Lecture 12

CompletableFuture: Composable Asynchronous Programming

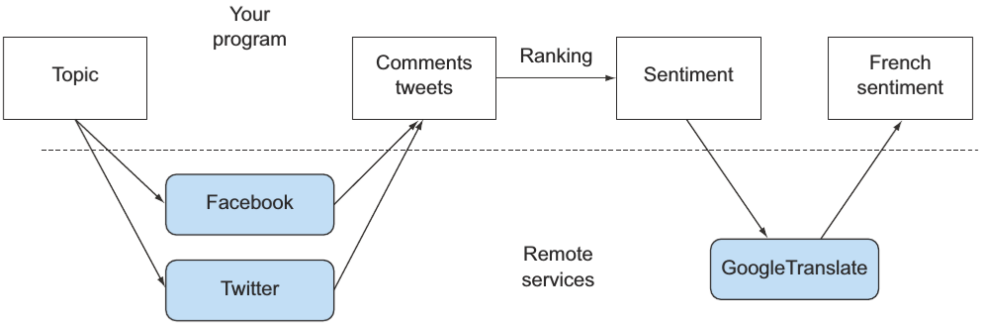
SOEN 6441, Summer 2018

René Witte
Department of Computer Science
and Software Engineering
Concordia University

Outline

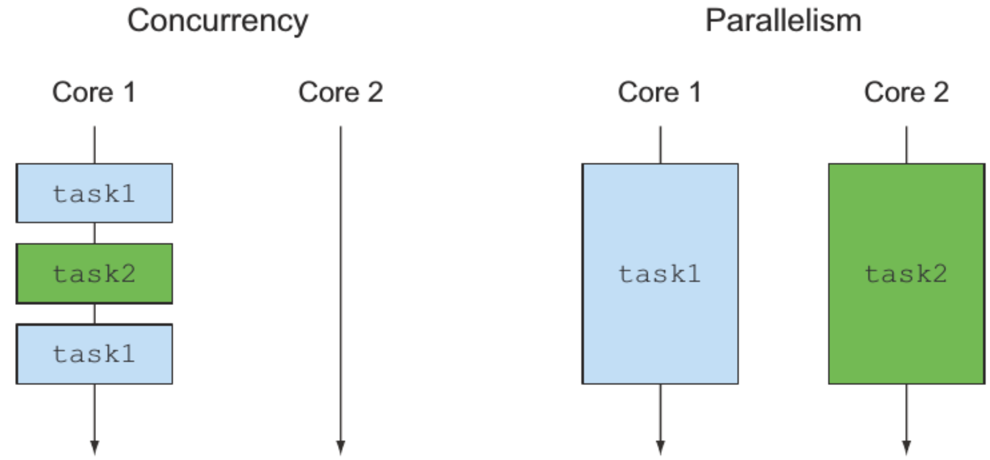
1. Motivation
2. Futures
3. Implementing an asynchronous API
4. Non-blocking code
5. Pipelining asynchronous tasks
6. Reacting to a CompletableFuture completion
7. Summary
8. Notes and Further Reading

A typical mash-up application



Copyright 2015 by Manning Publications Co., [UFM14]

Concurrency vs. Parallelism



Copyright 2015 by Manning Publications Co., [UFM14]

