

Concurrent and Asynchronous Programming

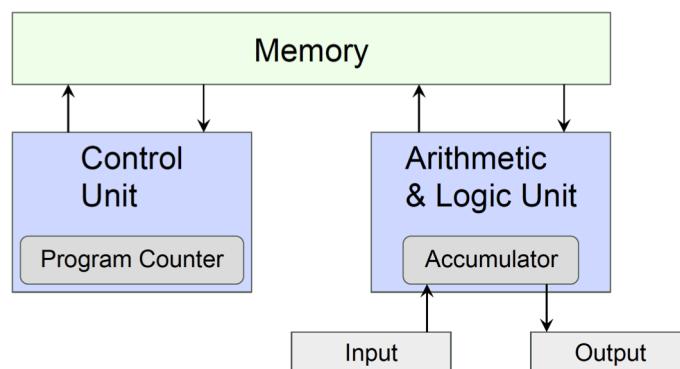


Some Definitions

- Concurrent
 - more than one task logically proceeding within a given time interval
- Parallel
 - more than one task executing at the same instant
- Can be concurrent without being parallel
 - single CPU, multi-tasking OS
- Parallelism implies concurrency

The Von Neumann Architecture

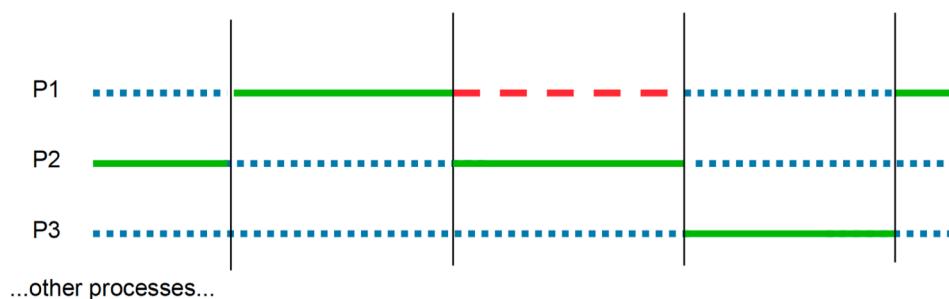
- Stored Program execution
 - Program stored in memory
 - Memory shared between program and data
 - Execution follows a sequential model



© J&G Services Ltd, 2017

Process Based Concurrency

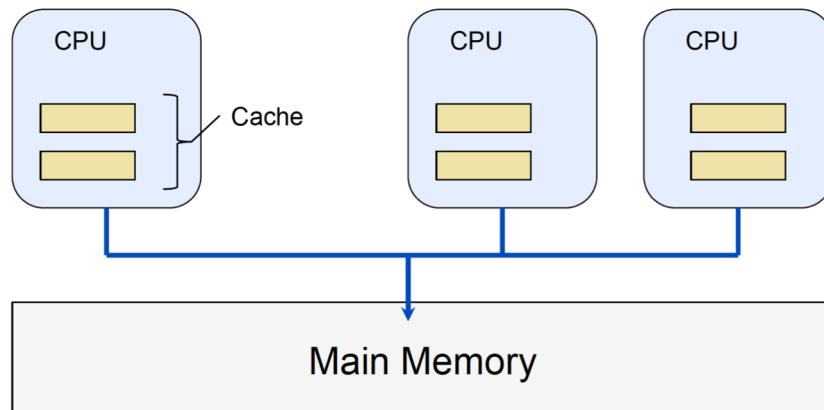
- Operating system schedules processes
 - concurrency achieved by sharing work across processes
 - can operate on single cpu core – no parallelism



