# Java HotSpot™ Client Compiler Visualizer

# User Guide

## Project Information

Summary: Visualization tool for the Java HotSpot™ client compiler
Project homepage: https://c1visualizer.dev.java.net
Project owner: Christian Wimmer
License: Common Development and Distribution License

## Quick Start

- Download and install the visualizer application.
- Download and install a DEBUG build of the JDK 6, JDK 7, or OpenJDK.
- Generate input data using the Java command line option `-XX:+PrintCFGToFile`.
- Load the generated `output.cfg` file into the visualizer application.
- Expand a method in the *Compiled Methods* view.
- Open an editor for the *Intermediate Representation*, *Bytecodes*, *Control Flow Graph*, *Data Flow Graph*, or *Intervals* of a compilation phase.
- Use the detail views to see additional information for the element that is currently selected in the active editor.

## Installation

A Java SE 6 Runtime Environment (JRE) or Development Kit (JDK) or higher must be installed on your system. Any JRE for Java 6 or higher can be used to run the visualizer application, the application itself does not depend on the Java HotSpot™ VM.

The Java HotSpot™ Client Compiler Visualizer application is based on the NetBeans Platform 6.0. It is distributed as a single zip-file with the appropriate launchers for all supported platforms. Extract the zip-file and start the executable in the subdirectory `/bin` for your platform, e.g. `c1visualizer.exe` on Microsoft Windows.

## Generating Input Data

The application visualizes the internal data structures of the Java HotSpot™ client compiler, a just-in-time compiler inside the Java HotSpot™ VM. The compiler can be configured to dump debug information into a file while the VM is running. This feature is not included in the product version of the VM, therefore a DEBUG build must be used. You can download the DEBUG build for your platform (or download the source code and build a DEBUG version yourself) from the following locations:

- JDK 6: http://download.java.net/jdk6/binaries/
- JDK 7: http://download.java.net/jdk7/binaries/
- OpenJDK: http://openjdk.java.net/

The DEBUG builds contain the `java` executable in the directory `/fastdebug/bin`. In contrast to the product builds, you can specify additional command line options that are useful for debugging and testing the VM.

The command line option `-XX:+PrintCFGToFile` activates the dump of the debug information. A file named `output.cfg` is created in the directory of your Java application that can be loaded into the visualizer application. The following table contains other useful command line options:

| Option | Description |
|---|---|
| -XX:+PrintCFGToFile | Enables the dumping of debug information. |
| -XX:+PrintCompilation | Prints the name of all methods that are compiled to the console. |
| -XX:+PrintInlining | Prints the name of all methods that are inlined into a method to the console. |
| -XX:-Inline | Deactivates method inlining. |
| -Xcomp | All methods that are executed are compiled. Normally, only few important methods are compiled, therefore this increases the number of compiled methods greatly. |
| -XX:CompileOnly=<class>[.<method>] | Limits the compilation to classes that match the pattern. It is possible to append a list of such patterns. Packages are separated by "/", not by "." |

