Asynchronous Java: Completable Future

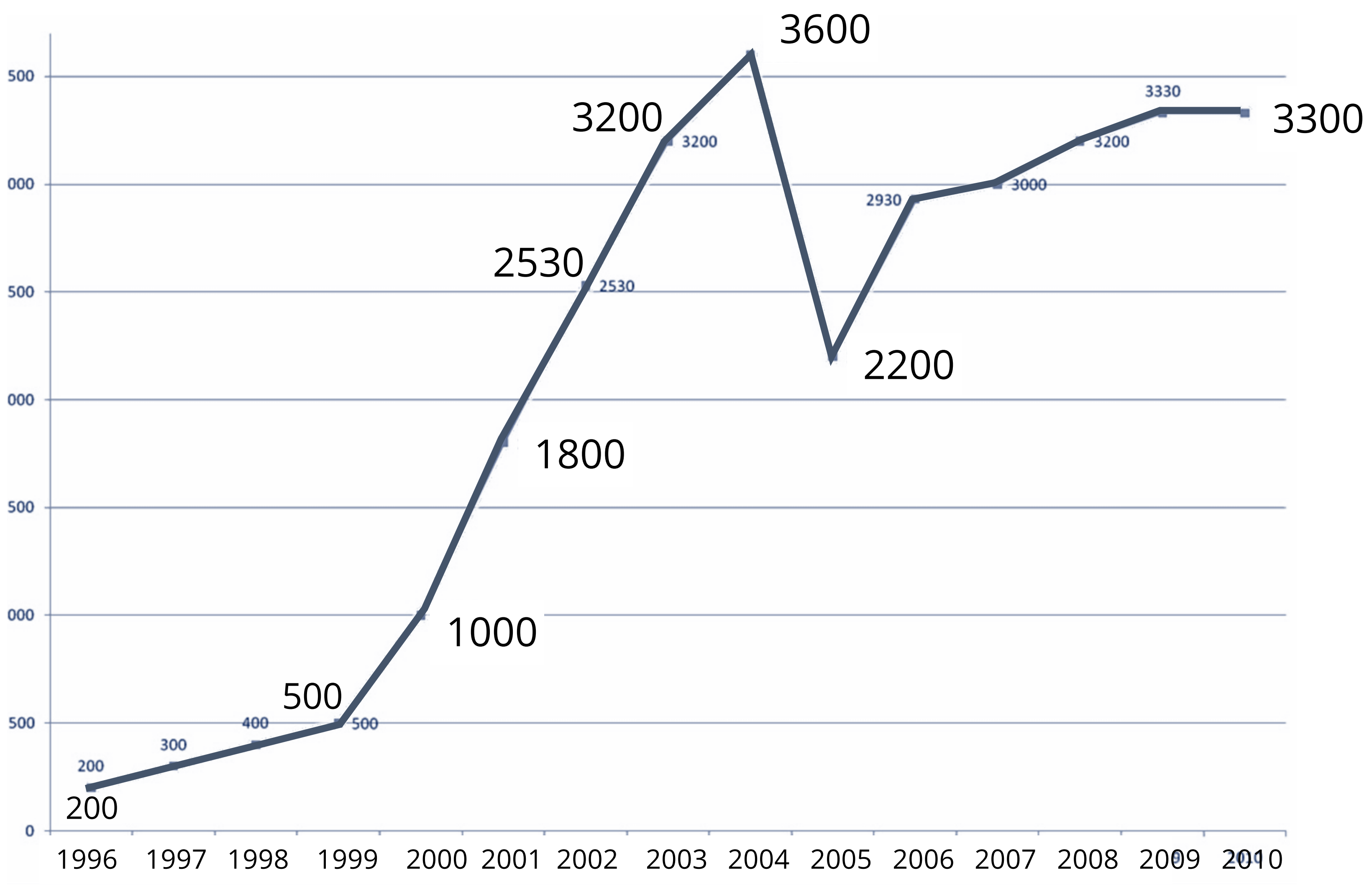
Advanced Java I. Functional, Asynchronous, Reactive Java

Module 4

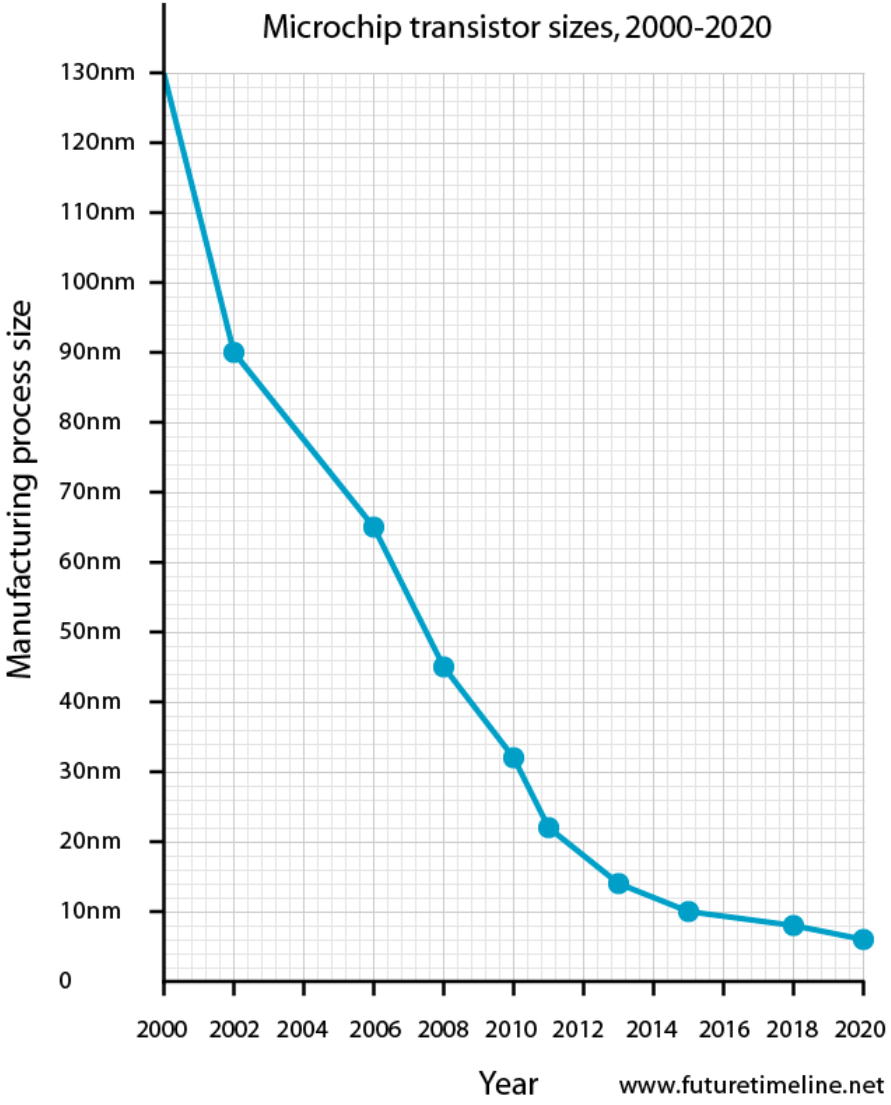
think.
create.
accelerate.

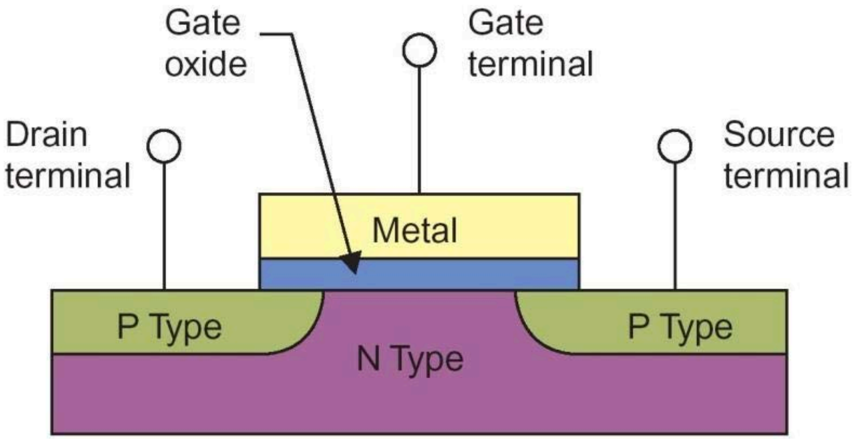
luxoft | training
A DXC Technology Company

CPU frequency is not growing anymore



Why CPU frequency doesn’t grow?