© J&G Services Ltd, 2017

Multiprocessor/Multicore

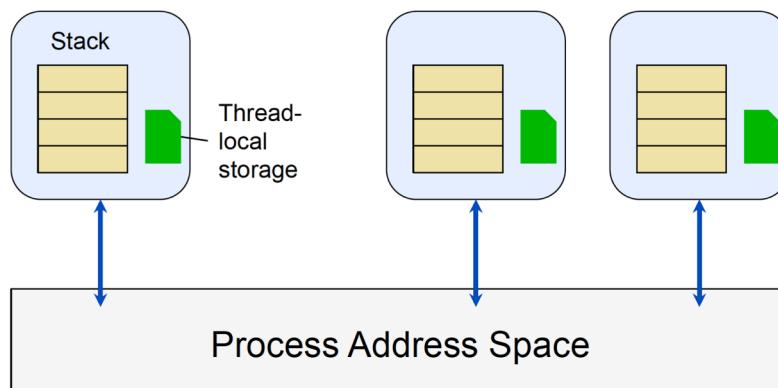
- Multiple CPUs (cores) sharing memory



© J&G Services Ltd, 2017

Traditional Threading Model

- Aim is to provide more lightweight concurrency model
- Multiple threads share the same address space



© J&G Services Ltd, 2017

Threads and Processes

- Thread

- a sequence of instructions that can execute independently (of other threads)
- on multi-core CPUs threads can execute in parallel
- on single-core CPUs threads are time-sliced

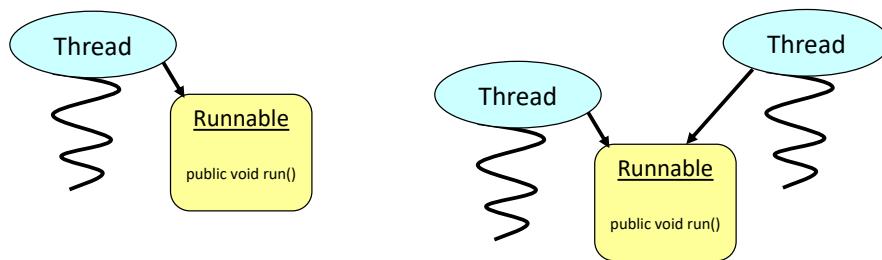
- Process

- provides resources needed to execute a program
- e.g. address space, security context, environment
- a process will contain at least one thread

© J&G Services Ltd, 2017

JVM Threading: Threads and Runnable Objects

- Thread encapsulates a schedulable entity
- Runnable interface used to represent work for thread to do



© J&G Services Ltd, 2017

Creating a Thread

- Instantiate Thread class
- Associate with Runnable object
- Start execution of thread
 - start() method invokes run() on the Runnable object

```

...
Runnable r = new RunnableImpl();
Thread t1 = new Thread( r );
Thread t2 = new Thread( r );
t1.start();
t2.start();
...

```



```

...
Thread t1 = new Thread( () -> { ... } );
t1.start();
...

```

© J&G Services Ltd, 2017

Creating a Thread

- Thread class implements Runnable
 - empty run() method
- Define subclass of Thread
 - override run() method with required action
- Create Thread object via subclass

```

public class SubThread
    extends Thread
{
    @Override
    public void run()
    {
        doSomething();
    }
}

SubThread s = new SubThread();
s.start();

```

© J&G Services Ltd, 2017

Creating a Thread

- Creating `Thread` object doesn't create a thread
- The `start()` method creates the thread

```
Runnable r = () -> doSomething();
(new Thread(r)).start();
```

- Calls out to the operating system to create and register the thread

© J&G Services Ltd, 2017

Thread Example

```
public class TickTock implements Runnable {
    private String word;
    private int delay;
    public TickTock(String w, int d) {
        word = w;
        delay = d;
    }

    public void run() {
        while (!Thread.interrupted()) { // More later
            try {
                System.out.print(word + " ");
                Thread.sleep(delay);
            } catch (InterruptedException ie) {
                Thread.currentThread().interrupt();
            }
        }
    }
}

public static void main(String[] args) {
    Thread t1 = new Thread(new TickTock("tick", 50));
    Thread t2 = new Thread(new TickTock("tock", 100));
    t1.start(); t2.start();
    Thread.sleep(5000); t1.interrupt(); t2.interrupt();
}
```

© J&G Services Ltd, 2017

Thread Example

```

public class TickTock2 extends Thread {
    private String word;
    private int delay;
    public TickTock2(String w, int d) {
        word = w;
        delay = d;
    }

    public void run() {
        while (!Thread.interrupted()) {
            try {
                System.out.print(word + " ");
                Thread.sleep(delay);
            } catch (InterruptedException ie) {
                Thread.currentThread().interrupt();
            }
        }
    }
}

public static void main(String[] args) {
    TickTock2 tt1 = new TickTock2("tick", 50);
    TickTock2 tt2 = new TickTock2("tock", 100);
    tt1.start(); tt2.start();
    Thread.sleep(5000); tt1.interrupt(); tt2.interrupt();
}