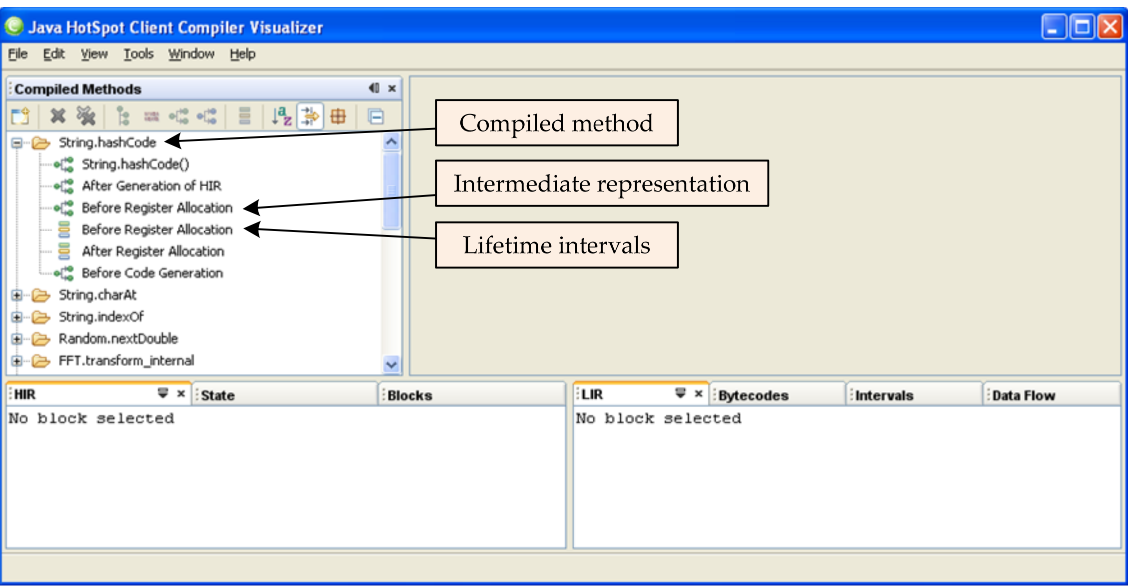
The examples in this user guide show the methods that are compiled when running the SciMark 2.0 benchmark (available from http://math.nist.gov/scimark2/). The input file was created using the following command line:

```
java -XX:+PrintCFGToFile jnt.scimark2.commandline
```

You can also download this sample file from the project homepage.

## Loading Input Data

Use *File / Open Compiled Methods* to load an `output.cfg` file. The new methods are added to the *Compiled Methods* view (usually on the left side of the window). The first level of the three contains the methods that were compiled. The second level shows a snapshot of the compiler data structures in different compilation phases.




Usually, the following states are produced (compiler developers can easily add their own states when working on the compiler):

- *Bytecode parsing*: The first state contains the bytecodes and the control flow graph of a method when the bytecodes are parsed by the compiler. The name of the method and the type of the parameters are shown in the list.
- *Inlined methods*: If the compiler inlined methods, the bytecodes and the control flow graph of the inlined methods are shown equally to the first method.
- *After Generation of HIR*: During compilation, two intermediate representations are used before the final machine code is generated. The first one is called *high-level intermediate representation* (HIR). Normally one HIR instruction is generated for one Java bytecode This state shows the complete HIR after all bytecodes have been parsed.
- *Before Register Allocation*: This state is generated after global optimizations on the HIR have been performed. Also the *low-level intermediate representation* (LIR), which is close to machine code, is already available in this state.
- *Before Register Allocation (Lifetime intervals)*: A special view on the data structures used during linear scan register allocation.

- *After Register Allocation (Lifetime intervals)*: Displays the register allocation data after physical registers are assigned to the virtual ones. It also shows which values are spilled, i.e. temporarily stored in memory.
- *Before Code Generation*: This last step shows the LIR just before machine code is generated.

Depending on the selected element, different editors can be opened using the context menu or the toolbar of the *Compiled Methods* view. The editors are explained in detail in the following sections.