Multithreading in reality

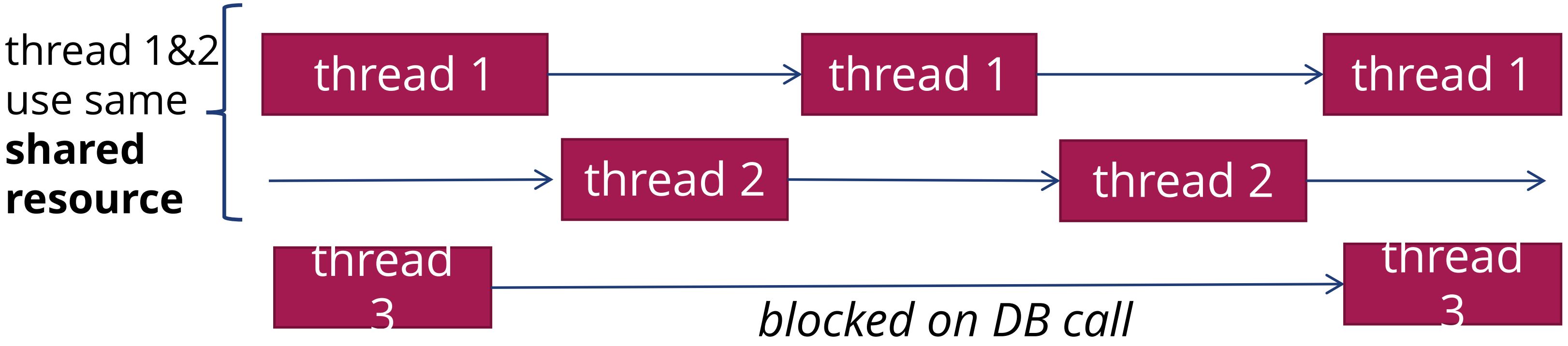
Parallel and asynchronous programming

parallel

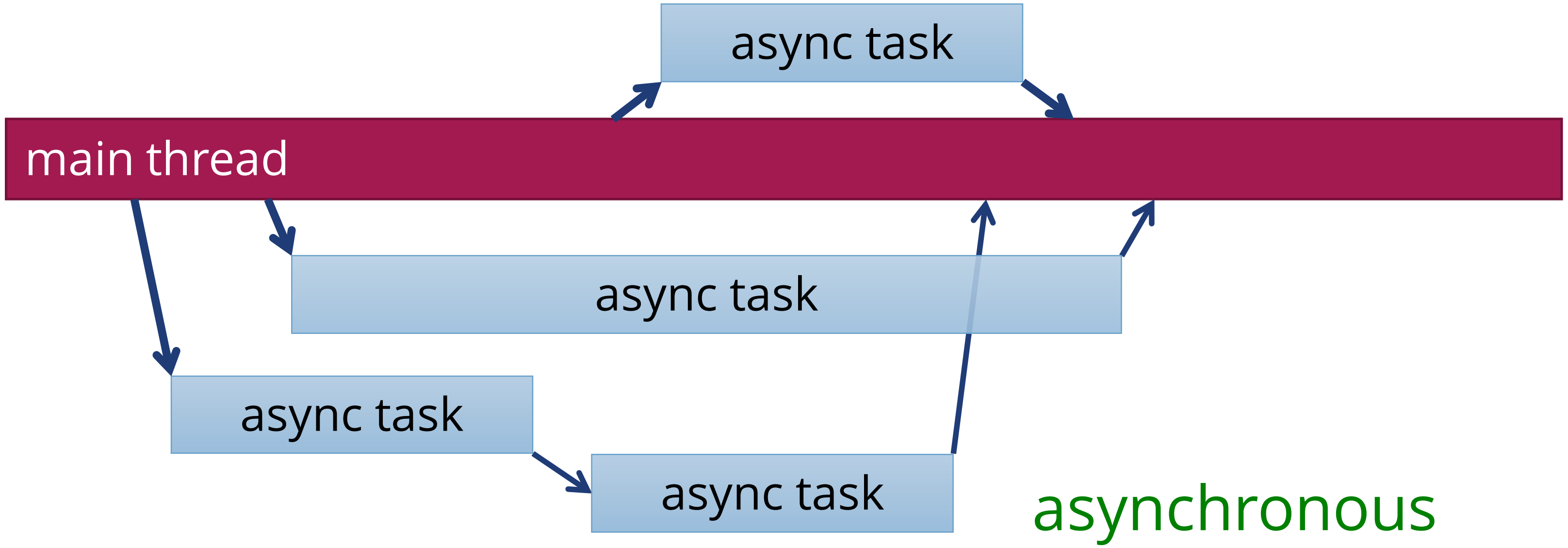
asynchronous

- no issues with atomicity or visibility
- no context switch

Parallel and asynchronous programming

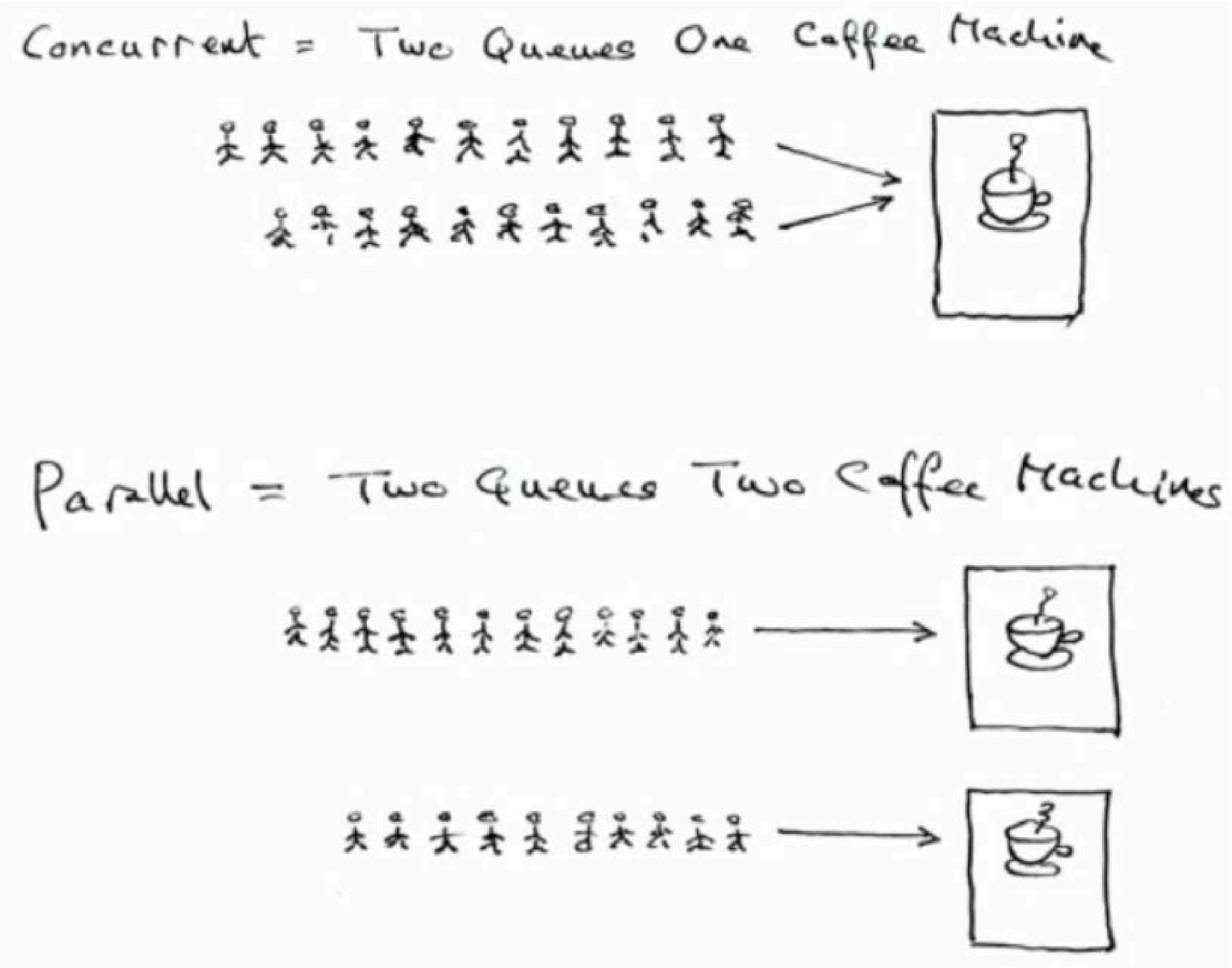


parallel



asynchronous

Concurrency and parallelism

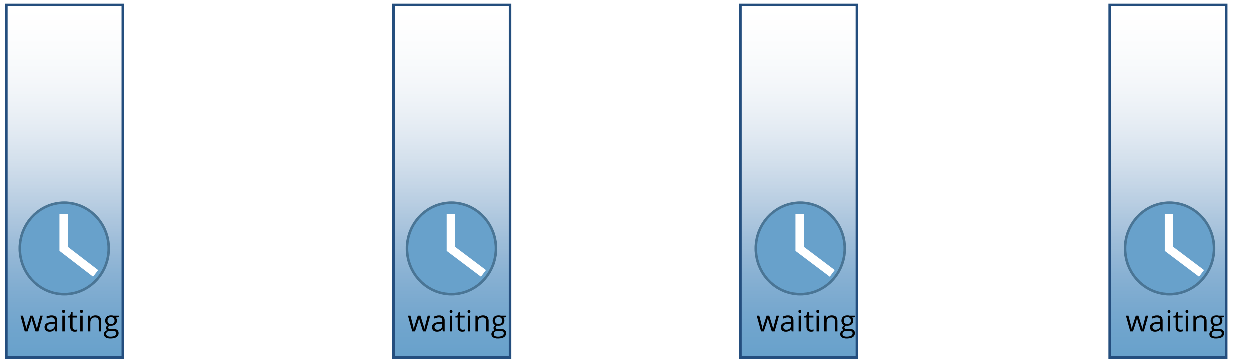

Concurrent = Two Queues One Coffee Machine
Parallel = Two Queues Two Coffee Machines

Question: how this scenario will work with asynchronousity?

