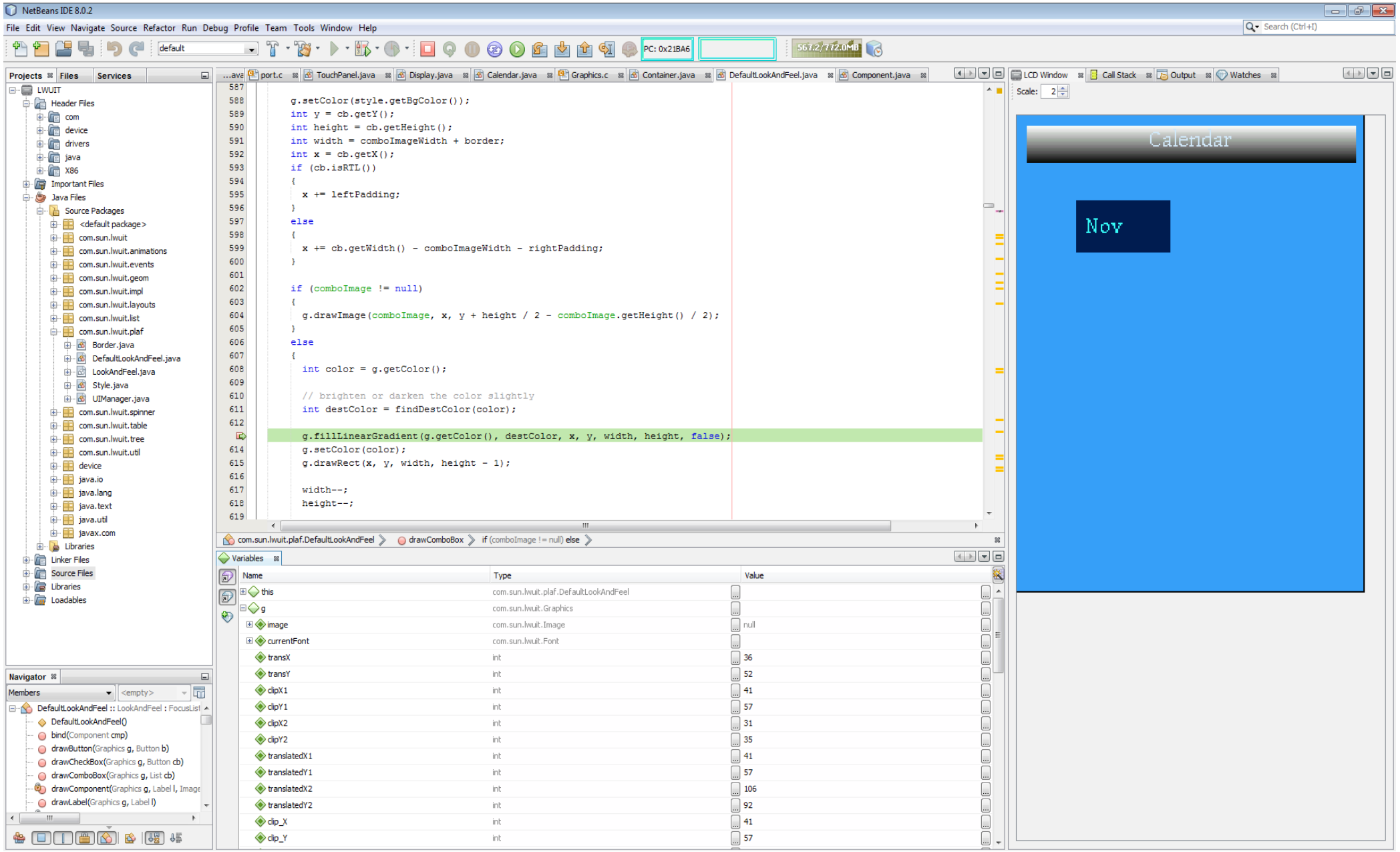


Native Methods Debugging

We stopped on Java native method implementation. Call Stack Window reflects this situation: it shows Java methods and also native methods and C functions.

