Getting started

The easiest way to get started is to configure your IDE to use a recent JDK 19 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 JDK rather than the operating system. Virtual threads are best suited to executing code that spends most of its time blocked, waiting for data to arrive on a network socket or waiting for an element in queue for example.

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 for ExecutorServices that start a new 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 starts a virtual thread to print a message. It invokes the *join* method to wait for the thread to terminate.

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

The following is an example that start a virtual thread to put an element into a queue after sleeping. The main thread blocks on the queue, waiting for the element.

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

        Thread.ofVirtual().start(() -> {
            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 a ThreadFactory. The ThreadFactory created by the following snippet will create virtual threads named "worker-0", "worker-1", "worker-2", ...

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

Executors/ExecutorService API

The following example uses the Executors API to create an ExecutorService that starts a new virtual thread for each task. The example uses the try-with-resources construct to ensure that the ExecutorService has terminated before continuing.

ExecutorService defines submit methods to execute tasks for execution. The submit methods don't block, instead they return a Future object that can be used to wait for the result or exception. The submit method that takes a collection of tasks returns a Stream is lazily populated with completed Future objects representing the results.

The example also uses the invokeAll and invokeAny combinator methods to 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"), /*waitAll*/
false);

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

        }
```