Synchronous I/O: Threads are get blocked!

readFile()   readDB() …

It blocks the thread!



blocked on
reading file

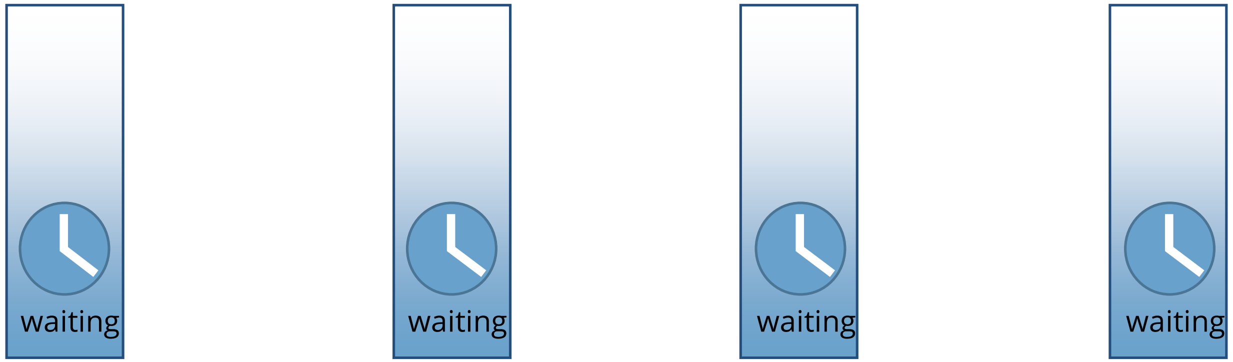
blocked on
reading
from DB

blocked on
heavy
calculations

blocked on
responding
the client

Synchronous I/O: We get the queue of clients

the queue of the clients



blocked on
reading file

blocked on
reading
from DB

blocked on
heavy
calculations

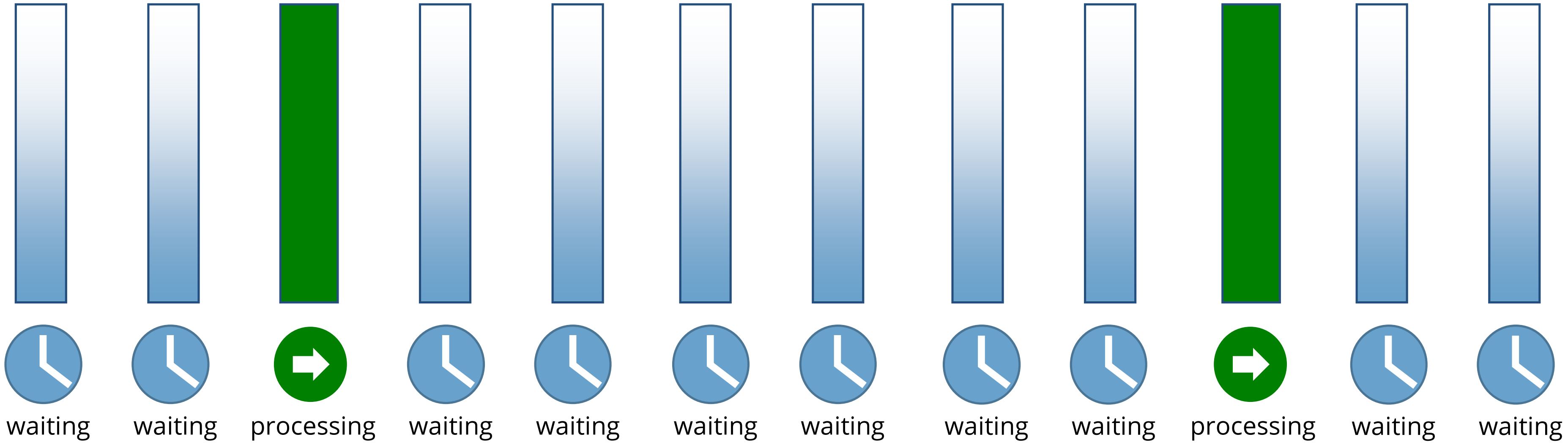
blocked on
responding
the client

What is a problem in synchronous code?

readFile()   readDB() ...

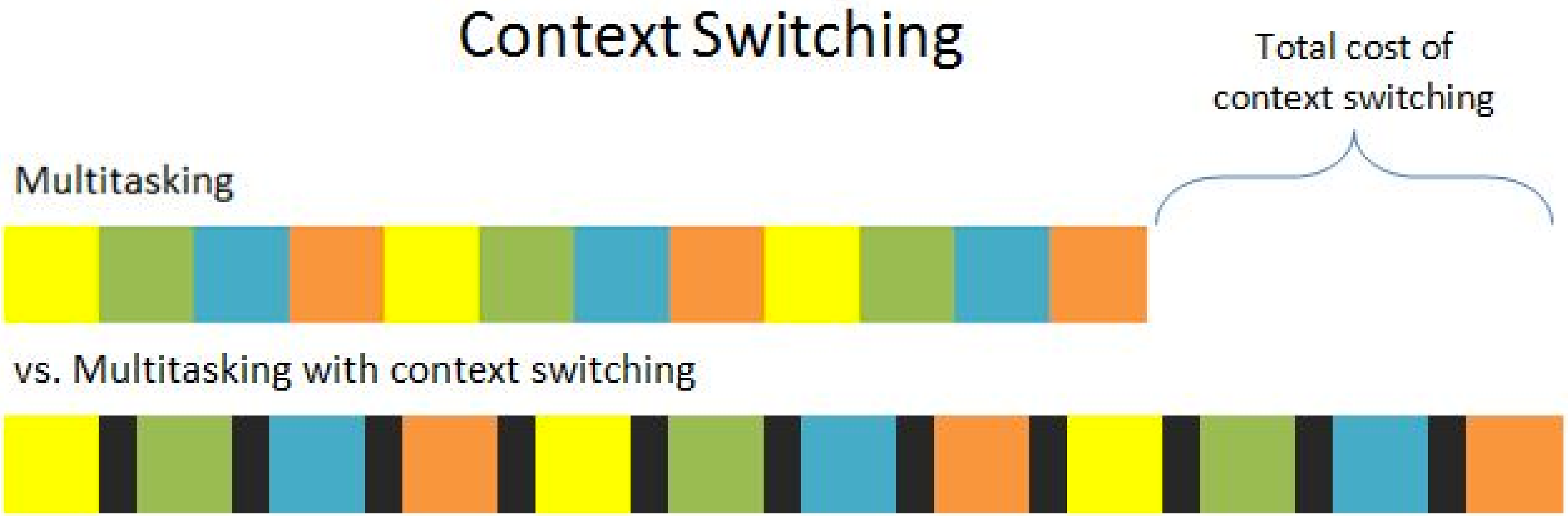
Let’s create more threads...

We’ll have the same situation, just a little bit later...



What is a problem in synchronous code?

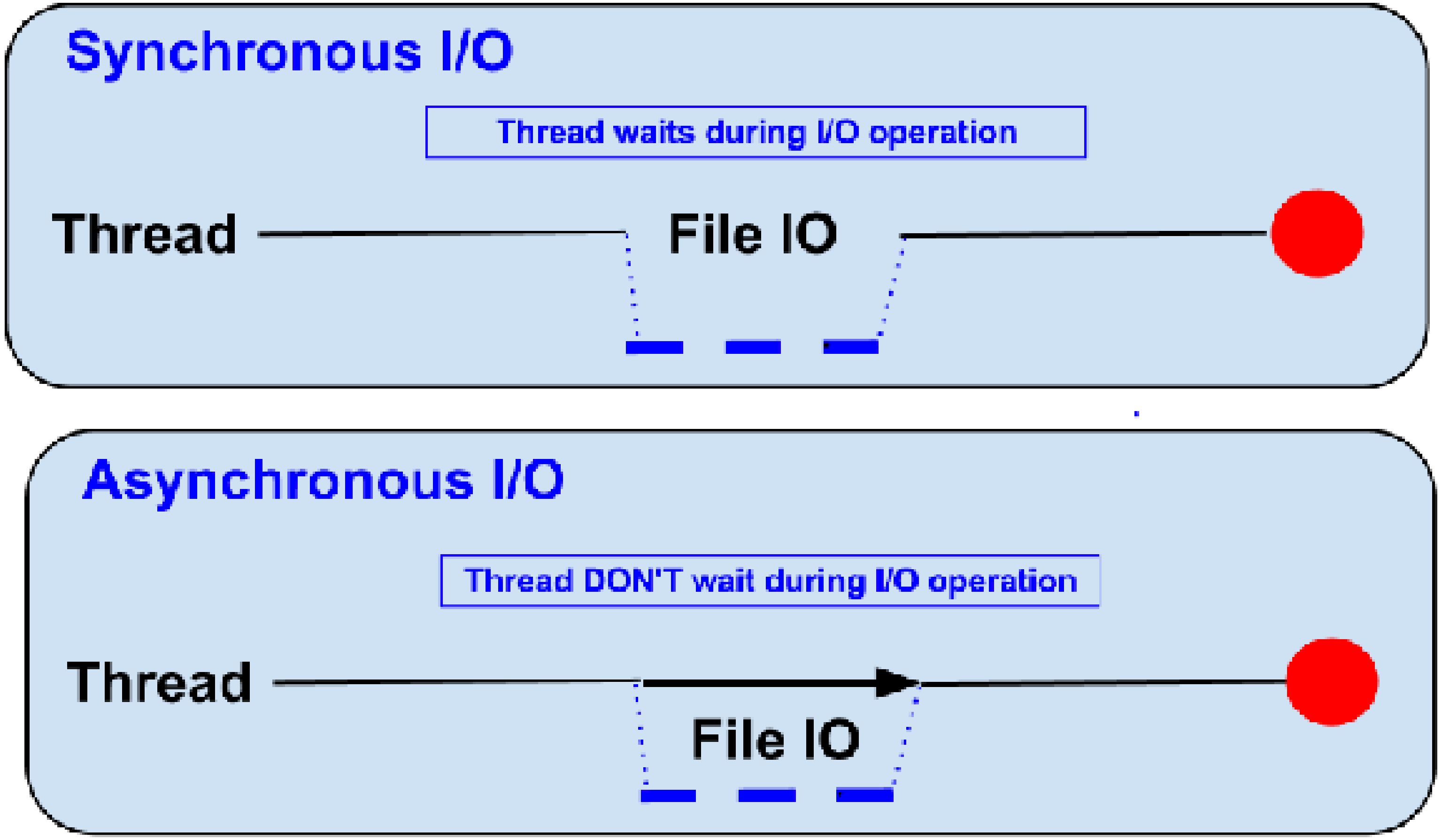
…and we will have context switch overhead