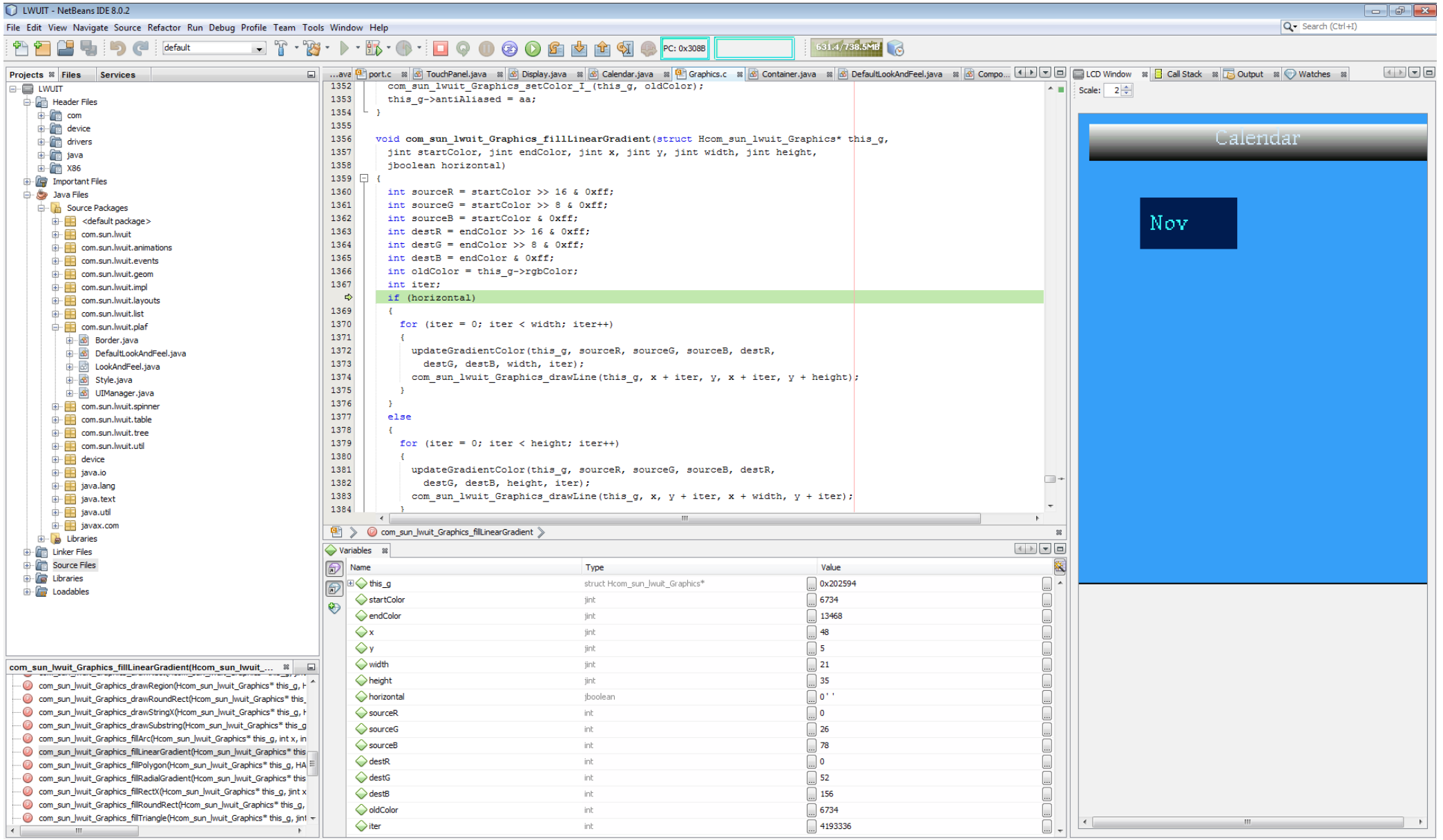
Native Methods Debugging

Before entering the native method



Native Methods Debugging

After entering the native method



Native Methods Debugging

Then we can open original Java class source from which its native method was implemented and see some methods in current calling sequence.

Two upper methods in calling stack are native methods, which are implemented on C and other methods are Java methods.

Native Methods Debugging