

ORACLE®

# Asynchronous API with CompletableFuture

## Performance Tips and Tricks

Sergey Kuksenko  
Java Platform Group, Oracle  
November, 2017

JavaYourNext

(Cloud)

## Safe Harbor Statement

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

# About me

- Java Performance Engineer at Oracle, @since 2010
- Java Performance Engineer, @since 2005
- Java Engineer, @since 1996
- OpenJDK/OracleJVM is the third JVM in my experience
- Co-author of JMH (Java Microbenchmark Harness)

# j.u.c.CompletableFuture

- @since Java8

# j.u.c.CompletableFuture

- @since Java8
- Not used in Java8

# j.u.c.CompletableFuture

- @since Java8
- Not used in Java8



**Sergey Kuksenko**

@kuksen0



All, could you complete small survey about CompletableFuture?

26% CompletableWhat?

37% I toyed with it

27% I am using it and happy

10% I am using it and unhappy

358 votes • Final results

# j.u.c.CompletableFuture

- @since Java8
- Not used in Java8



Sergey Kuksenko

@kuksenk0

All, could you complete small survey about CompletableFuture?

26% CompletableFutureWhat?

37% I toyed with it

27% I am using it and happy

10% I am using it and unhappy



CompletableFuture

Pull requests

Issues

Marketplace

Repositories 103

Code 62K

Commits 3K

Issues 1K

Wikis 227



# j.u.c.CompletableFuture

- Usage in Java9:
  - Process API
  - HttpClient\*

\*Most tips are from here!

# HttpClient

(a.k.a. JEP-110)

# HttpClient a.k.a. JEP-110

- Part of JDK 9, but not included into Java SE
  - module: `jdk.incubator.httpclient`
  - package: `jdk.incubator.http`

# HttpClient a.k.a. JEP-110

- Part of JDK 9, but not included into Java SE
  - module: `jdk.incubator.httpclient`
  - package: `jdk.incubator.http`
- Incubator Modules a.k.a. JEP-11
  - «*The incubation lifetime of an API is limited: It is expected that the API will either be standardized or otherwise made final in the next release, or else removed*»

# HttpClient

Two ways to send request:

- synchronous/blocking
- asynchronous

# synchronous/blocking

```
HttpClient client = «create client»;  
HttpRequest request = «create request»;  
  
HttpResponse<String> response =  
  
    client.send(request, BodyHandler.asString());  
  
if (response.statusCode() == 200) {  
    System.out.println("We've got: " + response.body());  
}  
  
...
```

# asynchronous

```
HttpClient client = «create client»;  
HttpRequest request = «create request»;  
  
CompletableFuture<HttpResponse<String>> futureResponse =  
  
    client.sendAsync(request, BodyHandler.asString());  
  
futureResponse.thenAccept( response -> {  
    if (response.statusCode() == 200) {  
        System.out.println("We've got: " + response.body());  
    }  
});  
...  

```

# Client builder

*Good habit for async API*



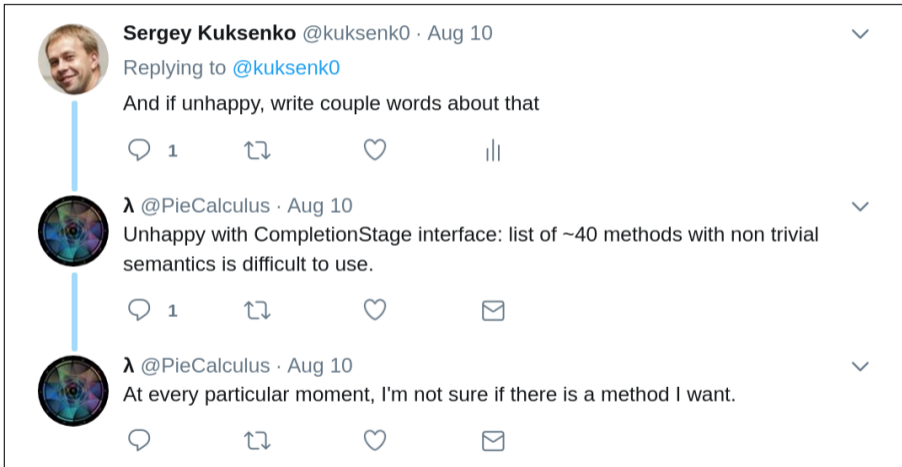
```
HttpClient client = HttpClient.newBuilder()
    .authenticator(someAuthenticator)
    .sslContext(someSSLContext)
    .sslParameters(someSSLParameters)
    .proxy(someProxySelector)
    .executor(someExecutorService)
    .followRedirects(HttpClient.Redirect.ALWAYS)
    .cookieManager(someCookieManager)
    .version(HttpClient.Version.HTTP_2)
    .build();
```



# First step of performance:

First step of performance:  
developers!

# What about Java developers performance?



The image shows a screenshot of a Twitter thread. It features three tweets. The first tweet is from Sergey Kuksenko (@kuksen0) replying to himself, asking for feedback on a performance issue. The second and third tweets are from λ (@PieCalculus), who expresses dissatisfaction with the CompletionStage interface and mentions a list of ~40 methods with non-trivial semantics. The tweets include icons for replies, retweets, likes, and a share icon.