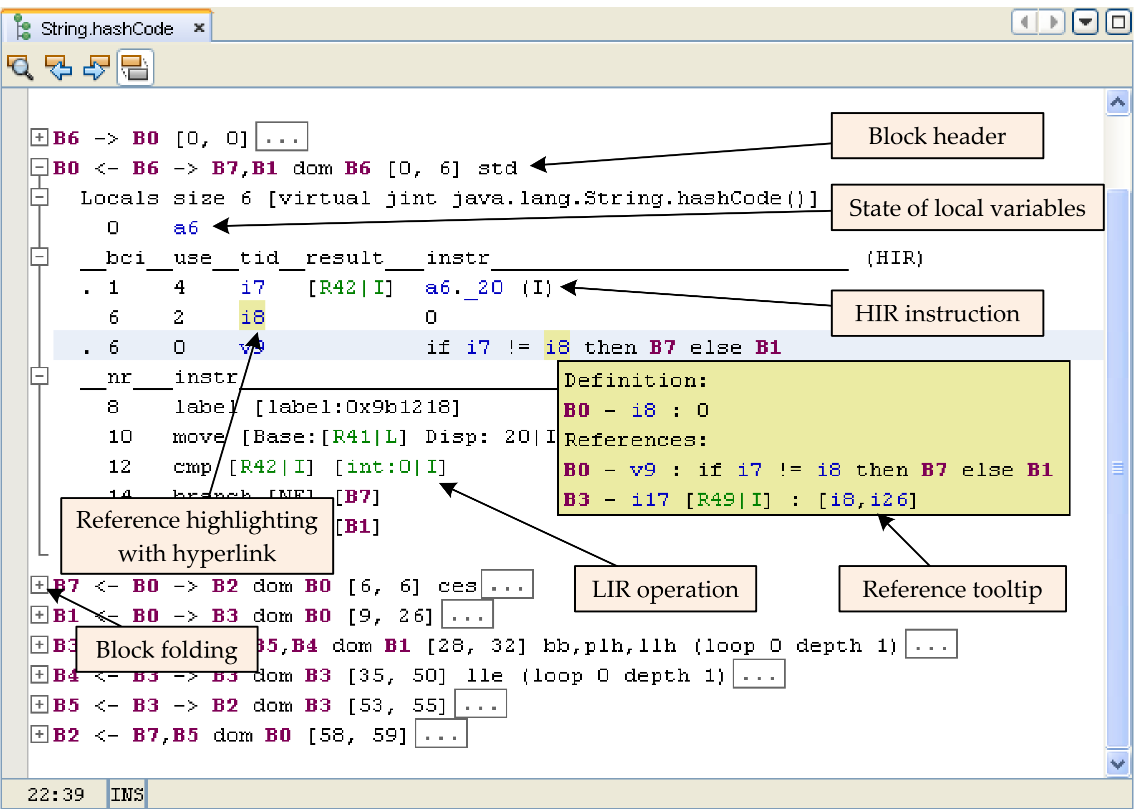
- *Intermediate Representation*: Textual view of the currently available intermediate representations.
- *Bytecodes*: Textual view of the bytecodes.
- *Control Flow Graph*: Graphical view of the control flow. The control flow in all compilation states, i.e. for the bytecodes, the HIR and the LIR.
- *Data Flow Graph*: Graphical view of the data flow. It is only available for the HIR.
- *Intervals*: Graphical view of the lifetime intervals, which are the main data structure of the linear scan register allocator.

The remaining icons in the toolbar of the *Compiled Methods* view format the list of compiled methods:

- Sort the list of methods (the first level of the tree) alphabetically.
- Hide inlined methods that are short and consist only of a single block because such methods are usually not of interest.
- Show the fully qualified package name for all classes.

## Intermediate Representation

This editor displays the HIR and the LIR in textual form. It can be viewed as a textual console output enhanced with syntax coloring and navigation. The following figure shows the intermediate representation of the method `String.hashCode()` in the state *Before Register Allocation*. At this state both, HIR and LIR are available, so this is usually the state with the most valuable information.