Context Switching
Total cost of context switching
Multitasking
vs. Multitasking with context switching

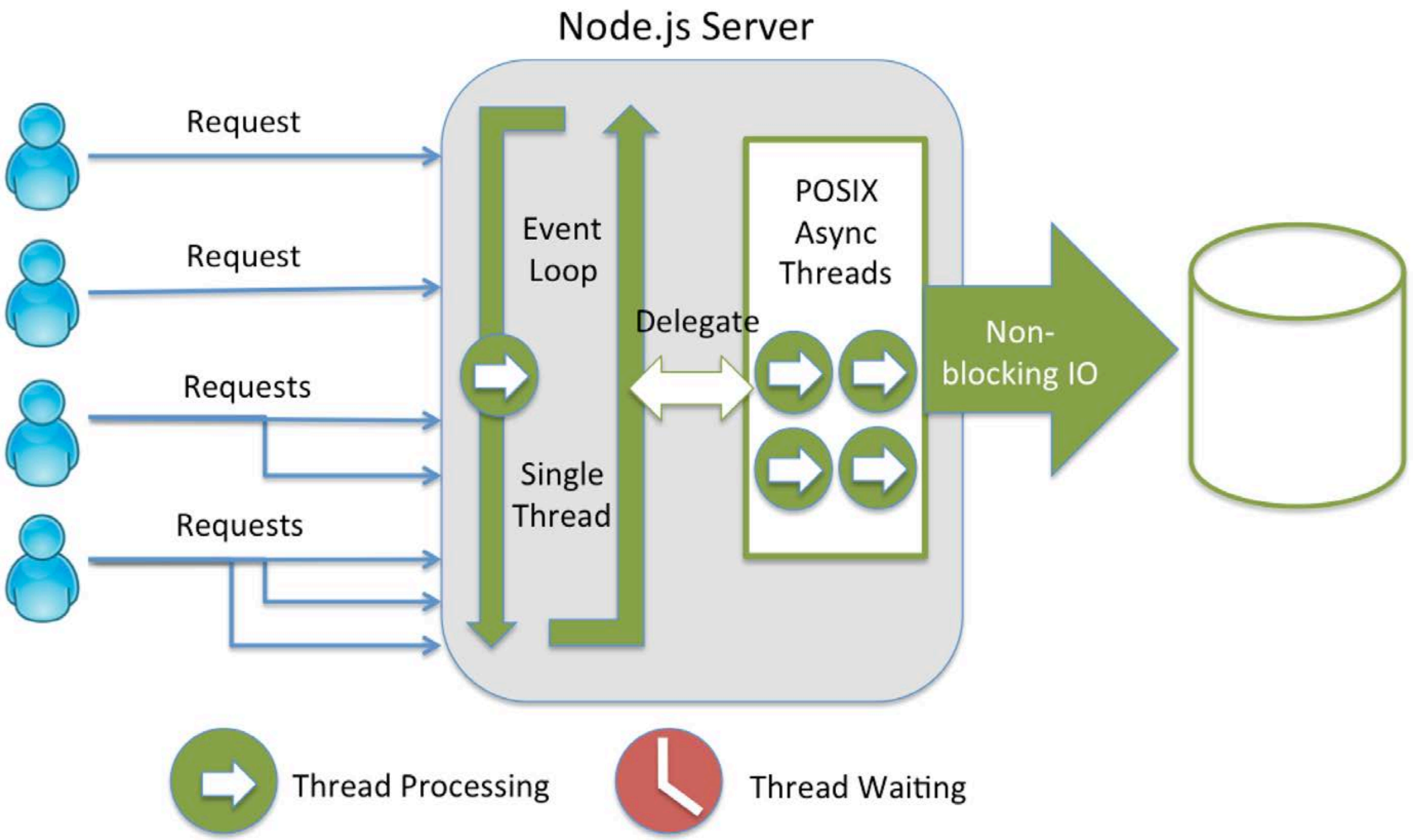
For sure, we have instruments to deal with these issues:

- BlockingQueue
- ThreadPool

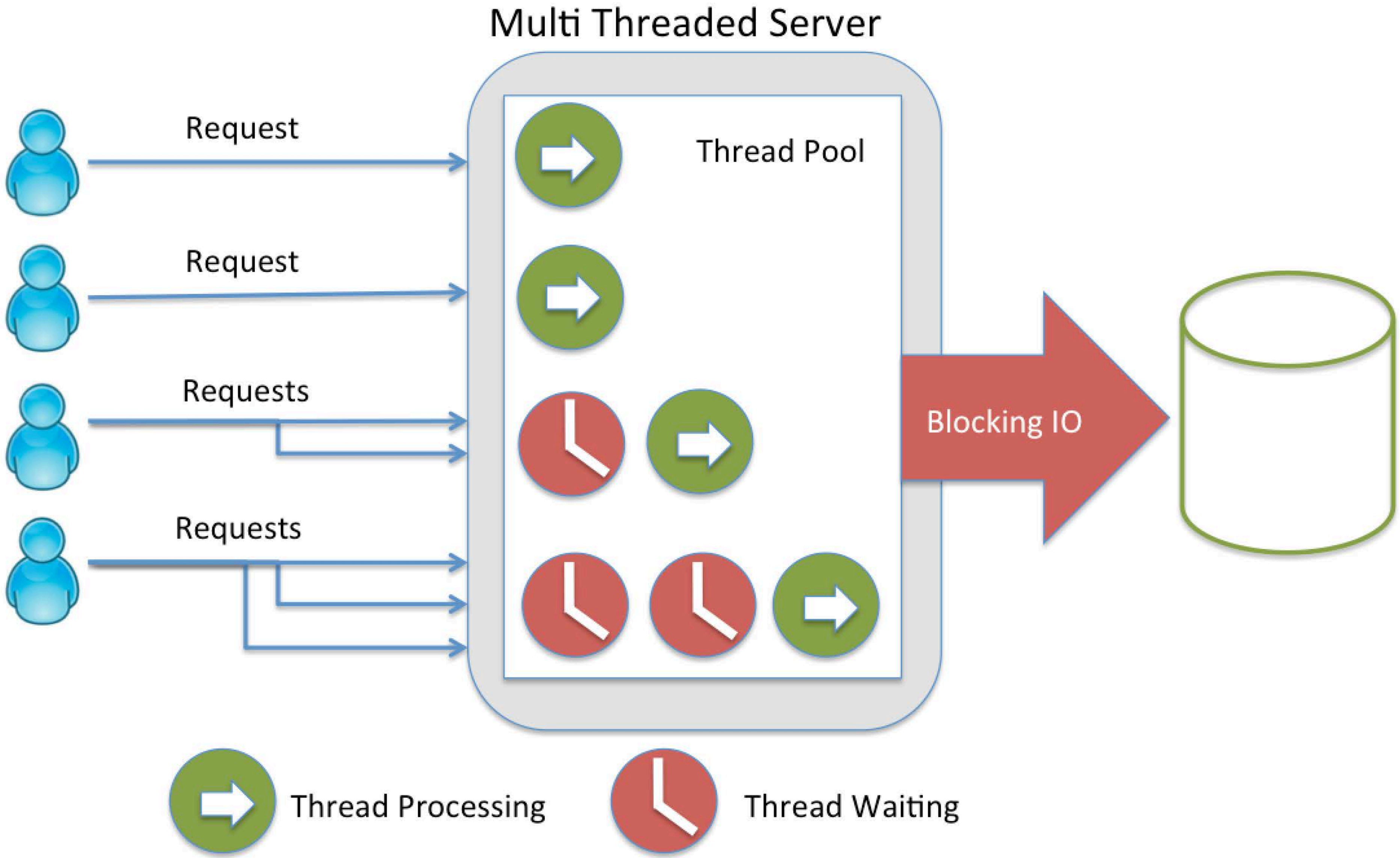
Solution: asynchronous I/O


Synchronous I/O
Thread waits during I/O operation
Thread
File IO
Asynchronous I/O
Thread DON'T wait during I/O operation
Thread
File IO

Node.js – pioneer in asynchronous execution



What happens in Java?



History of multithreading: plain old Java

Old good Java (before 1.5):

- Threads
- synchronization
- wait/notify

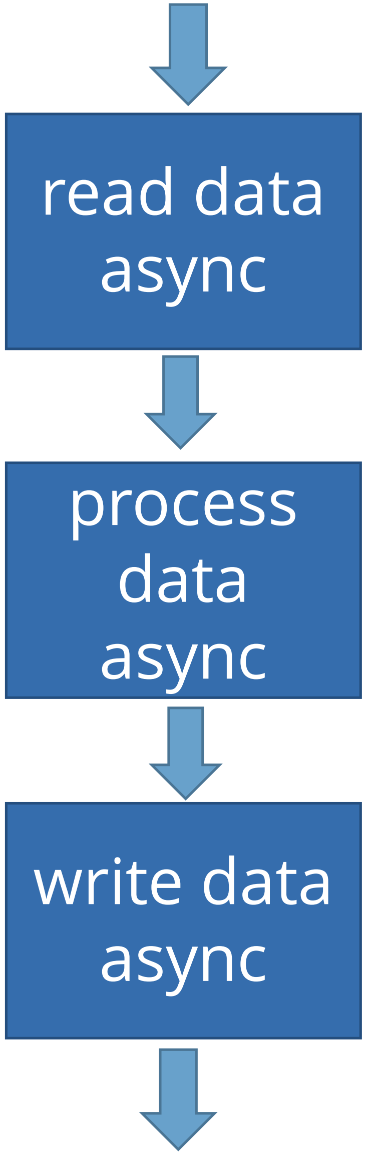
Difficult to write, debug and test

History of multithreading: Java 5

- Future interface
  - V get()
  - boolean cancel()
  - boolean isCancelled()
  - boolean isDone()
- Executors
- Callable interface
- BlockingQueue

It’s easy to execute in parallel, but how to define data flow?