**Sergey Kuksenko** @kuksen0 · Aug 10  
Replying to @kuksen0  
And if unhappy, write couple words about that

1   ↻   ♥   |||

**λ** @PieCalculus · Aug 10  
Unhappy with CompletionStage interface: list of ~40 methods with non trivial semantics is difficult to use.

1   ↻   ♥   ✉

**λ** @PieCalculus · Aug 10  
At every particular moment, I'm not sure if there is a method I want.

↻   ♥   ✉

# j.u.c.CompletionStage

- contains 38 methods
- 36 of them has 3 forms:
  - `somethingAsync(..., executor)`
  - `somethingAsync(...)`
  - `something(...)`

# j.u.c.CompletionStage

- `somethingAsync(..., executor)`
  - runs action chain in the executor
- `somethingAsync(...)`
  - `somethingAsync(..., ForkJoinPool.commonPool())`
- `something(...)`
  - default execution \*

# j.u.c.CompletionStage

- 12 methods remained
- 9 of them has 3 forms:
  - **Apply** – function from input to R, result `CompletableFuture<R>`
  - **Accept** – consumer of input, result `CompletableFuture<Void>`
  - **Run** – just execute `Runnable`, result `CompletableFuture<Void>`

# j.u.c.CompletionStage

- single input
  - thenApply, thenAccept, thenRun
- binary «or»
  - applyToEither, acceptEither, runAfterEither
- binary «and»
  - thenCombine, thenAcceptBoth, runAfterBoth

# j.u.c.CompletionStage

- 3 methods remained
  - thenCompose
  - handle
  - whenComplete



# j.u.c.CompletionStage

- thenCompose
  - function from input to CompletableFuture<R>, result CompletableFuture<R>
  - a.k.a flatMap

# j.u.c.CompletionStage

- handle
  - function from input and exception to R, result `CompletableFuture<R>`

# j.u.c.CompletionStage

- `whenComplete`
  - consumer from input and exception
  - similar to `Accept` methods above
  - result - the same as input

# j.u.c.CompletionStage

- 2 methods remained (doesn't have async versions)
  - `exceptionally` - function from exception to R, result `CompletableFuture<R>`
  - `toCompletableFuture`

# j.u.c.CompletableFuture

- contains 38 methods inherited from `CompletionStage`  
**and**
- 22 other instance methods
- 12 static methods

# j.u.c.CompletableFuture

- 9 ways to complete future
  - `complete/completeAsync/completeExceptionally`
  - `cancel`
  - `obtrudeValue/obtrudeException`
  - `completeOnTimeout/orTimeout`

# j.u.c.CompletableFuture

- 4 ways to get value
  - `get/join` – blocking
  - `get(timeout, TimeUnit)` – not so blocking
  - `getNow(valueIfAbsent)` – non-blocking

# j.u.c.CompletableFuture

- 3 ways to know status
  - `isDone`
  - `isCompletedExceptionally`
  - `isCancelled`



# j.u.c.CompletableFuture

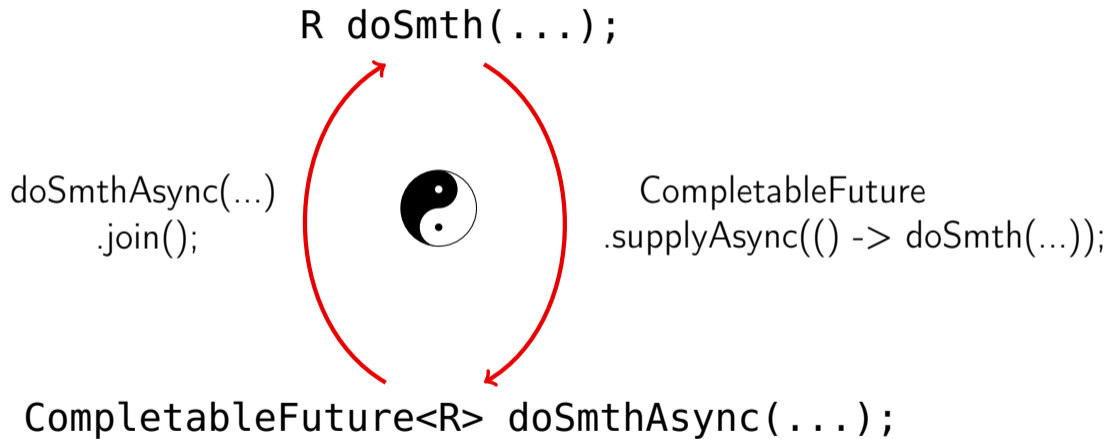
- 8 static methods to create future
  - `completedFuture/completedStage`
  - `failedFuture/failedStage`
  - `runAsync(Runnable) → CompletableFuture<Void>`
  - `supplyAsync(Supplier<U>) → CompletableFuture<U>`

# Blocking or asynchronous?

# Blocking or asynchronous

- Blocking:
  - `R doSmth(...);`
- Asynchronous:
  - `CompletableFuture<R> doSmthAsync(...);`