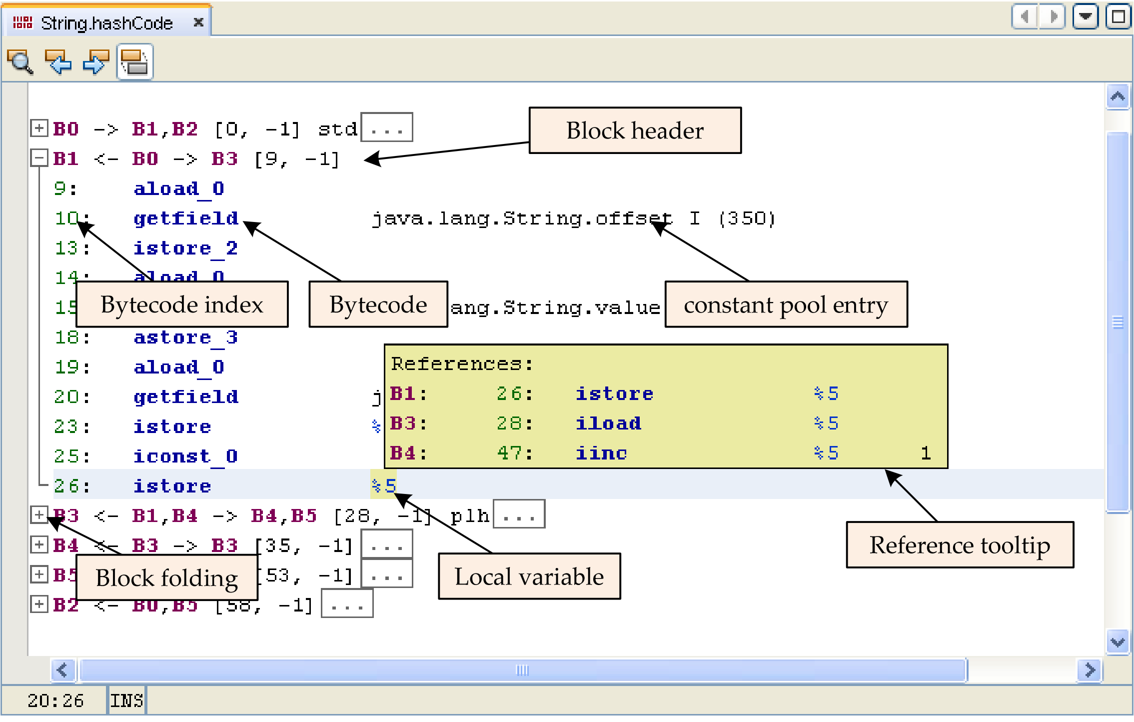



The intermediate representation is grouped in basic blocks, i.e. longest possible sequences of instructions without jumps or jump targets in the middle. The first line of each block contains information about the connections to other blocks as well as a set of flags for this block. This line is followed by up to three subsections that show the state of the local variables at the beginning of the block, the HIR of the block and the LIR of the block. Depending on the compilation state, only the HIR, only the LIR or both the HIR and LIR are available. Block folding, syntax coloring, tooltips, highlighting of name references and hyperlink navigation improve the readability and navigation in the editor.

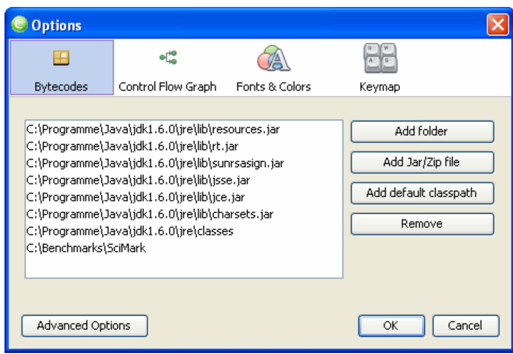
The views *State*, *HIR*, and *LIR* show the same textual information for the currently selected blocks. This information is useful when working with other editors. When blocks are e.g. selected in the control flow graph editor, the textual intermediate representation is available via these views.

## Bytecodes

This editor displays the bytecodes in textual form. The bytecodes are only available in the early states of the compilation, i.e. during bytecode parsing. After method inlining, the control flow graph and the intermediate representation contain information from all inlined methods, therefore a mapping back to the bytecodes is neither possible nor useful. Similarly to the intermediate representation editor, block folding, syntax coloring, tooltips, highlighting of name references and hyperlink navigation are available.