Data flow



readData.get()

processData.get()

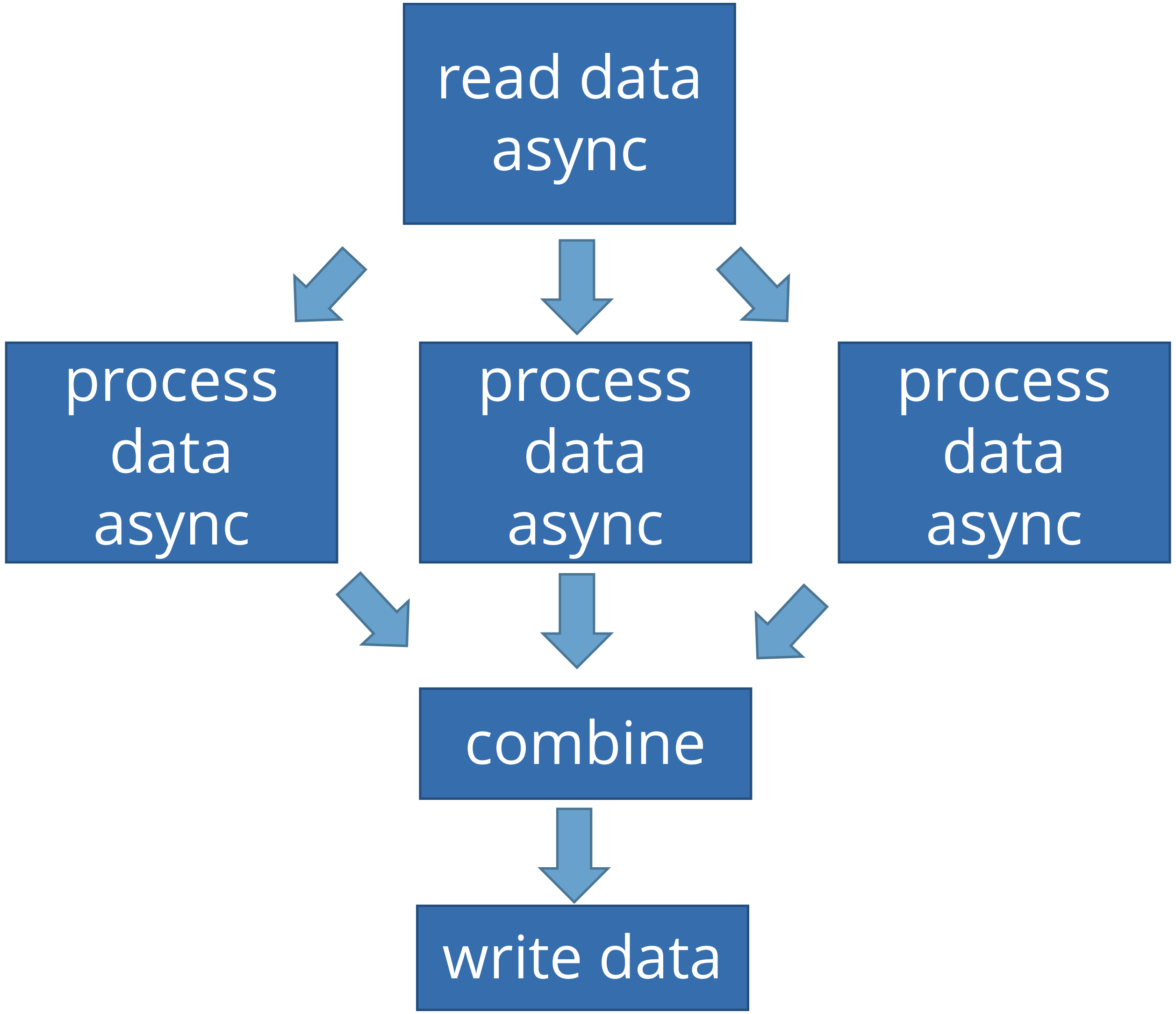
writeData.get()

⇒ we get synchronous code

We need to use it asynchronously…

But how?

Data flow



read data async

process data async

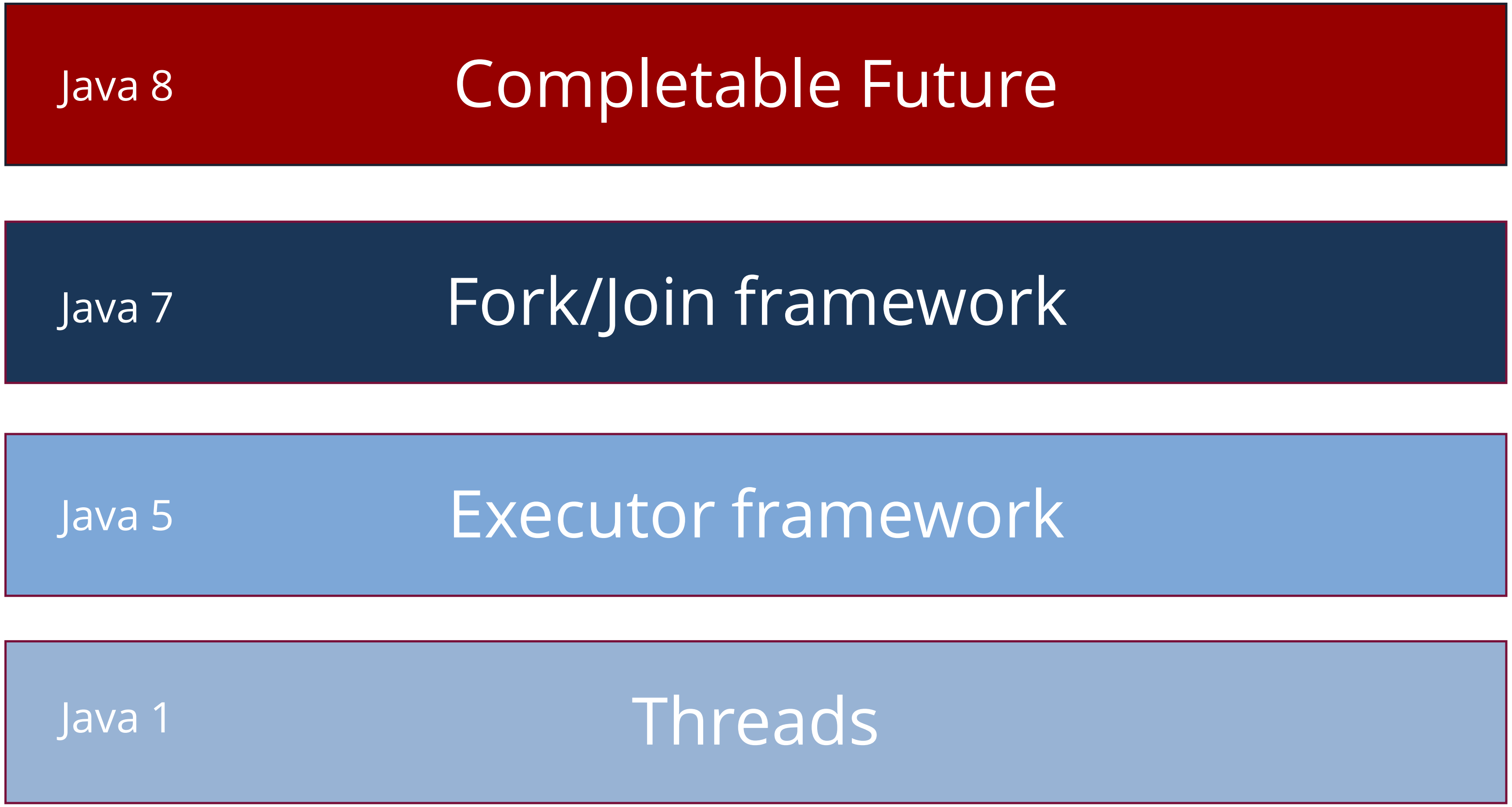
process data async

process data async

combine

write data

Here comes CompletableFuture!



Java 8 Completable Future

Java 7 Fork/Join framework

Java 5 Executor framework

Java 1 Threads

CompletableFuture

Long running method slowInit()

```
public Integer slowInit() {
        System.out.println("started task slowInit()");
        try {
                Thread.sleep(1000);
        } catch (InterruptedException e) {
                e.printStackTrace();
        }
        return 1;
}
```

Java 1.0-1.4: Plain old Java multithreading

```
int result;
public void testFutureOldStyle() throws InterruptedException {
    Thread t = new Thread() {
        public void run() {
            result = slowInit();
        };
    };
    t.start();
    t.join();
    System.out.println("futureTest() is finished: "+
        result);
}
```

```
started task slowInit()
futureTest() is finished: 1
```

Java 5 – 7: using Executor to run slowInit()

```
public void futureTest()
        throws InterruptedException, ExecutionException {

        Callable<Integer> r = this::slowInit;
        ExecutorService es =
                        Executors.newFixedThreadPool(10);
        Future<Integer> future = es.submit(r);

        Integer res = future.get();

        System.out.println("futureTest() is finished: "
                +res);
}
```

```
started task slowInit()
futureTest() is finished: 1
```

Java 8: using CompletableFuture to run slowInit()

```
public void promiseTest()
        throws InterruptedException, ExecutionException {

        CompletableFuture<Integer> future =
                CompletableFuture.supplyAsync(this::slowInit);

        Integer res = future.get();

        System.out.println("promiseTest() is finished: "
                +res);
}
```

```
started task 1
promiseTest() is finished: 1
```

Using CompletableFuture to execute several tasks

```
public void promiseTestNext()
throws InterruptedException, ExecutionException {
        CompletableFuture<Void> future =
                CompletableFuture
                .supplyAsync(this::slowInit)
                .thenAccept(
                         (res) -> { System.out.println("finished "+res); }
                        )
                .thenRun(
                        () -> { System.out.println("look at results"); }
                );
        future.get();
        System.out.println("promiseTestNext() is finished");
}
```

```
started task 1
finished 1
look at results
promiseTestNext() is finished
```