Outline

1 Motivation

2 **Futures**
- Java 5 Future
- Java 8 CompletableFuture

3 Implementing an asynchronous API

4 Non-blocking code

5 Pipelining asynchronous tasks

6 Reacting to a CompletableFuture completion

7 Summary

8 Notes and Further Reading

Futures

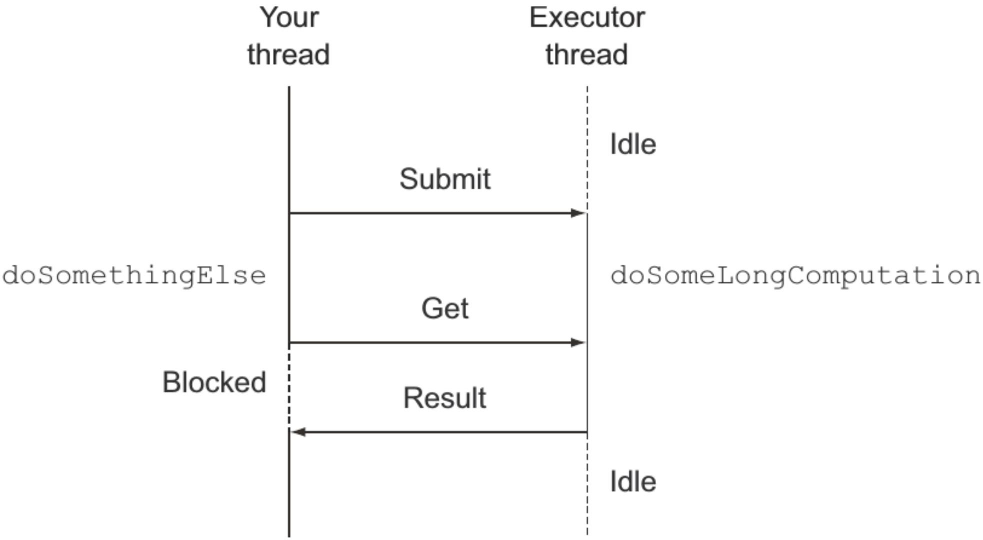
Futures before Java 8

```
ExecutorService executor = Executors.newCachedThreadPool();
Future<Double> future = executor.submit(new Callable<Double>() {
  public Double call() {
    return doSomeLongComputation();
  }});

doSomethingElse();

try {
  Double result = future.get(1, TimeUnit.SECONDS);
} catch (ExecutionException ee) {
  // the computation threw an exception
} catch (InterruptedException ie) {
  // the current thread was interrupted while waiting
} catch (TimeoutException te) {
  // the timeout expired before the Future completion
}
```

Using a `Future` to execute a long operation asynchronously



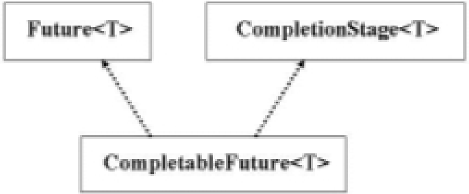
Copyright 2015 by Manning Publications Co., [UFM14]

Java 5 Futures’ limitations

Missing features before Java 8

- Combining two asynchronous computations in one – both when they’re independent and when the second depends on the result of the first
- Waiting for the completion of all tasks performed by a set of `Future`s
- Waiting for the completion of only the quickest task in a set of `Future`s (possibly because they’re trying to calculate the same value in different ways) and retrieving its result
- Programmatically completing a `Future` (that is, by manually providing the result of the asynchronous operation)
- Reacting to a `Future` completion (that is, being notified when the completion happens and then having the ability to perform a further action using the result of the `Future`, instead of being blocked waiting for its result)

Using `CompletableFutures` to build an asynchronous application



New `CompletableFuture` in Java 8

Example: online shop finding best prices

- Develop an asynchronous API for your customer
- Make code non-blocking for a consumer of a synchronous API.
- Pipeline two subsequent asynchronous operations, merging them into a single asynchronous computation
- Reactively process events representing the completion of an asynchronous operation

Synchronous vs. asynchronous API

Synchronous API

Call method, wait for result (blocking call)

Asynchronous API

Call method, return immediately (non-blocking call)

Outline

1. Motivation
2. Futures
   - Java 5 Future
   - Java 8 CompletableFuture
3. **Implementing an asynchronous API**
   - Convert into asynchronous
   - Dealing with errors
4. Non-blocking code
   - Parallelizing requests using parallel Stream
   - Making asynchronous requests
   - Using a custom Executor
5. Pipelining asynchronous tasks
   - Implementing a discount service
   - Using the Discount service
   - Composing synchronous and asynchronous operations
   - Combining two `CompletableFuture`s
6. Reacting to a CompletableFuture completion
   - Refactoring the best-price-finder application
   - Putting it to work
7. Summary
8. Notes and Further Reading

Implementing an asynchronous API

```
public class Shop {
  public double getPrice(String product) {
    // to be implemented
  }
}
```