# ~~Yin and yang~~ Blocking and asynchronous



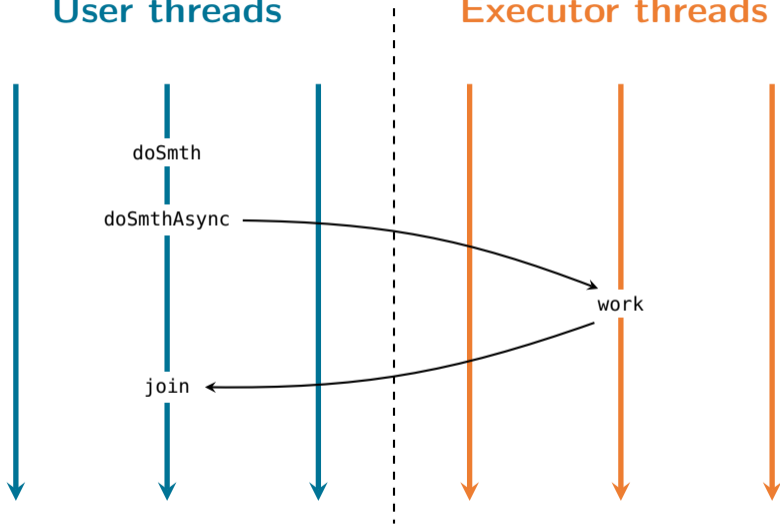
# Blocking via async

```
R doSmth(...) {  
    return doSmthAsync(...).join();  
}
```

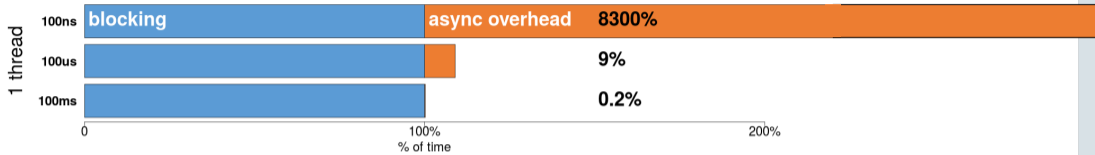
Will it work?

## User threads

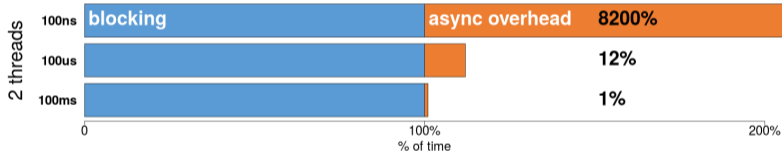
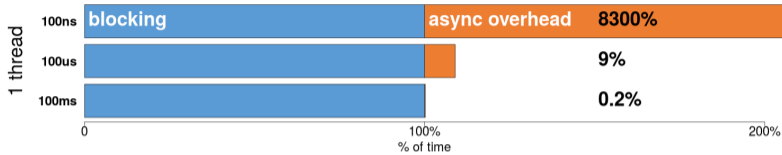
## Executor threads



# Let's measure

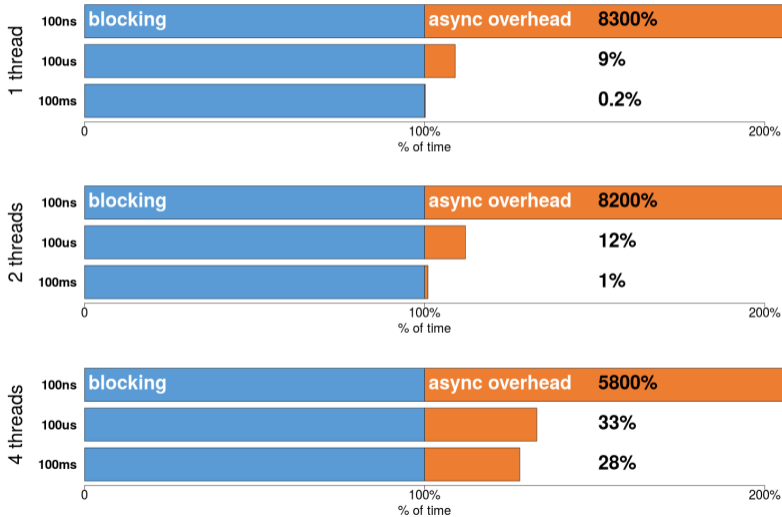


# Let's measure





# Let's measure



**Avoid transition task from one thread to another.  
It costs.**

# Async via blocking

```
CompletableFuture<R> doSmthAsync(...) {  
    return CompletableFuture.supplyAsync(()->doSmth(...), executor);  
}
```

Will it work?

# Back to HttpClient

```
public <T> HttpResponse<T>
send(HttpRequest req, HttpResponse.BodyHandler<T> responseHandler) {
    ...
}

public <T> CompletableFuture<HttpResponse<T>>
sendAsync(HttpRequest req, HttpResponse.BodyHandler<T> responseHandler) {
    return CompletableFuture.supplyAsync(() -> send(req, responseHandler), executor);
}
```

Will it work?

