

Main

[blocked URL](#)

Project Loom is intended to explore, incubate and deliver Java VM features and APIs built on top of them for the purpose of supporting easy-to-use, high-throughput lightweight concurrency and new programming models on the Java platform.

This [OpenJDK](#) project is sponsored by the [HotSpot Group](#).

Source Code

<https://github.com/openjdk/loom>
Early Access Binaries

<http://jdk.java.net/loom/>
Resources

[JEP 425: Virtual Threads \(Preview\)](#)

[JEP 428: Structured Concurrency \(Incubator\)](#)

[On the Performance of User-Mode Threads and Coroutines](#)

More on [inside.java](#)

Outdated:

[State of Loom](#)

Talks

[Philly ETE 2021 - Video](#)

[Code Mesh 2020 - Video](#)

[Joker 2020 - Video](#)

[AccentoDev 2020 - Video](#)

[Devoxx BE 2019 - Video](#)

[JVMLS 2019 - Video](#)

[Curry On 2019 - Video](#)

[QCon London 2019 - Video and Slides](#)

[FOSDEM 2019 - Video](#)

[Devoxx BE 2018 - Video | Slides](#)

[JVMLS 2018 – Video | Slides](#)

[JFokus 2018 – Video](#)

Meetings

[October 2018 - Slides](#)


Mailing List

[Subscribe](#) | [Archive](#)

Project

[Proposal](#) | [JEP](#) | [Members](#) | [Page](#)

Note

 Loom is under active development, which means that information and advice given here might change in the future.

Supported Platforms

Mac and Linux on x86-64

Download and Build from Source

```
$ git clone https://github.com/openjdk/loom
$ cd loom
$ git checkout fibers
$ sh configure
$ make images
```

How to Contribute

The most valuable way to contribute at this time is to try out the current prototype and provide feedback and bug reports to the loom-dev mailing list. In particular, we welcome feedback that includes a brief write-up of experiences adapting existing libraries and frameworks to work with Fibers.

If you have a login on the JDK Bug System then you can also submit bugs directly. We plan to use an Affects Version/s value of "repo-loom" to track bugs.

How to run the JDK tests

1. Download `jtreg` (the JDK test harness) and place its `bin` subdirectory on your path.
2. Create a debug JDK configuration (inside the top directory of the Loom repo) and build it. This step requires having `jtreg` on your path, or running the tests would fail:

```
$ sh configure --with-jtreg --with-debug-level=fastdebug
$ make images
```

3. Run the tests. The following example assumes a Mac build (replace `macosx` with `linux` for a Linux build), and the `java/lang/Continuation/Basic.java` test, which contains some basic Continuation tests. The `java/lang/Continuation` directory contains Continuation test, while the `java/lang/Continuation` directory contains fiber tests. Supplying just the directory name runs all tests in the directory.

```
$ make run-test TEST=open/test/jdk/java/lang/Continuation/Basic.java CONF=macosx-x86_64-server-fastdebug
```

Virtual Threads

Design

See [JEP 425: Virtual Threads \(Preview\)](#)

Implementation

Virtual threads are implemented in the core libraries. A virtual thread is implemented as a continuation that is wrapped as a task and scheduled by a `java.util.concurrent.Executor`. Parking (blocking) a virtual thread results in yielding its continuation, and unparking it results in the continuation being resubmitted to the scheduler. The scheduler worker thread executing a virtual thread (while its continuation is mounted) is called a *carrier* thread.

The continuations used in the virtual thread implementation override `onPinned` so that if a virtual thread attempts to park while its continuation is pinned (see above), it will block the underlying carrier thread.

The implementation of the networking APIs in the `java.net` and `java.nio.channels` packages have as been updated so that virtual threads doing blocking I/O operations park, rather than block in a system call, when a socket is not ready for I/O. When a socket is not ready for I/O it is registered with a background multiplexer thread. The virtual thread is then unpacked when the socket is ready for I/O.

Debugging

See the [Virtual Thread Debugging Support](#) page.

Continuations

Design

The primitive continuation construct is that of a `scoped` (AKA multiple-named-prompt), stackful, one-shot (non-reentrant) delimited continuation. To implement reentrant delimited continuations, we could make the continuations cloneable. Continuations aren't exposed as a public API, as they're unsafe (they can change `Thread.currentThread()` mid-method). However, higher level public constructs, such as virtual threads or (thread-confined) generators will make internal use of them.

Tail Calls

Design

We envision explicit tail-call elimination. It is not the intention of this project to implement *automatic* tail-call optimization.