```

© J&G Services Ltd, 2017

Thread Example

```

public class TickTock implements Runnable {
    private String word;
    private int delay;
    public TickTock(String w, int d) {
        word = w;
        delay = d;
    }

    public void run() {
        while (!Thread.interrupted())
            try {
                System.out.print(word + " ");
                Thread.sleep(delay);
            } catch (InterruptedException ie) {
                Thread.currentThread().interrupt();
            }
    }
}

import java.util.concurrent._;
...

public static void main(String[] args) {
    Runnable r1 = new TickTock("tick", 50);
    Runnable r2 = new TickTock("tock", 100);

    Executor engine = Executors.newFixedThreadPool(2);

    engine.execute(r1);
    engine.execute(r2);

    Thread.sleep(5000);
    engine.shutdownNow();
}

```

© J&G Services Ltd, 2017

Problems

- Limited capabilities
 - run method returns void
- Scalability is limited
 - threads relatively heavyweight objects
 - limited number can be supported in VM
- Shared state difficult to protect
 - locks/synchronized blocks
 - introduces complexity to code
 - difficult to debug or reason about
 - blocking threads wastes resources



© J&G Services Ltd, 2017

Problems

Why Threads Are A Bad Idea (for most purposes)

John Ousterhout
Sun Microsystems Laboratories

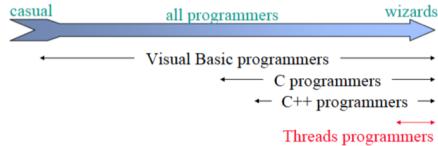
john.ousterhout@eng.sun.com
<http://www.sunlabs.com/~ouster>



© J&G Services Ltd, 2017

Problems

What's Wrong With Threads?



- ▼ Too hard for most programmers to use.
- ▼ Even for experts, development is painful.

Why Threads Are A Bad Idea

September 28, 1995, slide 5

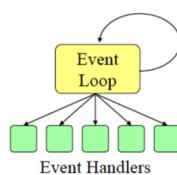


© J&G Services Ltd, 2017

Problems

Event-Driven Programming

- ▼ One execution stream: no CPU concurrency.
- ▼ Register interest in events (callbacks).
- ▼ Event loop waits for events, invokes handlers.
- ▼ No preemption of event handlers.
- ▼ Handlers generally short-lived.



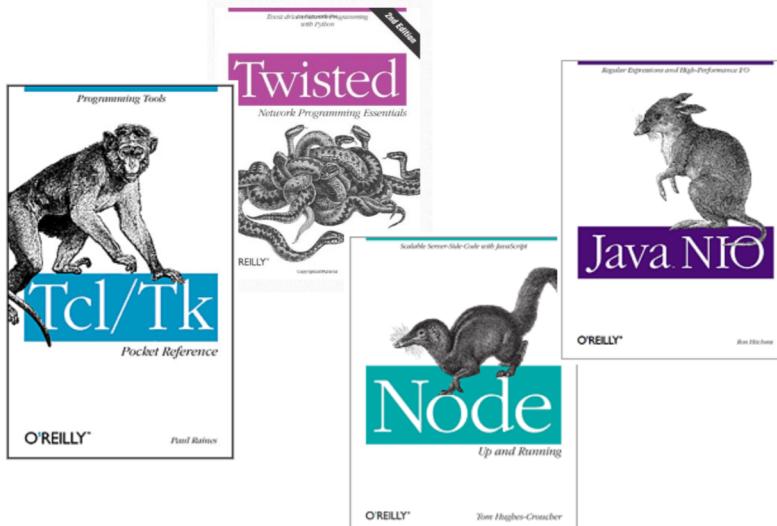
Why Threads Are A Bad Idea

September 28, 1995, slide 9



© J&G Services Ltd, 2017

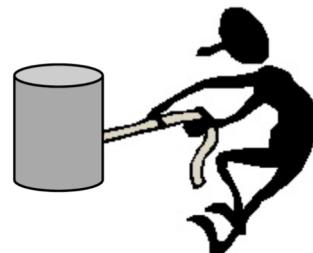
Moving to a different model



© J&G Services Ltd, 2017

Moving to a different model

- Conventional program ***pulls*** data from its sources
- Active polling or blocking can cause issues
- Event based programming is a solution
- Reactive Programming refines the approach further



© J&G Services Ltd, 2017

Reactive?