Simulating processing delay (e.g., web service call)

```java
public static void delay() {
  try {
    Thread.sleep(1000L);
  } catch (InterruptedException e) {
    throw new RuntimeException(e);
  }
}
```

Introducing a simulated delay in the `getPrice` method

```
public double getPrice(String product) {
  return calculatePrice(product);
}

private double calculatePrice(String product) {
  delay();
  return random.nextDouble() * product.charAt(0) + product.charAt(1);
}
```

Converting a synchronous method into an asynchronous one

```
public Future<Double> getPriceAsync(String product) {
   ...
}
```

Implementing the getPriceAsync method

```java
public Future<Double> getPriceAsync(String product) {
  CompletableFuture<Double> futurePrice = new CompletableFuture<>();
  new Thread( () -> {
    double price = calculatePrice(product);
    futurePrice.complete(price);
  }).start();
  return futurePrice;
}
```

Using an asynchronous API

```java
Shop shop = new Shop("BestShop");
long start = System.nanoTime();
Future<Double> futurePrice = shop.getPriceAsync("my␣favorite␣product");
long invocationTime = ((System.nanoTime() - start) / 1_000_000);
System.out.println("Invocation␣returned␣after␣" + invocationTime
                                                + "␣msecs");

// Do some more tasks, like querying other shops
doSomethingElse();

// while the price of the product is being calculated
try {
  double price = futurePrice.get();
  System.out.printf("Price␣is␣%.2f%n", price);
} catch (Exception e) {
  throw new RuntimeException(e);
}

long retrievalTime = ((System.nanoTime() - start) / 1_000_000);
System.out.println("Price␣returned␣after␣" + retrievalTime + "␣msecs");
```

Output Example

```
Invocation returned after 43 msecs
Price is 123.26
Price returned after 1045 msecs
```

Dealing with errors

```
public Future<Double> getPriceAsync(String product) {
  CompletableFuture<Double> futurePrice = new CompletableFuture<>();
  new Thread( () -> {
    try {
      double price = calculatePrice(product);
      futurePrice.complete(price);
    } catch (Exception ex) {
      futurePrice.completeExceptionally(ex);
    }
  }).start();
  return futurePrice;
}
```

Product not available: `RuntimeException`

```
java.util.concurrent.ExecutionException: java.lang.RuntimeException:
    product not available
  at java.util.concurrent.CompletableFuture.get(CompletableFuture.java:2237)
  at lambdasinaction.chap11.AsyncShopClient.main(AsyncShopClient.java:14)
  ... 5 more
Caused by: java.lang.RuntimeException: product not available
  at lambdasinaction.chap11.AsyncShop.calculatePrice(AsyncShop.java:36)
  at lambdasinaction.chap11.AsyncShop.lambda$getPrice$0(AsyncShop.java:23)
  at lambdasinaction.chap11.AsyncShop$$Lambda$1/24071475.run(Unknown Source)
  at java.lang.Thread.run(Thread.java:744)
```

Creating a `CompletableFuture` with the `supplyAsync` Factory Method

```
public Future<Double> getPriceAsync(String product) {
  return CompletableFuture.supplyAsync(() -> calculatePrice(product));
}
```

René Witte

Motivation

Futures
Java 5 Future
Java 8 CompletableFuture

Implementing an asynchronous API
Convert into asynchronous
Dealing with errors

Non-blocking code
Parallelizing requests using parallel Stream
Making asynchronous requests
Using a custom Executor

Pipelining asynchronous tasks
Implementing a discount service
Using the Discount service
Composing synchronous and asynchronous operations
Combining two CompletableFutures

Reacting to a CompletableFuture completion
Refactoring the best-price-finder application
Putting it to work

Summary

Notes and Further Reading

Outline

1. Motivation
2. Futures
   - Java 5 Future
   - Java 8 CompletableFuture
3. Implementing an asynchronous API
   - Convert into asynchronous
   - Dealing with errors
4. **Non-blocking code**
   - Parallelizing requests using parallel Stream
   - Making asynchronous requests
   - Using a custom Executor
5. Pipelining asynchronous tasks
   - Implementing a discount service
   - Using the Discount service
   - Composing synchronous and asynchronous operations
   - Combining two `CompletableFutures`
6. Reacting to a CompletableFuture completion
   - Refactoring the best-price-finder application
   - Putting it to work
7. Summary
8. Notes and Further Reading