# Back to HttpClient

```
public <T> HttpResponse<T>  
send(HttpRequest req, HttpResponse.BodyHandler<T> responseHandler) {  
    ...  
}
```

```
public <T> CompletableFuture<HttpResponse<T>>  
sendAsync(HttpRequest req, HttpResponse.BodyHandler<T> responseHandler) {  
    return CompletableFuture.supplyAsync(() -> send(req, responseHandler), executor);  
}
```

Sometimes.

# One does not simply make «sendAsync»

- send header
- send body
- receive header from server
- receive body from server

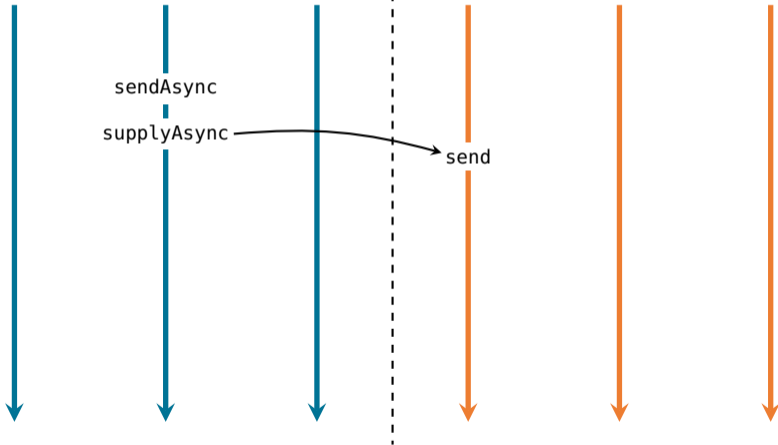
# One does not simply make «sendAsync»