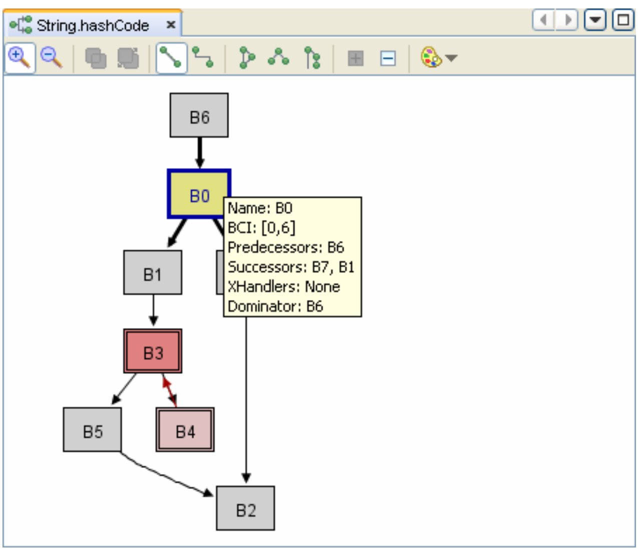



The bytecodes are not part of the dumped debug information in the `output.cfg` file. The file contains only the method name, so the class files must be available for the visualizer application. Use the preferences dialog available via the menu *Tools / Options* to configure the classpath for your application. For example, the following figure shows the correctly configured classpath for the SciMark benchmarks. The base folder of the benchmark, which contains the subfolders with the Java packages, is added to the default list that represents the Java boot classpath.



## Control Flow Graph

The graphical visualization of the control flow graph is the best way to get a structural overview of a method. The nodes of the graph are the basic blocks, the edges are possible control flow paths. Blocks have different colors depending on which flags are set for the blocks. For examples, loop headers and exception handlers are marked with special colors. This automatic coloring can be overridden manually using the *color* button of the toolbar.




To make loops easier to detect, the loop depth of a block affects the drawing of a block too. The border of the blocks `B3` and `B4` is a double line, meaning loop depth one. All other blocks in this example have loop depth zero and therefore the border is just a single line. Backward edges are edges going from a loop end to the loop header. They have a special color. In the example there is only one backward edge going from block `B4` to `B3` and it is painted red.

For drawing the edges, there are two possible modes available: The first mode (Button ) draws straight edges whenever possible, otherwise it tries to evade any obstacles and paints the edges as Bezier curves. The other one (Button ) is called *Manhattan Router* and makes all connections orthogonal. Three different automatic block positioning algorithms (Buttons , , and ) optimize different criteria like the overall size of the graph or the highlighting of loops.

The visualization can be modified manually in several ways to get a better overview. First of all, the blocks can be dragged using the left mouse button. There exist several other operations on blocks which can be applied either using the toolbar buttons or the context menu:

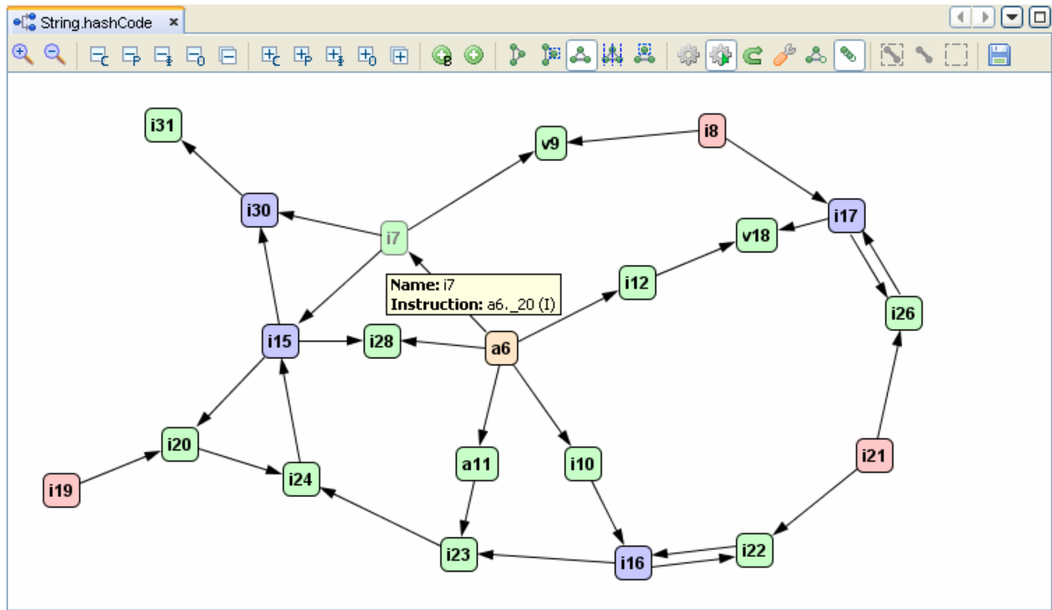
- *Combine*: An important functionality when analyzing large graphs is combining several blocks to a single node. This way unimportant parts can be removed from the graph.
- *Split*: This is the opposite operation to combining nodes. Instead of the combined node, all original nodes are restored.

*Show/hide edges*: Some blocks, especially exception handler blocks, have many incoming or outgoing edges, which can make the graph look bad. By hiding those edges, the overview can be improved.

*Color*: The automatic coloring of the nodes depending on flags can be manually overwritten to mark certain parts of the graph.

## Data Flow Graph

In the graphical visualization of the data flow, HIR instructions are shown as nodes. The edges represent data dependencies, i.e. an edge goes from the usage of a value to its definition. Data flow graphs for large methods are big and difficult to understand. Therefore, the editor is accompanied by a view that lists all instructions, and it is possible to reduce the amount of data shown in the editor. Instructions can be hidden and shown using the toolbar, the context menu, or the checkboxes in the view.




Normally, only the instruction numbers are shown in the graph. Only when a node is expanded, the details like the instruction string and the Block are visible. Use the context menu to expand and collapse nodes. Clustering of nodes brings control flow information into the data flow graph. When clustering is enabled, instructions of the same block are located as close as possible.

The toolbar offers several possibilities to customize the view of the editor:

- The first five buttons are used to hide groups of nodes. It is possible to hide constants, parameters, phi-functions and operations. Additionally, one may hide all nodes at once.
- The second five buttons can make groups of nodes visible. Each button has its counterpart within the previous five ones.
- Expansion of block-data and instruction-string data can be handled using the next two buttons. These can be toggled. The expansion is applied to the whole graph.
- To select the layout algorithm, one of the following buttons has to be pressed. The first two ones activate hierarchical layouts. The third, fourth and fifth apply force-directed layouts. The dashed blue rectangles shown in the icons two and four indicate that these layout algorithms support clustering.