Querying multiple shops

```
List<Shop> shops = Arrays.asList(new Shop("BestPrice"),
                                 new Shop("LetsSaveBig"),
                                 new Shop("MyFavoriteShop"),
                                 new Shop("BuyItAll"));
```

A `findPrices` implementation sequentially querying all the shops

```java
public List<String> findPrices(String product) {
  return shops.stream()
              .map(shop -> String.format("%s␣price␣is␣%.2f",
                                         shop.getName(),
                                         shop.getPrice(product)))
              .collect(toList());
}
```

Checking `findPrices` correctness and performance

```
long start = System.nanoTime();
System.out.println(findPrices("myPhone27S"));
long duration = (System.nanoTime() - start) / 1_000_000;
System.out.println("Done␣in␣" + duration + "␣msecs");
```

```
[BestPrice price is 123.26, LetsSaveBig price is 169.47,
   MyFavoriteShop price is 214.13, BuyItAll price is 184.74]
Done in 4032 msecs
```

Parallelizing the findPrices method

```
public List<String> findPrices(String product) {
  return shops.parallelStream()
              .map(shop -> String.format("%s␣price␣is␣%.2f",
                                          shop.getName(),
                                          shop.getPrice(product)))
              .collect(toList());
}
```

```
[BestPrice price is 123.26, LetsSaveBig price is 169.47,
  MyFavoriteShop price is 214.13, BuyItAll price is 184.74]
Done in 1180 msecs
```

Making asynchronous requests with `CompletableFutures`

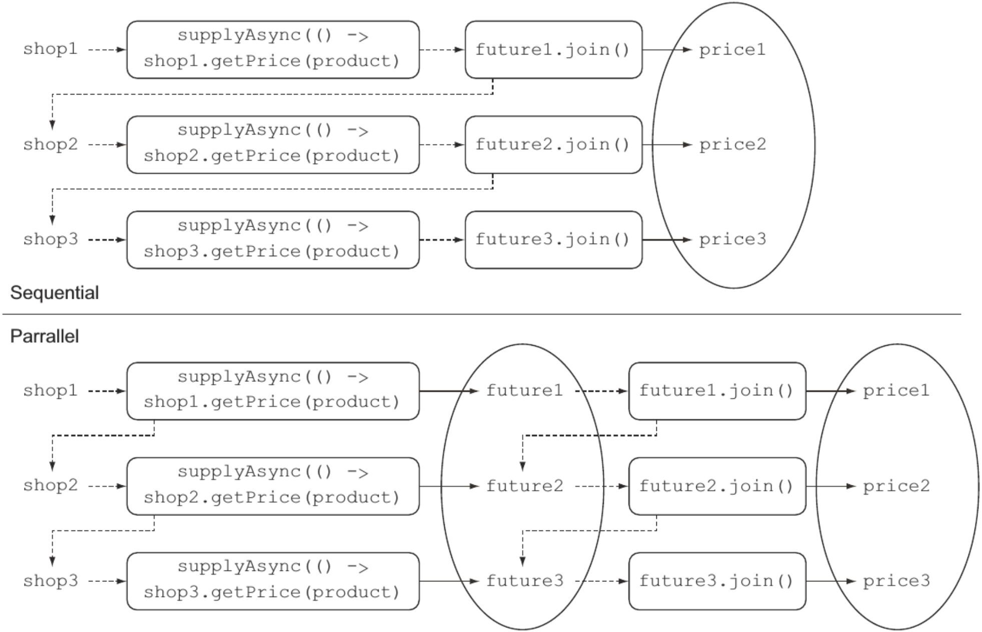
```
List<CompletableFuture<String>> priceFutures =
  shops.stream()
       .map(shop -> CompletableFuture.supplyAsync(
                      () -> String.format("%s␣price␣is␣%.2f",
                                          shop.getName(),
                                          shop.getPrice(product))))
       .collect(toList());
```

