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.

In addition to to the Thread API, the java.util.concurrent.ExecutorService and Executors APIs are have been updated to make it easy to work with virtual threads. Virtual threads are cheap enough that a new virtual thread can be created for each task, no need for pooling of 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 Thread.Builder API can also be used to create virtual threads that are configured at build time. 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();
```

ExecutorService API

The following creates an ExecutorService that runs each task in its own virtual thread. The example uses the *try-with-resources* construct to ensure that the ExecutorService is shutdown and that the two tasks (each run in its own virtual thread) complete before continuing.

```
try (ExecutorService executor = Executors.newVirtualThreadExecutor()) {
    executor.execute(() -> System.out.println("Hello"));
    executor.execute(() -> System.out.println("Hi"));
}
```

The example below runs three tasks and selects the result of the first task to complete. The remaining tasks are cancelled, which causes the virtual threads running them to be interrupted.

```
try (ExecutorService executor = Executors.newVirtualThreadExecutor()) {
    Callable<String> task1 = () -> "foo";
    Callable<String> task2 = () -> "bar";
    Callable<String> task3 = () -> "baz";
    String result = executor.invokeAny(List.of(task1, task2, task3));
}
```

This following example uses *submitTasks* to submit three value returning tasks. It uses the *CompletableFuture.completed* method to obtain a stream that is lazily populated as the tasks complete.

```
try (ExecutorService executor = Executors.newVirtualThreadExecutor()) {
    Callable<String> task1 = () -> "foo";
    Callable<String> task2 = () -> "bar";
    Callable<String> task3 = () -> "baz";
    List<CompletableFuture<String>> cfs = executor.submitTasks(List.of(task1, task2, task3));
    CompletableFuture.completed(cfs)
       .map(CompletableFuture::join)
       .forEach(System.out::println);
}
```

The following creates an ExecutorService that runs each task in its own virtual thread with a deadline. If the deadline expires before the executor has terminated then it will be shutdown and any tasks running will be cancelled (which causes the virtual thread to be interrupted).

```
Instant deadline = Instant.now().plusSeconds(30);
try (ExecutorService executor = Executors.newVirtualThreadExecutor().withDeadline(deadline)) {
    :
}
```