- send header
- send body
- wait header from server
- wait body from server



## User threads

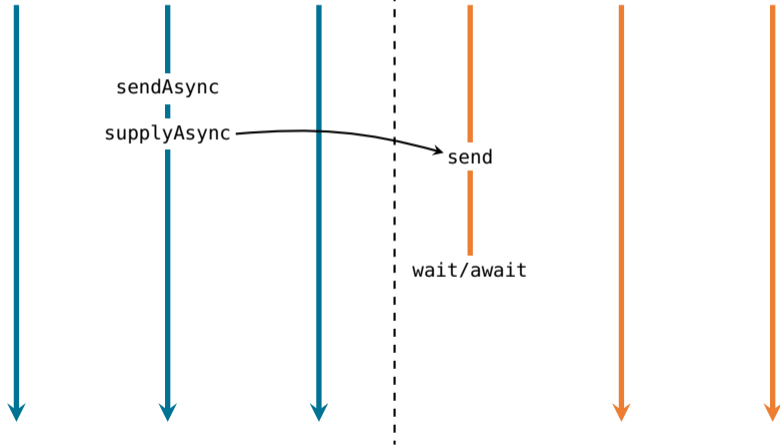
## Executor threads





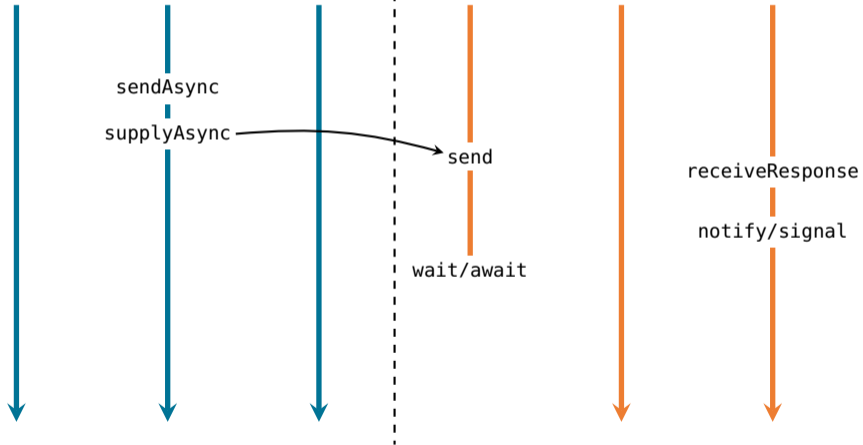
## User threads

## Executor threads



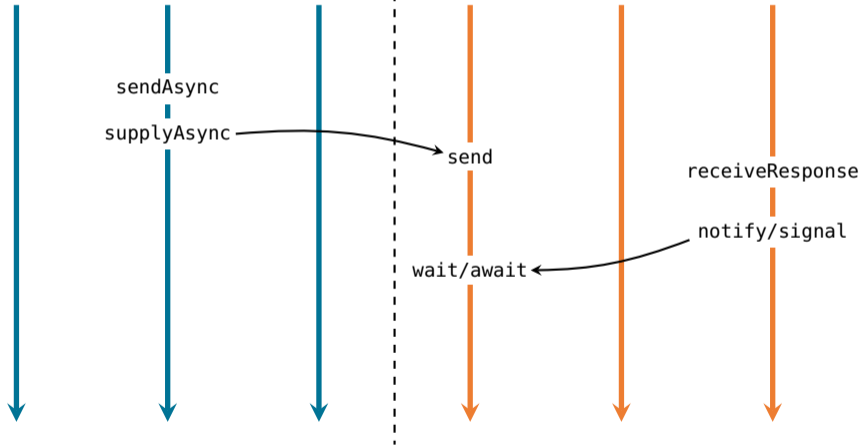
## User threads

## Executor threads



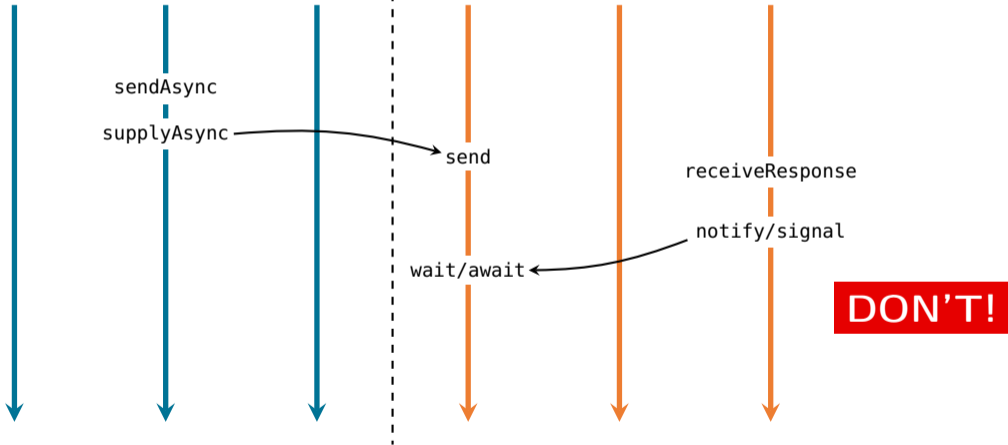
## User threads

## Executor threads



## User threads

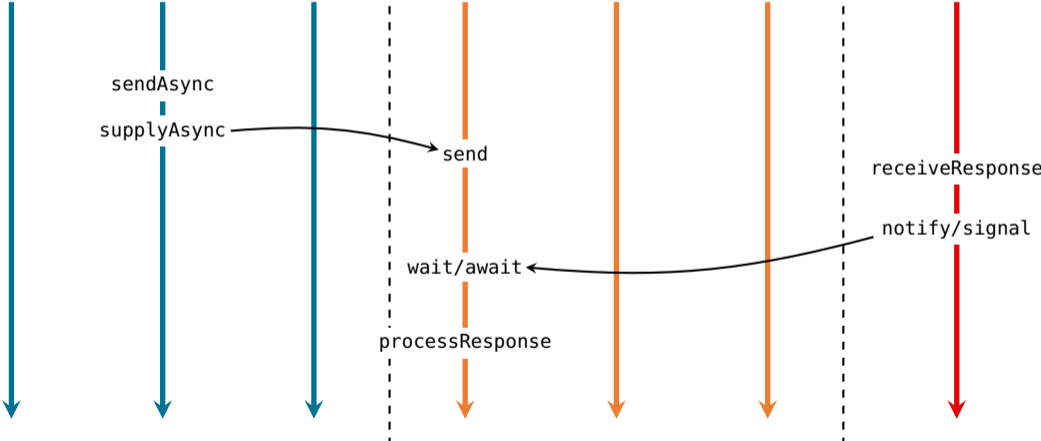
## Executor threads



# User threads

# Executor threads

# Aux threads



## RTFM (HttpClient.Builder)

```
/**
 * Sets the executor to be used for asynchronous tasks. If this method is
 * not called, a default executor is set, which is the one returned from
 * {@link java.util.concurrent.Executors#newCachedThreadPool()
 * Executors.newCachedThreadPool}.
 *
 * @param executor the Executor
 * @return this builder
 */
public abstract Builder executor(Executor executor);
```

If not explicitly stated otherwise, async API should be able to work with any kind of executor.

# RTFM (java.util.concurrent.Executors)

```
/**
 * Creates a thread pool that creates new threads as needed, but
 * will reuse previously constructed threads when they are
 * available. These pools will typically improve the performance
 * of programs that execute many short-lived asynchronous tasks.
 * Calls to {@code execute} will reuse previously constructed
 * threads if available. If no existing thread is available, a new
 * thread will be created and added to the pool. Threads that have
 * not been used for sixty seconds are terminated and removed from
 * the cache. Thus, a pool that remains idle for long enough will
 * not consume any resources. Note that pools with similar
 * properties but different details (for example, timeout parameters)
 * may be created using {@link ThreadPoolExecutor} constructors.
 *
 * @return the newly created thread pool
 */
public static ExecutorService newCachedThreadPool()
```

# CachedThreadPool

- Pro:
  - If all threads are busy, the task will be executed in a new thread
- Con:
  - If all threads are busy, a new thread is created



## sendAsync via send

The single HttpRequest uses ~ 20 threads.

Does it mean that

100 simultaneous requests  $\Rightarrow$  ~ 2000 threads?

# sendAsync via send

The single HttpRequest uses ~ 20 threads.

Does it mean that

100 simultaneous requests  $\Rightarrow$  ~ ~~2000~~ threads?

100 simultaneous requests  $\Rightarrow$  OutOfMemoryError!