Implementing the `findPrices` method with `CompletableFutures`

```
public List<String> findPrices(String product) {
  List<CompletableFuture<String>> priceFutures =
    shops.stream()
         .map(shop -> CompletableFuture.supplyAsync(
                      () -> shop.getName() + "␣price␣is␣"
                           + shop.getPrice(product)))
         .collect(Collectors.toList());

    return priceFutures.stream()
                       .map(CompletableFuture::join)
                       .collect(toList());
}
```

Why `Stream`'s laziness causes a sequential computation and how to avoid it


shop1
supplyAsync(() -> shop1.getPrice(product)
future1.join()
price1
shop2
supplyAsync(() -> shop2.getPrice(product)
future2.join()
price2
shop3
supplyAsync(() -> shop3.getPrice(product)
future3.join()
price3
Sequential
Parrallel
shop1
supplyAsync(() -> shop1.getPrice(product)
future1
future1.join()
price1
shop2
supplyAsync(() -> shop2.getPrice(product)
future2
future2.join()
price2
shop3
supplyAsync(() -> shop3.getPrice(product)
future3
future3.join()
price3

Copyright 2015 by Manning Publications Co., [UFM14]

Performance (version with two `Stream` pipelines)

```
[BestPrice price is 123.26, LetsSaveBig price is 169.47,
  MyFavoriteShop price is 214.13, BuyItAll price is 184.74]
Done in 2005 msecs
```

(Running on a machine with four cores)

Looking for the solution that scales better

After adding a fifth shop

Sequential stream

```
[BestPrice price is 123.26, LetsSaveBig price is 169.47,
  MyFavoriteShop price is 214.13, BuyItAll price is 184.74,
  ShopEasy price is 176.08]
Done in 5025 msecs
```

Parallel stream

```
[BestPrice price is 123.26, LetsSaveBig price is 169.47,
  MyFavoriteShop price is 214.13, BuyItAll price is 184.74,
  ShopEasy price is 176.08]
Done in 2177 msecs
```

Using `CompletableFuture`s

```
[BestPrice price is 123.26, LetsSaveBig price is 169.47,
  MyFavoriteShop price is 214.13, BuyItAll price is 184.74,
  ShopEasy price is 176.08]
Done in 2006 msecs
```

Sizing thread pools

Finding optimal size for a thread pool

Calculate pool size for desired CPU utilization rate:

$$N_{\text{threads}} = N_{\text{CPU}} \times U_{\text{CPU}} \times \left(1 + \frac{W}{C}\right)$$

With

N_{CPU} number of cores, available through `Runtime.getRuntime().availableProcessors()`

U_{CPU} target CPU utilization (between 0 and 1)

W/C ratio of wait time to compute time

Shop example

Given a quad-core processor ($N_{\text{CPU}} = 4$)

- Application spends 99% of the time waiting for shops' responses
- Estimate W/C ratio of 100
- Target 100% CPU utilization

$\Rightarrow$ use 400 threads *(however, not realistic to use more than 1 thread/shop here)*

A custom `Executor` fitting our best-price-finder application

`Executor`

```
private final Executor executor =
    Executors.newFixedThreadPool(
      Math.min(shops.size(), 100),
      new ThreadFactory() {
        public Thread newThread(Runnable r) {
          Thread t = new Thread(r);
          t.setDaemon(true);
          return t;
        }
      });
```

Pool of *Daemon threads*

Can be terminated upon program completion (normal threads prevent Java from exiting)

Using the `executor`

```
CompletableFuture.supplyAsync(() -> shop.getName() + "␣price␣is␣" +
                                    shop.getPrice(product),executor);
```

Outline

1 Motivation

2 Futures

- Java 5 Future
- Java 8 CompletableFuture

3 Implementing an asynchronous API

- Convert into asynchronous
- Dealing with errors

4 Non-blocking code

- Parallelizing requests using parallel Stream
- Making asynchronous requests
- Using a custom Executor

5 Pipelining asynchronous tasks

- Implementing a discount service
- Using the Discount service
- Composing synchronous and asynchronous operations
- Combining two `CompletableFuture`s