- A developing "movement" in software architecture
 - programming and architecture

The Reactive Manifesto

<http://reactivemanifesto.org>

Published on September 16 2014. (v2.0)

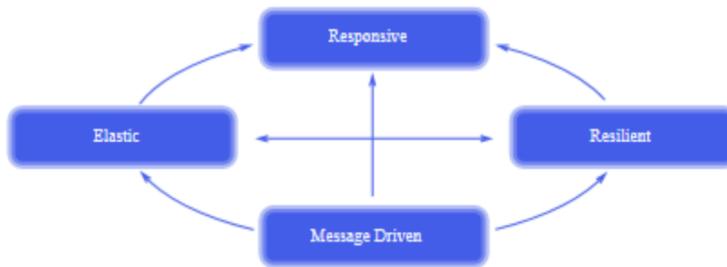
Organisations working in disparate domains are independently discovering patterns for building software that look the same. These systems are more robust, more resilient, more flexible and better positioned to meet modern demands.

These changes are happening because application requirements have changed dramatically in recent years. Only a few years ago a large application had tens of servers, seconds of response time, hours of offline maintenance and gigabytes of data. Today applications are deployed on everything from mobile devices to cloud-based clusters running thousands of multi-core processors. Users expect millisecond response times and 100% uptime. Data is measured in Petabytes. Today's demands are simply not met by yesterday's software architectures.

© J&G Services Ltd, 2017

Reactive?

- Reactive systems are
 - Responsive
 - Resilient
 - Elastic
 - Message driven (event driven)



© J&G Services Ltd, 2017

Functional?

- Reactive ideas often combined with functional programming
- Emphasis on immutable state
 - reduces the need for data synchronisation
- Declarative style abstracts away from implementation specifics
 - allows concurrency to be deployed where possible, without modifying application code

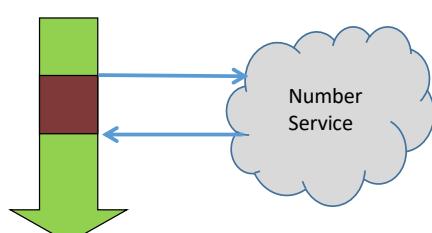


© J&G Services Ltd, 2017

Asynchronous Programming

- A different approach
 - "do something" while calling thread continues
 - hand back a result when finished
- Higher level of abstraction

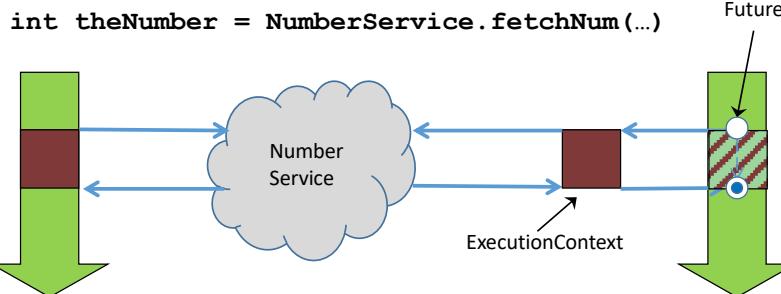
```
int theNumber = NumberService.fetchNum(...)
```



© J&G Services Ltd, 2017

Asynchronous Programming

- A different approach
 - "do something" while calling thread continues
 - hand back a result when finished
- Higher level of abstraction



© J&G Services Ltd, 2017

Fetching a Value

- Often a task is required to return a value
 - asynchronous calculation

```
public class NumberService {
    ...
    public static int getNumber () {
        delay();
        int theNum = random.nextInt() % 100;
        return theNum;
    }
    Callable<Integer> fetchNum = new Callable<Integer> () {
        @Override
        public Integer call() {
            return NumberService.getNumber();
        }
    };
    Callable<Integer> fetchNum = () -> NumberService.getNumber();
}
```