# Eliminate waiting (step 1)

## Executor thread

Condition responseReceived;

```
R send(...) {  
    sendRequest(...);  
    responseReceived.await();  
    processResponse();  
    ...  
}
```

## Aux thread

```
... receiveResponse(...) {  
    ...  
    responseReceived.signal();  
    ...  
}
```



# Eliminate waiting (step 1)

CompletableFuture as a single-use Condition.

## Executor thread

## Aux thread

```
CompletableFuture<...> responseReceived;
```

```
R send(...) {  
    sendRequest(...);  
    responseReceived.join();  
    processResponse();  
    ...  
}
```

```
... receiveResponse(...) {  
    ...  
    responseReceived.complete();  
    ...  
}
```



# Eliminate waiting (step 2)

```
CompletableFuture<...> sendAsync(...) {  
    return CompletableFuture.supplyAsync(() -> send(...));  
}
```

```
R send(...) {  
    sendRequest(...);  
    responseReceived.join();  
    return processResponse();  
}
```

# Eliminate waiting (step 2)

```
CompletableFuture<...> sendAsync(...) {  
    return CompletableFuture.supplyAsync(() -> sendRequest(...))  
        .thenApply(...) -> responseReceived.join()  
        .thenApply(...) -> processResponse();  
}
```

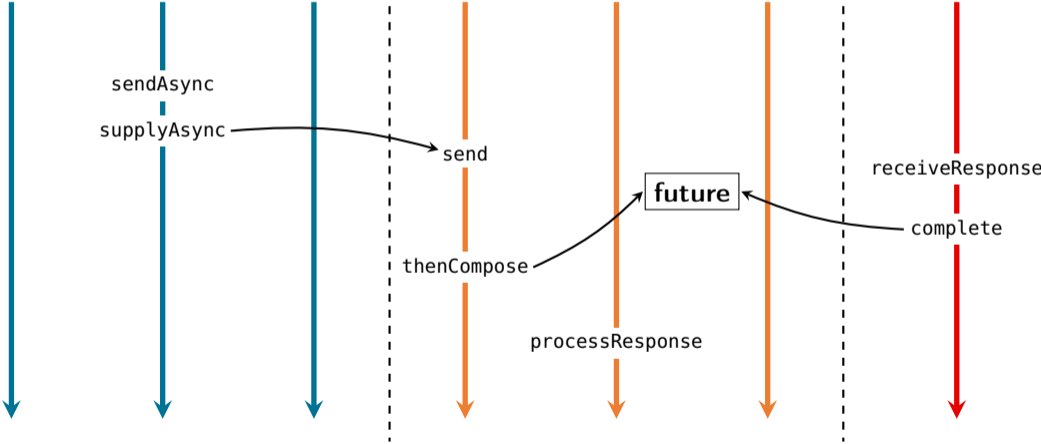
# Eliminate waiting (step 2)

```
CompletableFuture<...> sendAsync(...) {  
    return CompletableFuture.supplyAsync(() -> sendRequest(...))  
        .thenCompose((...) -> responseReceived)  
        .thenApply((...) -> processResponse());  
}
```

# User threads

# Executor threads

# Aux threads





# What about sendAsync performance?

`wait()/await()` elimination



**+40%**

**Avoid blocking inside `CompletableFuture` chains.  
It costs.**

# Quiz

Thread 1

```
future.thenApply((...) -> foo());
```

Thread 2

```
future.complete(...);
```

Which thread will execute `foo()`?

- A) thread 1
- B) thread 2
- C) thread 1 or thread 2
- D) thread 1 and thread 2

# Quiz

Thread 1

```
future.thenApply((...) -> foo());
```

Thread 2

```
future.complete(...);
```

Which thread will execute `foo()`?

- A) thread 1
- B) thread 2
- C) thread 1 or thread 2
- D) thread 1 and thread 2

← **correct answer**

Where exactly CompletableFuture chain of actions will be executed?

## Two simple rules

- Completion thread executes actions attached «long enough» before completion.
- Construction thread executes actions if CompletableFuture is already completed («long enough» before).

## Two simple rules (not always work)

- Completion thread executes actions attached «long enough» before completion.
- Construction thread executes actions if CompletableFuture is already completed («long enough» before).

**Races are coming!**

Chain of actions may be executed from:

- Completion thread
  - `complete`, `completeExceptionally` ...
- Construction thread
  - `thenApply`, `thenCompose` ...
- Value getting thread
  - `get`, `join` ...



Let's check!

# jsctress

<http://openjdk.java.net/projects/code-tools/jcstress/>

The Java Concurrency Stress tests (jcstress) is an experimental harness and a suite of tests to aid the research in the correctness of concurrency support in the JVM, class libraries, and hardware.