6 Reacting to a CompletableFuture completion

- Refactoring the best-price-finder application
- Putting it to work

7 Summary

8 Notes and Further Reading

An enumeration defining the discount codes

```
public class Discount {
  public enum Code {
    NONE(0), SILVER(5), GOLD(10), PLATINUM(15), DIAMOND(20);

    private final int percentage;

    Code(int percentage) {
      this.percentage = percentage;
    }
  }

  // Discount class implementation
}
```

Calculating price with `DiscountCode`

Updated `getPrice`

```java
public String getPrice(String product) {
  double price = calculatePrice(product);
  Discount.Code code = Discount.Code.values()[
  random.nextInt(Discount.Code.values().length)];
  return String.format("%s:%.2f:%s", name, price, code);
}

private double calculatePrice(String product) {
  delay();
  return random.nextDouble() * product.charAt(0) + product.charAt(1);
}
```

Invoking `getPrice`

```
BestPrice:123.26:GOLD
```

Implementing a discount service

```
public class Quote {
  private final String shopName;
  private final double price;
  private final Discount.Code discountCode;

  public Quote(String shopName, double price, Discount.Code code) {
    this.shopName = shopName;
    this.price = price;
    this.discountCode = code;
  }

  public static Quote parse(String s) {
    String[] split = s.split(":");
    String shopName = split[0];
    double price = Double.parseDouble(split[1]);
    Discount.Code discountCode = Discount.Code.valueOf(split[2]);
    return new Quote(shopName, price, discountCode);
  }

  public String getShopName() { return shopName; }
  public double getPrice() { return price; }
  public Discount.Code getDiscountCode() { return discountCode; }
}
```

The Discount service

```
public class Discount {
  public enum Code {
    // source omitted ...
  }

  public static String applyDiscount(Quote quote) {
    return quote.getShopName() + "␣price␣is␣"
           + Discount.apply(quote.getPrice(),
                            quote.getDiscountCode());
  }

  private static double apply(double price, Code code) {
    delay();
    return format(price * (100 - code.percentage) / 100);
  }
}
```

Using the Discount service

```
public List<String> findPrices(String product) {
  return shops.stream()
              .map(shop -> shop.getPrice(product))
              .map(Quote::parse)
              .map(Discount::applyDiscount)
              .collect(toList());
}
```

```
[BestPrice price is 110.93, LetsSaveBig price is 135.58,
  MyFavoriteShop price is 192.72, BuyItAll price is 184.74,
  ShopEasy price is 167.28]
Done in 10028 msecs
```

René Witte

Motivation

Futures
Java 5 Future
Java 8 CompletableFuture

Implementing an asynchronous API
Convert into asynchronous
Dealing with errors

Non-blocking code
Parallelizing requests using parallel Stream
Making asynchronous requests
Using a custom Executor

Pipelining asynchronous tasks
Implementing a discount service
Using the Discount service
Composing synchronous and asynchronous operations
Combining two CompletableFutures

Reacting to a CompletableFuture completion
Refactoring the best-price-finder application
Putting it to work