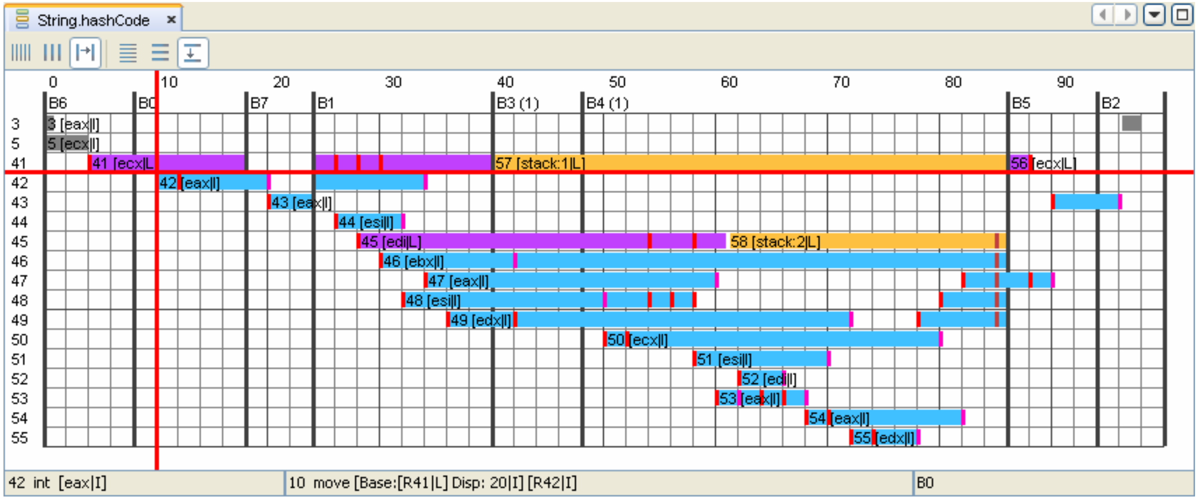
- The gearwheel represents the layout-calculation process. Pressing the first of the two buttons forces an immediate layout cycle. The second one is a toggle-button that enables auto-layout cycles after each user interaction. This option may be deactivated if a big graph has to be processed.
- The bent arrow symbolizes the re-usage of node positions. Some layout algorithms use an optimizing approach starting with a random node arrangement. If this option is activated, the layout is calculated based on the current node positions instead.
- To configure the current layout algorithm, the wrench-button can be used. This brings up a dialog with all exposed layout options.
- The next toggle-button decides if invisible nodes are included during the layout calculation. This option is helpful to keep the overview if a graph is built from nothing, because new nodes do not influence the positions of previously visible ones.
- The last option within this group decides if node animation is used. This means that the movement of a node is animated using linear interpolation.
- The next three options are only available if clustering is supported by the current layout algorithm. The first of them activates highlight clustering, i.e. the clustering of the currently expanded nodes. The next option decides if links between different clusters are grayed. The third is a toggle-button that enables or disables visible cluster borders.
- The last button can be used to export the currently visible graph to various file-formats, including vector graphics and graph representations.

Functions that operate on the currently selected nodes are available in the context menu of the editor or the view. They can be used to show and hide the selected nodes, to expand and collapse the nodes, and to navigate through the graph.

## Intervals

This editor displays the data structures used during register allocation. The client compiler uses the linear scan algorithm. The control flow graph is flattened to a list. All LIR operations are numbered using the block order and shown on the x-axis, so every column corresponds to one LIR operation. The numbers of the virtual registers, which are also the numbers of the intervals, are shown on the y-axis. A row with a graphical view of the lifetime interval is painted for every virtual register.

During register allocation, intervals can be split into several shorter ones. A new virtual register is assigned to each newly created interval, but shown in the same line to improve the readability. Additionally, a physical register is assigned to every interval. If there is no register available, the value needs to be stored temporarily in memory, and a stack slot number is assigned to the interval.




Two snapshots of the interval are available for each method: *Before Register Allocation* shows the initial state of the intervals after they have been created. No interval has been split yet, so every line contains one interval. The color of a bar indicates the type of its associated value, e.g. magenta for objects and blue for integer values. *After Register Allocation* shows the intervals after the register allocator has assigned a physical register to each interval. It is also possible that long intervals are split into several shorter ones. In the example below, the interval for the virtual register 41 was split into the three parts 41, 57 and 56. Physical registers like `eax` or `ebx` are assigned to the intervals.

Orange color is used to indicate that a value needs to be stored in memory for a while, which is also called *spilling*. The register allocator avoids spilling whenever possible and also optimizhes the position where the intervals are split and spilled. In this example, the value 41 is not accessed within the loop (which consists of the blocks `B3` and `B4`), therefore it is a good decision to spill this value, as it needs to be loaded from memory only once after the end of the loop and not at every loop iteration.