Long running method slowIncrement()

```
public Integer slowIncrement(Integer i) {
        try {
                Thread.sleep(1000);
        } catch (InterruptedException e) {
                e.printStackTrace();
        }
        System.out.println(
                "finished increment with result "+(i+1));
        return 1+i;
}
```

Using CompletableFuture to execute several tasks

```
public void promiseTestInc() throws Exception {
        long start = System.nanoTime();

        CompletableFuture<?> future =
                        CompletableFuture.supplyAsync(this::slowInit) // 1
                                .thenApply(this::slowIncrement) // 2
                                .thenApply(this::slowIncrement) // 3
                                .thenAccept( res ->
                                        System.out.println("async result: "+res) );
        future.get();

        long elapsedTime = System.nanoTime() - start;
        System.out.printf("%d sec passed",
                        TimeUnit.NANOSECONDS.toSeconds(elapsedTime));
}
```

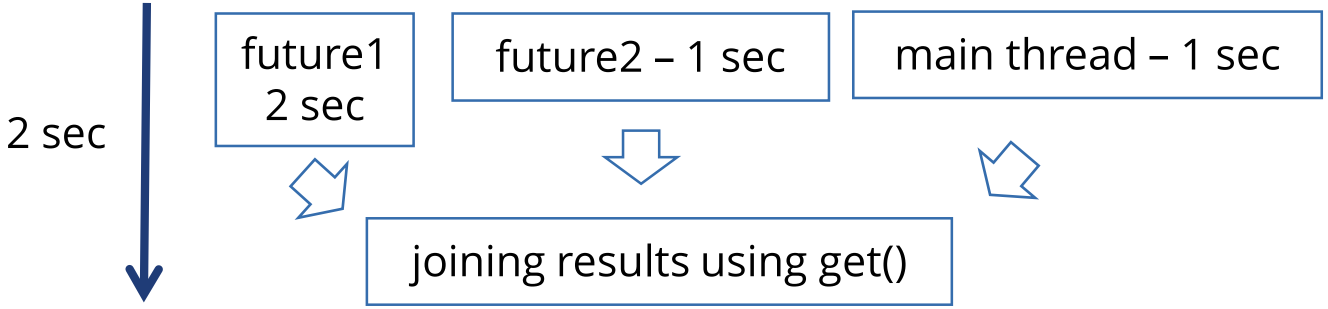
```
started task 1
finished increment with result 2
finished increment with result 3
async result: 3
3 sec passed
```

CompletableFutureTuto

Starting several pipelines from the main thread

```
CompletableFuture<Integer> future1 =
        CompletableFuture.supplyAsync(this::slowInit)
        .thenApply(this::slowIncrement);
CompletableFuture<Integer> future2 =   CompletableFuture.supplyAsync(this::slowInit);
Integer res0 = slowInit(); // here we are able to do self work
// then we are joining to the task results
Integer res1 = future1.get();
Integer res2 = future2.get();
System.out.println("tasks are finished with results ”
        +res0+", "+res1+" and "+res2);
```

```
tasks are finished
with results 1, 2, 1
```



2 sec

future1
2 sec

future2 – 1 sec

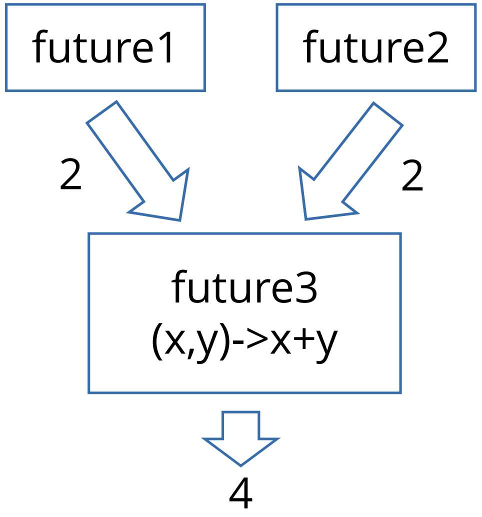
main thread – 1 sec

joining results using get()

Combining Futures

```
public void testThenCombine() throws Exception {
  CompletableFuture<Integer> future1 = CompletableFuture
           .supplyAsync(this::slowInit)
           .thenApply(this::slowIncrement); // 2
  CompletableFuture<Integer> future2 =
           CompletableFuture
           .supplyAsync(this::slowInit)
           .thenApply(this::slowIncrement); // 2
  CompletableFuture<?> future3 = future1
           .thenCombine(future2, (x,y)->x+y); // 4
  System.out.println("result: "+future3.get()); // result: 4
} // 2 sec passed
```

Pipeline:



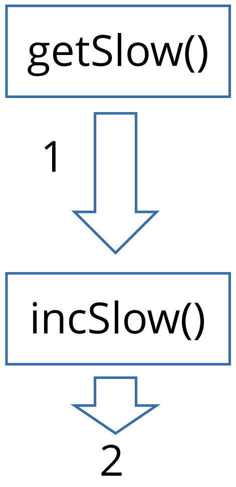
CombineTutor

Composing Futures

thenCompose() is used to compose functions returning CompletableFuture

```java
public CompletableFuture<Integer> getSlow() {
   sleep(1000);
   return CompletableFuture.completedFuture(1);
}
public CompletableFuture<Integer> incSlow(int i) {
   sleep(1000);
   return CompletableFuture.completedFuture(i+1);
}
```

Pipeline:



```java
getSlow()
   .thenApply(r->t.incSlow(r))
   .thenAccept(System.out::println);
```

```
java.util.concurrent.CompletableFuture
@568db2f2[Completed normally]
```

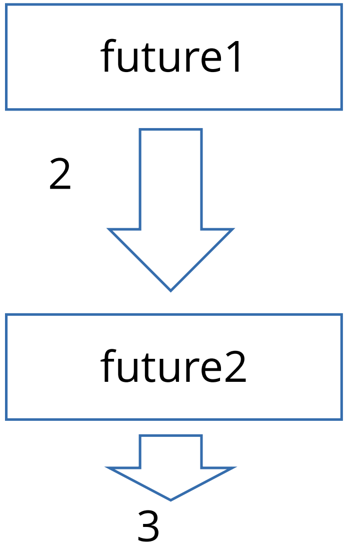
```java
getSlow()
   .thenCompose(r->t.incSlow(r))
   .thenAccept(System.out::println)
```

```
2
```

Composing Futures to create pipeline

```
public void promiseTestCompose2() throws Exception {
        CompletableFuture<Integer> future1 =
                        CompletableFuture.supplyAsync(this::slowInit) // 1
                                .thenApply(this::slowIncrement); // 2

        CompletableFuture<Integer> thenCompose =
                future1.thenCompose(
                        res -> CompletableFuture.supplyAsync(()->res)
                .thenApply(this::slowIncrement) ); // 3

        System.out.println(thenCompose.get());
}
```

Pipeline:



future1

2

future2

3

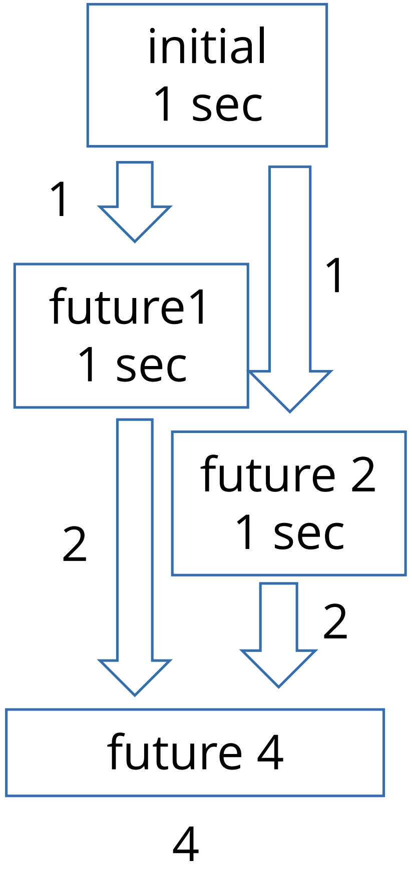
```
started task 1
finished increment with result 2
finished increment with result 3
3
```

ComposeTutor

Synchronous then – sequential execution

```
public void testThenCombineSync() throws Exception {
  CompletableFuture<Integer> initial =
              CompletableFuture.supplyAsync(this::slowInit);
  CompletableFuture<Integer> future1 =
              initial.thenApply(this::slowIncrement);
  CompletableFuture<Integer> future2 =
              initial.thenApply(this::slowIncrement);
  CompletableFuture<Integer> future3 =
              future1.thenCombine(future2, (x,y)->x+y);
  System.out.println("result: "+future3.get());
} // 3 sec passed
```

Pipeline:

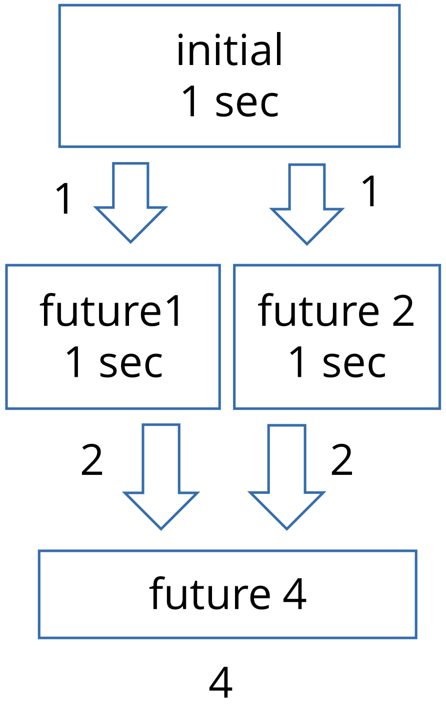


Methods without async execute task in the same thread as the previous task.