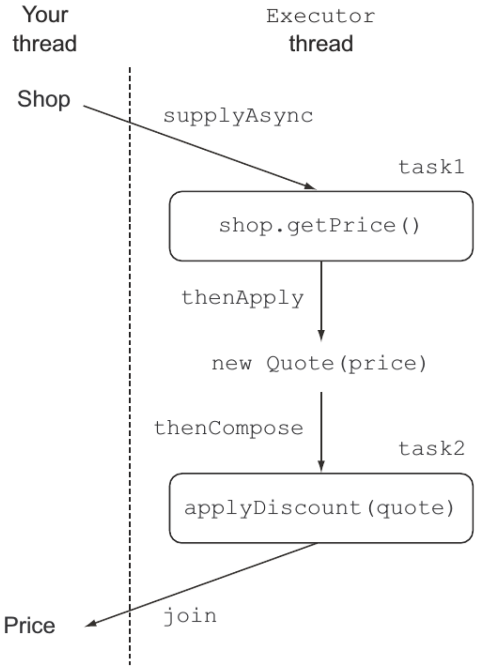
Summary

Notes and Further Reading

Implementing the `findPrices` method with `CompletableFutures`

```
public List<String> findPrices(String product) {
  List<CompletableFuture<String>> priceFutures =
    shops.stream()
         .map(shop -> CompletableFuture.supplyAsync(
                      () -> shop.getPrice(product), executor))
         .map(future -> future.thenApply(Quote::parse))
         .map(future -> future.thenCompose(quote ->
           CompletableFuture.supplyAsync(
             () -> Discount.applyDiscount(quote), executor)))
         .collect(toList());

  return priceFutures.stream()
                     .map(CompletableFuture::join)
                     .collect(toList());
}
```

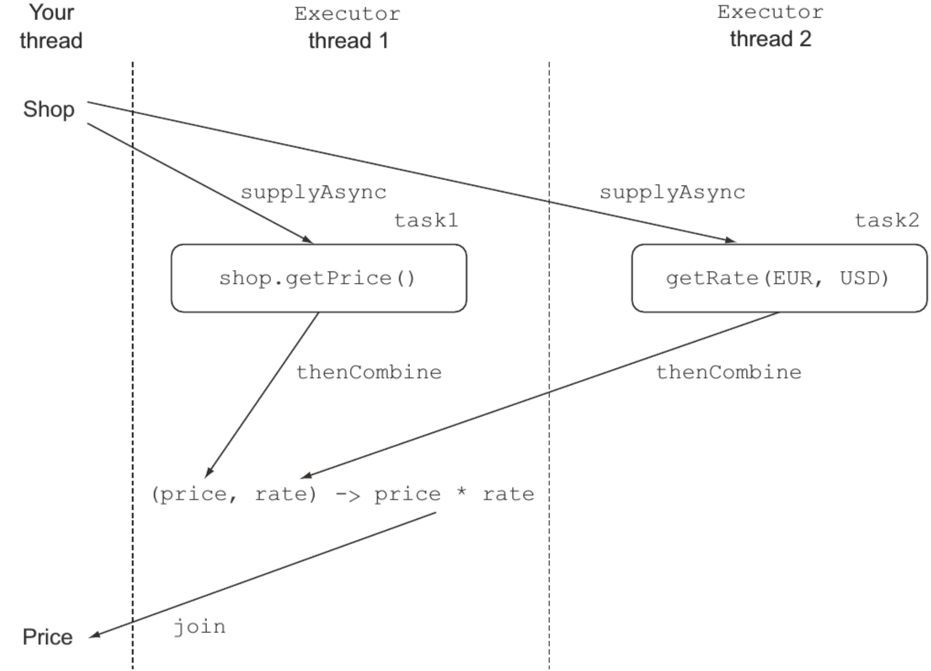


Copyright 2015 by Manning Publications Co., [UFM14]

Result

```
[BestPrice price is 110.93, LetsSaveBig price is 135.58,
  MyFavoriteShop price is 192.72, BuyItAll price is 184.74,
  ShopEasy price is 167.28]
Done in 2035 msecs
```

Combining two independent asynchronous tasks



Combining two independent `CompletableFuture`s

```
Future<Double> futurePriceInUSD =
CompletableFuture.supplyAsync(() -> shop.getPrice(product))
                 .thenCombine(
                   CompletableFuture.supplyAsync(
                     () -> exchangeService.getRate(Money.EUR,Money.USD)),
                   (price, rate) -> price * rate
                 ));
```

Outline

1 Motivation

2 Futures

- Java 5 Future
- Java 8 CompletableFuture

3 Implementing an asynchronous API

- Convert into asynchronous
- Dealing with errors

4 Non-blocking code

- Parallelizing requests using parallel Stream
- Making asynchronous requests
- Using a custom Executor

5 Pipelining asynchronous tasks

- Implementing a discount service
- Using the Discount service
- Composing synchronous and asynchronous operations
- Combining two `CompletableFuture`s