# Example 1

```
CompletableFuture<...> start =  
    new CompletableFuture<>();
```

---

```
start.complete(...); | start.thenApply(a -> action());
```

Results:

Occurrences	Expectation	Interpretation
1,630,058,138	ACCEPTABLE	action in chain construction thread
197,470,850	ACCEPTABLE	action in completion thread

## Example 2

```
CompletableFuture<...> start =  
    new CompletableFuture<>();  
start.thenApply(a -> action());  
-----  
start.complete(...); | start.complete(...);
```

### Results:

Occurrences	Expectation	Interpretation
819,755,198	ACCEPTABLE	action in successful completion thread
163,205,510	ACCEPTABLE	action in failed completion thread

## Example 3

```
CompletableFuture<...> start =  
    new CompletableFuture<>();  
start.thenApply(a -> action());  
-----  
start.complete(...); | start.join();
```

Results:

Occurrences	Expectation	Interpretation
904,651,258	ACCEPTABLE	action in completion thread
300,524,840	ACCEPTABLE	action in join thread

## Example 4

```
CompletableFuture<...> start =  
    new CompletableFuture<>();  
start.thenApply(a -> action1());  
start.thenApply(a -> action2());  
-----  
start.complete(...); | start.join();
```

### Results:

Occurrences	Expectation	Interpretation
179,525,918	ACCEPTABLE	both actions in the same thread
276,608,380	ACCEPTABLE	actions in different threads

# What is faster?

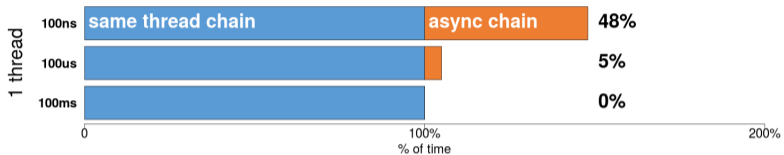
## Same thread chain

```
future
  .thenApply(...) -> foo1()
  .thenApply(...) -> foo2()
```

## Async chain

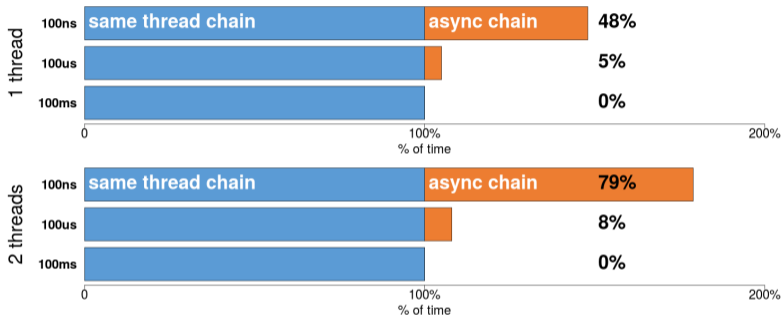
```
future
  .thenApplyAsync(...) -> foo1(), executor)
  .thenApplyAsync(...) -> foo2(), executor);
```

# Let's measure

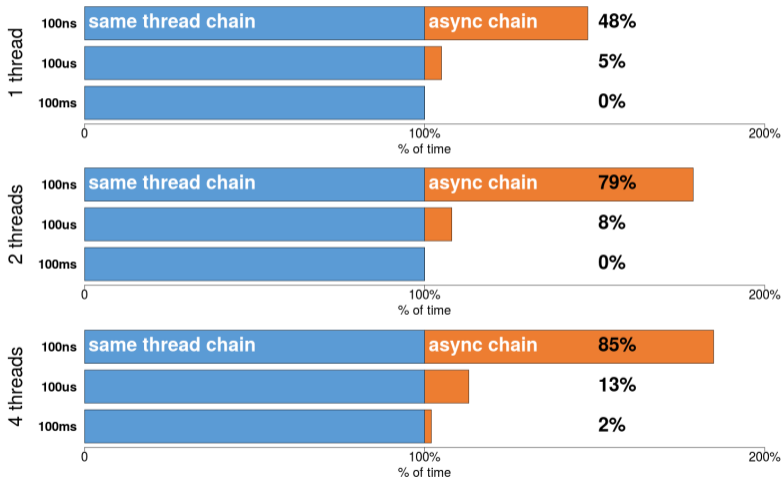




# Let's measure



# Let's measure



## CompletableFuture chaining

- `thenSomethingAsync(...)` – gives predictability.
- `thenSomething(...)` – gives performance.

**Avoid transition task from one thread to another.  
It costs.**

# When predictability is important

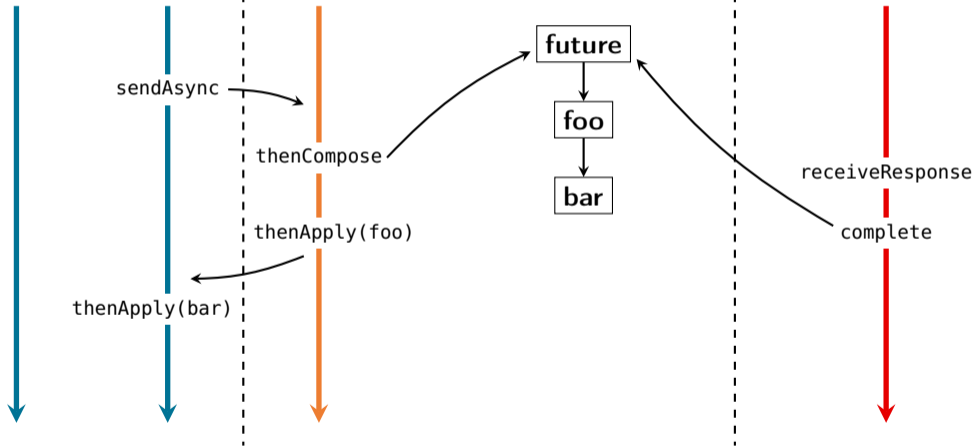
`HttpClient` has the single auxiliary thread «`SelectorManager`».