Asynchronous then – parallel execution

```
public void testThenCombineSync() throws Exception {
  CompletableFuture<Integer> initial =
              CompletableFuture.supplyAsync(this::slowInit);
  CompletableFuture<Integer> future1 =
              initial.thenApplyAsync(this::slowIncrement);
  CompletableFuture<Integer> future2 =
              initial.thenApplyAsync(this::slowIncrement);
  CompletableFuture<Integer> future3 =
              future1.thenCombine(future2, (x,y)->x+y);
  System.out.println("result: "+future3.get());
} // 2 sec passed
```

Pipeline:



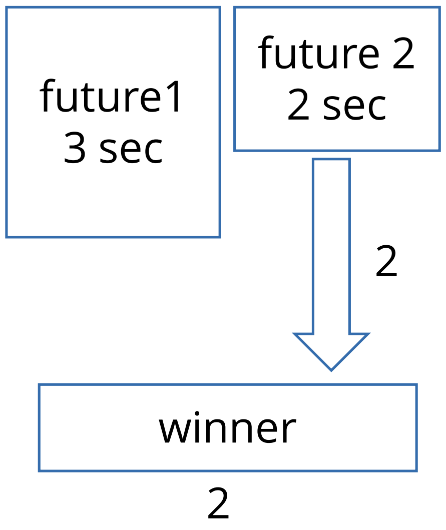
Methods with async execute task in the separate thread.

CombineTutor

AnyOf – get the winner of competition

```
public void testAnyOf() throws Exception {
  CompletableFuture<Integer> future1 =
            CompletableFuture.supplyAsync(this::slowInit) // 1
            .thenApply(this::slowIncrement) // 2
            .thenApply(this::slowIncrement); // 3
  CompletableFuture<Integer> future2 =
            CompletableFuture.supplyAsync(this::slowInit) // 1
            .thenApply(this::slowIncrement); // 2
  CompletableFuture<?> winner =
            CompletableFuture.anyOf(future1, future2);
  System.out.println("result: "+winner.get()); // result: 2
} // 2 sec passed
```

Pipeline:

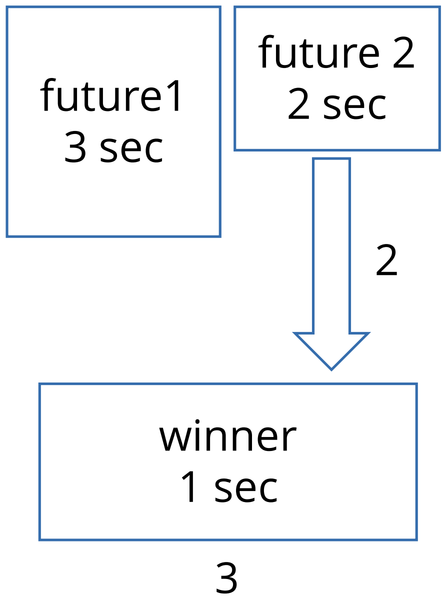


WinnerTutor

applyToEither – apply function to the winner of competition

```
public void testApplyToEither() throws Exception {
  CompletableFuture<Integer> future1 =
              CompletableFuture.supplyAsync(this::slowInit)
              .thenApply(this::slowIncrement)
              .thenApply(this::slowIncrement);
  CompletableFuture<Integer> future2 =
              CompletableFuture.supplyAsync(this::slowInit)
              .thenApply(this::slowIncrement);
  CompletableFuture<Integer> winner = future1
              .applyToEither(future2, this::slowIncrement);
  System.out.println("result: "+winner.get()); // result: 3
} // 3 sec passed
```

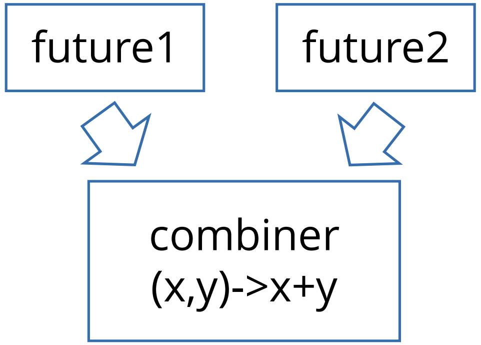
Pipeline:



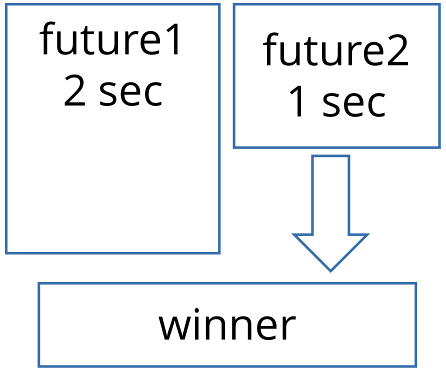
CompletableFuture methods summary

Input	Output	Sync	Async
-	+		supplyAsync
+	+	thenApply	...Async
+	-	thenAccept	...Async
-	-	thenRun	runAsync
+ other Future: combining			
+	+	thenCombine	...Async
+	-	runAfterBoth	...Async
-	-	allOf	
+ other Future: the quicker wins			
+	+	applyToEither	...Async
+	-	acceptEither	...Async
-	-	anyOf	
+ other Future: composing			
+	+	thenCompose	...Async

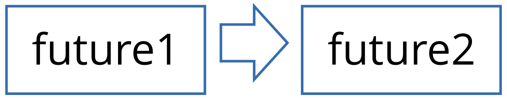
Combining:



Quicker wins:



Composing:



Handling exceptions: method exceptionally

```
CompletableFuture<Integer> future =
        CompletableFuture.supplyAsync(this::slowInit)
        .thenApply(this::slowIncrementException)
        .thenApply(this::slowIncrement)
        // this function will be executed only in case of Exception
        .exceptionally(ex -> {
                System.out.println("exception happened!");
                return 0;
        }).thenApply(this::slowIncrement);
        Integer result = future.get();
        System.out.println(result);
```

```
if exception raised:
exception happened!
1

if there was no
exception:
4
```

Handling exceptions: method handle

```java
CompletableFuture<Integer> future =
        CompletableFuture.supplyAsync(this::slowInit)
        .thenApply(this::slowIncrementException)
        .handle((ok, ex) -> {
                if (ex!=null) System.out.println("exception happened");
                return ok==null?0:ok; // return ok or null if exception
                // 0 is the replacement result that may enable
                // further processing by other dependent stages
        }).thenApply(this::slowIncrement);

Integer result = future.get();
System.out.println(result);
```

```
exception happened
1
```

Cancellation of CompletableFuture

```
CompletableFuture<Integer> future =
        CompletableFuture.supplyAsync(this::slowInit)
                .thenApplyAsync(this::slowIncrement)
                .thenApplyAsync(this::slowIncrement); // only last is cancelled
future.cancel(true); // mayInterruptIfRunning - no matter true or false
System.out.println(future.isCancelled()); // true
try {
    future.get(); // CancellationException
} catch (Exception e) {e.printStackTrace();}
```

```
true
java.util.concurrent.CancellationException
	at
	java.util.concurrent.CompletableFuture.cancel(CompletableFuture.java:2263)
	at completable.CancelTutor.test(CancelTutor.java:16)
	at completable.CancelTutor.main(CancelTutor.java:27)
started task slowInit()
slowIncrement()
finished increment with result 2
```

Pass executor to async methods

We can customize the executor (but only for methods ending on Async)

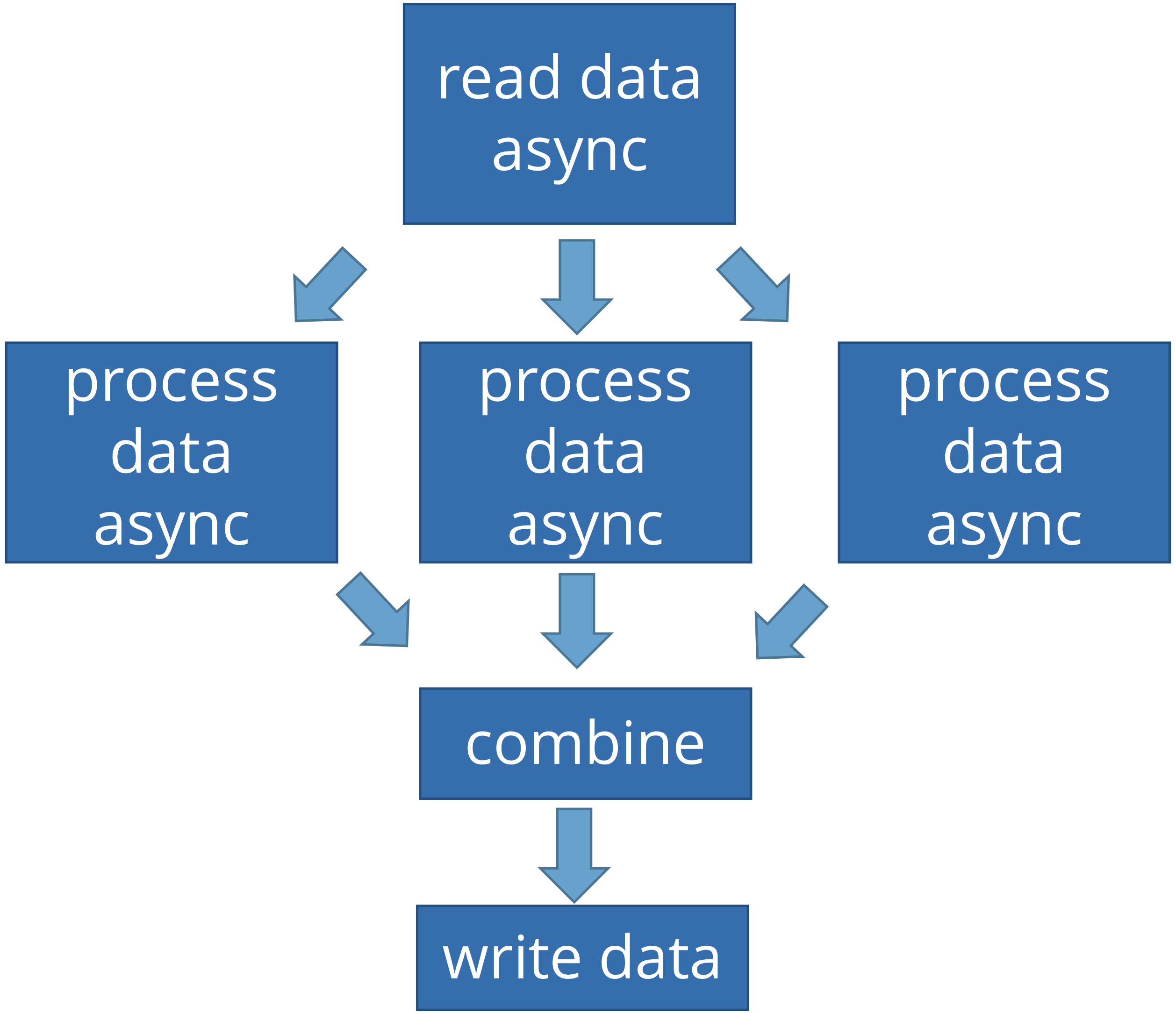
```
ExecutorService executorService = Executors.newFixedThreadPool(10);


CompletableFuture<Integer> future =
CompletableFuture.supplyAsync(this::slowInit, executorService)
.thenApplyAsync(this::slowIncrement, executorService);
```