6 Reacting to a CompletableFuture completion

- Refactoring the best-price-finder application
- Putting it to work

7 Summary

8 Notes and Further Reading

A method to simulate a random delay between 0.5 and 2.5 seconds

```java
private static final Random random = new Random();
public static void randomDelay() {
  int delay = 500 + random.nextInt(2000);
  try {
    Thread.sleep(delay);
  } catch (InterruptedException e) {
    throw new RuntimeException(e);
  }
}
```

Refactoring the `findPrices` method to return a stream of `Futures`

```
public Stream<CompletableFuture<String>> findPricesStream(String product) {
  return shops.stream()
              .map(shop -> CompletableFuture.supplyAsync(
                           () -> shop.getPrice(product), executor))
              .map(future -> future.thenApply(Quote::parse))
              .map(future -> future.thenCompose(quote ->
                             CompletableFuture.supplyAsync(
                               () -> Discount.applyDiscount(quote), executor)));
}
```

React to completion

```
findPricesStream("myPhone").map(f -> f.thenAccept(System.out::println));
```

`thenAccept` vs. `thenAcceptAsync`

The `Async` variant schedules the execution of the `Consumer` on a new thread

Wait for completion of all `CompletableFuture`s

```
CompletableFuture[] futures = findPricesStream("myPhone")
   .map(f -> f.thenAccept(System.out::println))
   .toArray(size -> new CompletableFuture[size]);

CompletableFuture.allOf(futures).join();
```

`allOf` vs. `anyOf`

`allOf` returns `CompletableFuture<Void>` when all `CompletableFuture`s have completed

`anyOf` returns `CompletableFuture<Object>` with the value of the first-to-complete `CompletableFuture`

Putting it to work

```
long start = System.nanoTime();
CompletableFuture[] futures = findPricesStream("myPhone27S")
     .map(f -> f.thenAccept(
          s -> System.out.println(s + "␣(done␣in␣" +
               ((System.nanoTime() - start) / 1_000_000) + "␣msecs)")))
     .toArray(size -> new CompletableFuture[size]);

CompletableFuture.allOf(futures).join();

System.out.println("All␣shops␣have␣now␣responded␣in␣"
                        + ((System.nanoTime() - start) / 1_000_000)
                        + "␣msecs");
```

Output Example

```
BuyItAll price is 184.74 (done in 2005 msecs)

MyFavoriteShop price is 192.72 (done in 2157 msecs)

LetsSaveBig price is 135.58 (done in 3301 msecs)

ShopEasy price is 167.28 (done in 3869 msecs)

BestPrice price is 110.93 (done in 4188 msecs)

All shops have now responded in 4188 msecs
```

Summary

Futures and asynchronous programming

- Executing relatively long-lasting operations using asynchronous tasks can increase the performance and responsiveness of your application, especially if it relies on one or more remote external services.
- You should provide an asynchronous API to your clients, implemented using Java 8 `CompletableFuture`.
- A `CompletableFuture` also allows you to propagate and manage errors generated within an asynchronous task.
- You can asynchronously consume from a synchronous API by simply wrapping its invocation in a `CompletableFuture`.
- You can compose or combine multiple asynchronous tasks both when they're independent and when the result of one of them is used as the input to another.
- You can register a callback on a `CompletableFuture` to reactively execute some code when the `Future` completes and its result becomes available.
- You can determine when all values in a list of `CompletableFutures` have completed, or alternatively you can wait for just the first to complete.

Outline

1. Motivation
2. Futures
3. Implementing an asynchronous API
4. Non-blocking code
5. Pipelining asynchronous tasks
6. Reacting to a CompletableFuture completion
7. Summary
8. **Notes and Further Reading**

Reading Material

Required

- [UFM14, Chapter 11] (CompletableFuture)

Supplemental

- [War14, Chapter 9] (Lambda-Enabled Concurrency)

References

[UFM14] Raoul-Gabriel Urma, Mario Fusco, and Alan Mycroft.
Java 8 in Action: Lambdas, streams, and functional-style programming.
Manning Publications, 2014.
https://www.manning.com/books/java-8-in-action.

[War14] Richard Warburton.
Java 8 Lambdas.
O'Reilly, 2014.