Getting started

The easiest way to get started is to configure your IDE to use a recent Project Loom Early Access (EA) build and get familiar with using the java.lang.Thread API to create a virtual thread to execute some code. Virtual threads are just threads that are scheduled by the Java virtual machine rather than the operating system. Virtual threads are suited to executing code that spends most of its time blocked, maybe waiting for a data to arrive on a network socket. Virtual threads are not suited to running code that is compute bound.

Many applications won't use the Thread API directly but instead will use the java.util.concurrent.ExecutorService and Executors APIs. The Executors API has been updated with new factory methods that create ExecutorService to start a thread for each task. Virtual threads are cheap enough that a new virtual thread can be created for each task, there should never be a need to pool virtual threads.

Thread API

The following uses a static factory method to start a virtual thread. It invokes the *join* method to wait for the thread to terminate.

```
Thread thread = Thread.startVirtualThread(() -> System.out.println("Hello"));
thread.join();
```

The following is an example that creates a virtual thread that puts an element into a queue when it has completed a task. The main thread blocks on the queue, waiting for the element.

```
        var queue = new SynchronousQueue<String>();

        Thread.startVirtualThread(() -> {
            try {
                Thread.sleep(Duration.ofSeconds(2));
                queue.put("done");
            } catch (InterruptedException e) { }
        });

        String msg = queue.take();
```

The Thread.Builder API can also be used to create virtual threads. The first snippet below creates an *un-started thread*. The second snippet creates and starts a thread with name "bob".

```
Thread thread1 = Thread.builder().virtual().task(() -> System.out.println("Hello")).build();

Thread thread2 = Thread.builder()
                       .virtual()
                       .name("bob")
                       .task(() -> System.out.println("I'm Bob!"))
                       .start();
```

The Thread.Builder API can also be used to create a ThreadFactory. The ThreadFactory created by the following snippet will create virtual threads named "worker-0", "worker-1", "worker-2", ...

```
ThreadFactory factory = Thread.builder().virtual().name("worker", 0).factory();
```

Executors/ExecutorService API

The following example uses the Executors API to create an ExecutorService that runs each task in its own virtual thread. The example uses the try-with-resources construct to ensure that the ExecutorService has terminated before continuing. The example demonstrates the use of the submit methods (these methods do not block), and the invokeAll/invokeAny combinator methods that execute several tasks and wait them to complete.

```
        try (ExecutorService executor = Executors.newVirtualThreadExecutor()) {

            // Submits a value-returning task and waits for the result
            Future<String> future = executor.submit(() -> "foo");
            String result = future.join();

            // Submits two value-returning tasks to get a Stream that is lazily populated
            // with completed Future objects as the tasks complete
            Stream<Future<String>> stream = executor.submit(List.of(() -> "foo", () -> "bar"));
            stream.filter(Future::isCompletedNormally)
                    .map(Future::join)
                    .forEach(System.out::println);

            // Executes two value-returning tasks, waiting for both to complete
            List<Future<String>> results1 = executor.invokeAll(List.of(() -> "foo", () -> "bar"));

            // Executes two value-returning tasks, waiting for both to complete. If one of the
            // tasks completes with an exception, the other is cancelled.
            List<Future<String>> results2 = executor.invokeAll(List.of(() -> "foo", () -> "bar"), true);

            // Executes two value-returning tasks, returning the result of the first to
            // complete, cancelling the other.
            String first = executor.invokeAny(List.of(() -> "foo", () -> "bar"));

        }
```