Custom CompletableFuture

We can manually create CompletableFuture for asynchronous process.

```
public CompletableFuture<String> getCF() {
    CompletableFuture<String> cf = new CompletableFuture<>();
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        e.printStackTrace();
        cf.completeExceptionally(e);
    }
    cf.complete("result");

    return cf;
}
```

Data flow



read data async

process data async

process data async

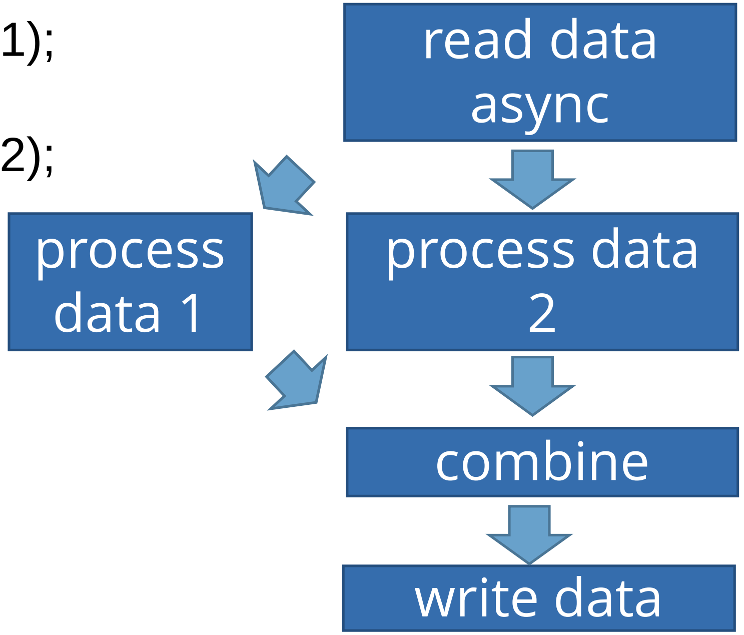
process data async

combine

write data

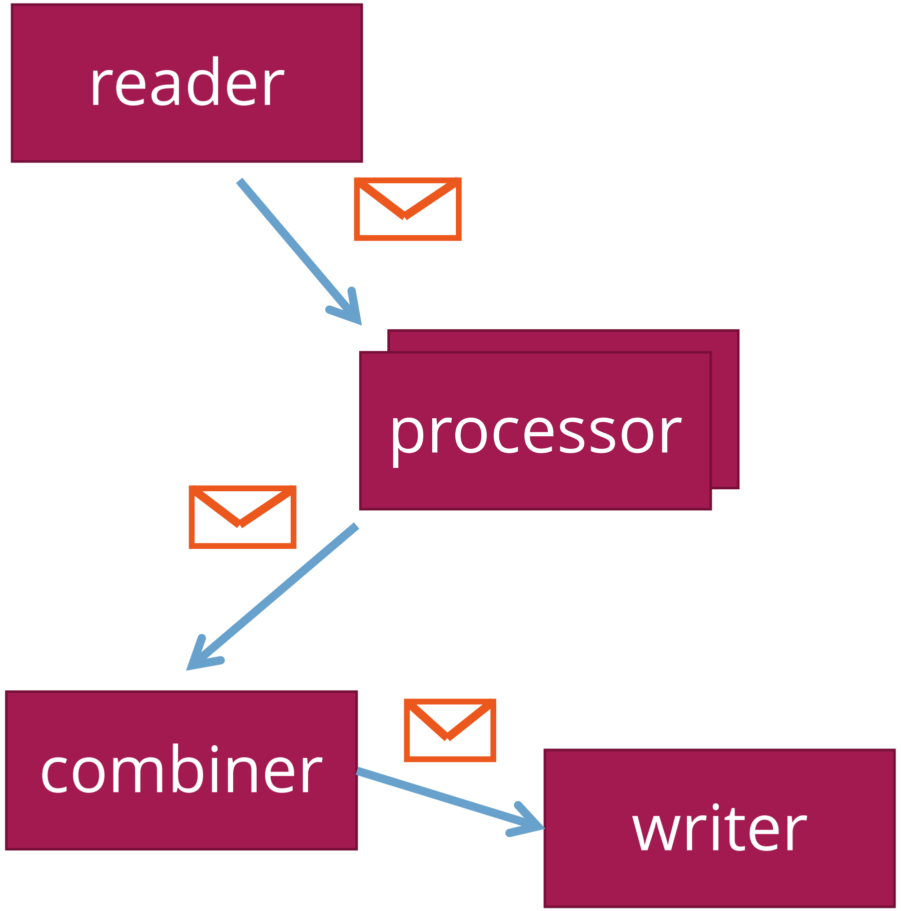
Using CompletableFuture for real-life Data Flow

```
// API
public CompletableFuture<Data> readData(Source source);
public Data processData1(Data data);
public Data processData2(Data data);
public Data mergeData(Data a, Data b);
public void writeData(Data data, Destination dest);

CompletableFuture<Data> data = readData(source);
CompletableFuture<Data> processData1 =
            data.thenApplyAsync(this::processData1);
CompletableFuture<Data> processData2 =
            data.thenApplyAsync(this::processData2);

processData1
            .thenCombine(processData2,
                        (a, b)->mergeData(a,b))
            .thenAccept(
                        data->writeData(data, dest));
```



Using events for asynchronous calls

event-based



reader

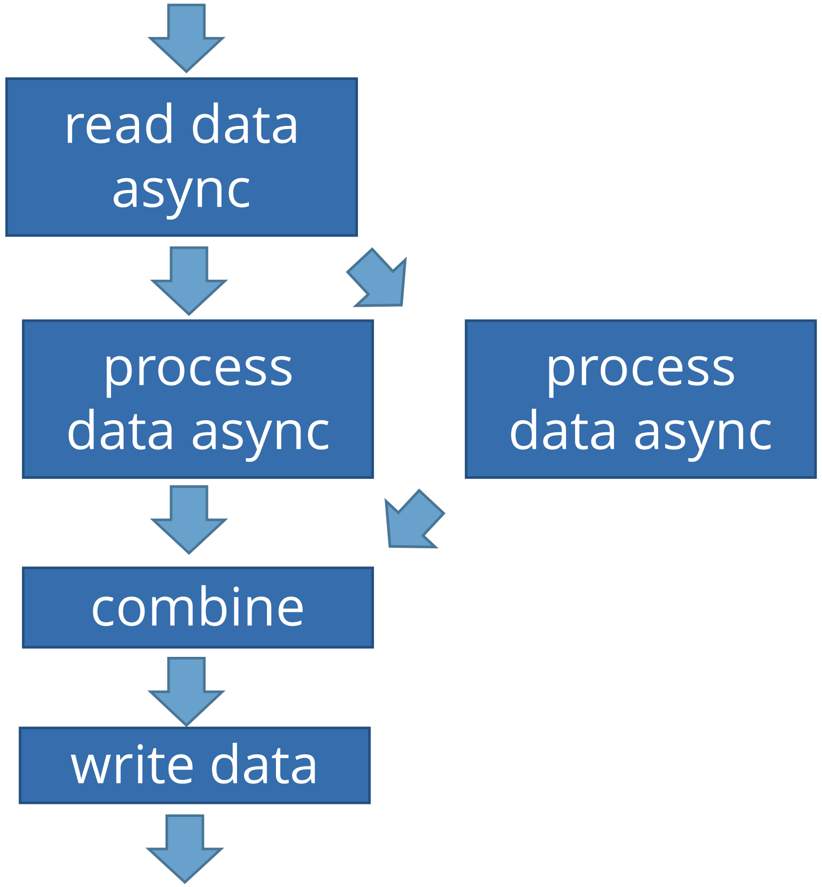
processor

combiner

writer

JMS, AKKA

data flow



read data async

process data async

process data async

combine

write data

CompletableFuture

Java technologies supporting asynchronous

- Servlets
- JAX-RS – asynchronous on server and client
- EJB
- WebSocket
- NIO (but has no CompletableFuture support)
- Spring MVC, Spring REST
- …
- but not JDBC!..

Thank You!

think.
create.
accelerate.

luxoft | training
A DXC Technology Company