- waits on `Selector.select`
- reads data from `Socket`
- extracts HTTP2 frames
- distributes frames to receivers

## User threads

## Executor threads

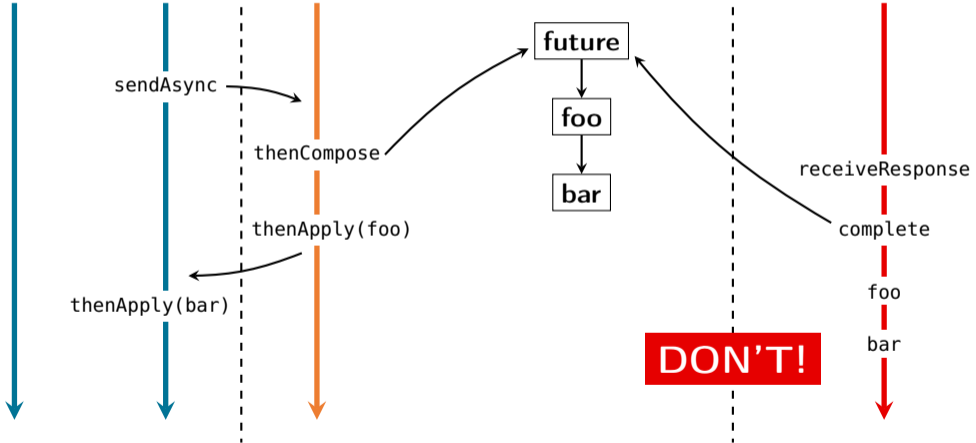
## SelectorManager



## User threads

## Executor threads

## SelectorManager



# When predictability is important

```
CompletableFuture<...> response;
```

Executor thread

```
...  
.thenCompose(() -> response)  
...
```

«SelectorManager»

```
response.complete(...);
```



# One way (@since 9)

```
CompletableFuture<...> response;
```

Executor thread

```
...  
.thenCompose(() -> response)  
...
```

«SelectorManager»

```
response.completeAsync(..., executor);
```

# Another way

```
CompletableFuture<...> response;
```

Executor thread

«SelectorManager»

```
...  
.thenComposeAsync(() -> response, executor)    response.complete(...);  
...
```

# What we've got (in both cases)

- Pro:
  - «SelectorManager» is protected
- Con:
  - Switching from one executor thread to another executor thread (costs).

# Third way

```
CompletableFuture<...> response;
```

## Executor thread

```
CompletableFuture<...> cf = response;  
if(!cf.isDone()) {  
    cf = cf.thenApplyAsync(x -> x, executor);  
}  
...thenCompose(() -> cf);  
...
```

## «SelectorManager»

```
response.complete(...);
```

# What about sendAsync performance?

Tuning complete()



**+16%**

**Carefully avoid transition task  
from one thread to another.  
It costs.**

# What if server responds quickly?

```
CompletableFuture<...> sendAsync(...) {  
    return  
    sendHeaderAsync(..., executor)  
    .thenCompose(() -> sendBody())  
    .thenCompose(() -> getResponseHeader())  
    .thenCompose(() -> getResponseBody())  
    ...  
}
```

Sometimes (3% cases)

CompletableFuture  
is already completed

getResponseBody() is executed from user thread

## We have `thenComposeAsync()`

- Pro:
  - User thread is protected
- Con:
  - Switching from one executor thread to another executor thread (costs).



# Do it

```
CompletableFuture<...> sendAsync(...) {
    CompletableFuture<Void> start = new CompletableFuture<>();

    CompletableFuture<...> end = start.thenCompose(v -> sendHeader())
        .thenCompose(() -> sendBody())
        .thenCompose(() -> getResponseHeader())
        .thenCompose(() -> getResponseBody())
        ...;

    start.completeAsync(() -> null, executor); // trigger execution
    return end;
}
```

# What about sendAsync performance?

Delayed start



**+10%**

**It may be useful to build chain of actions before execution.**

# Let's back to `CachedThreadPool`

- Pro:
  - If all threads are busy, the task will be executed in a new thread
- Con:
  - If all threads are busy, a new thread is created

Is that a good choice for the default executor?

# Try different executors

CachedThreadPool      35500 ops/sec

FixedThreadPool(2)      61300 ops/sec

**+72%**

**Different ThreadPools show different performance.**

Q & A ?

# Appendix



# another example of thenCompose

```
// e.g. how to make recursive CompletableFuture chain
```

```
CompletableFuture<...> makeRecursiveChain(...) {  
    if(«recursion ends normally») {  
        return CompletableFuture.completedFuture(...);  
    } else if(«recursion ends abruptly») {  
        return CompletableFuture.failedFuture(...); // appeared in Java9  
    }  
    return CompletableFuture.supplyAsync(() -> doSomething(...))  
        .thenCompose((...) -> makeRecursiveChain(...));  
}
```

ORACLE®