© J&G Services Ltd, 2017

Returning a Value

- Task submitted to `ExecutorService` using `submit()`
 - returns `Future<T>` to represent result

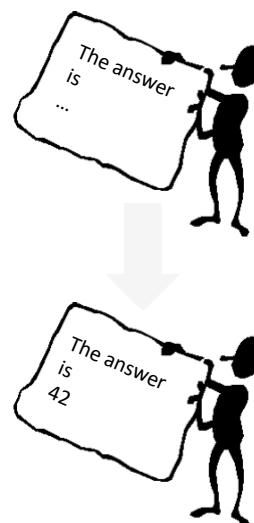
```
...
ExecutorService engine = Executors.newFixedThreadPool(2);
Future<Integer> numF = engine.submit(fetchNum);

...
Future<Integer> numF = engine.submit(
    () -> NumberService.getNumber()
);
...
```

© J&G Services Ltd, 2017

Working with `Future<T>`

- Operations:
 - fetch value, block thread if task not complete
 - check if task complete or cancelled
 - cancel task execution
- Operations throw exceptions to indicate failures
 - `ExecutionException` if task throws an exception
 - `CancellationException` if task was cancelled
 - `InterruptedException` if waiting thread is interrupted
 - `TimeoutException` if optional fetch timeout expires



© J&G Services Ltd, 2017

Working with Future<T>

```
ExecutorService engine = Executors.newFixedThreadPool(2);
Future<Integer> numF = engine.submit(
    () -> NumberService.getNumber() );
...
int theNum = 0;
try {
    while ( !numF.isDone() ) {
        System.out.print(".");
        Thread.sleep(200);
    }
    theNum = numF.get();
} catch ( InterruptedException ieIgnore ) {
} catch ( ExecutionException exIgnore ) {
} catch ( CancellationException ccIgnore ) {
}
System.out.println("Returned number: " + theNum);
```

Main thread id: 1
.....Returned number: -95

© J&G Services Ltd, 2017

Limitations of Java Futures

- Not simple to trigger behavior once future completes
- Difficulty in combining results from futures
 - it is necessary to wait for the future value
- Slow operations may never return
 - which may cause the application to hang
- Futures can actually represent bottlenecks
 - where it is necessary to wait on a future operation
 - for example while waiting for a group of futures to complete

© J&G Services Ltd, 2017

CompletableFuture

- Promotes asynchronous, event-driven model
 - with asynchronous callback support
- Support for lambda expressions
 - for functional style
- Settable features
 - allowing any thread to set the value of the CompletableFuture
- Ability to chain operations
 - allowing several asynchronous operations to be combined
 - see CompletionStage interface

© J&G Services Ltd, 2017

CompletableFuture<T>

- Concrete implementation of Future<T>
- Created using static factory methods
 - supplyAsync(Supplier<U> supplier)
 - supplyAsync(Supplier<U> supplier, Executor executor)
 - runAsync(Runnable runnable)
 - runAsync(Runnable runnable, Executor executor)
- Supplier<T> interface
 - represents a supplier of results
 - functional interface
 - defines a single method T get()

```
CompletableFuture<String> future =
    CompletableFuture.supplyAsync(() -> {
        //...long running...
        return "Hello World";
    }, executor);
```

© J&G Services Ltd, 2017

Using a CompletableFuture

- Create a Supplier and check for completion

```
ExecutorService executor = Executors.newFixedThreadPool(2);

System.out.println("Setting up Future");

CompletableFuture<String> future =
    CompletableFuture.supplyAsync(new Supplier<String>() {
        public String get() {
            try {
                Thread.sleep(500);
            } catch (InterruptedException e){
                e.printStackTrace();
            }
            return "Hello Completable World!";
        }
    }, executor);
```

© J&G Services Ltd, 2017

Using a CompletableFuture

- Create a Supplier and check for completion

```
System.out.println("Get value from CompletableFuture");

try {
    while (!future.isDone()) {
        System.out.println("Waiting ...");
        Thread.sleep(250);
    }
    String msg = future.get();
    System.out.println("Obtained result: " + msg);
} catch (Exception exp) {
    exp.printStackTrace();
}
```

Setting up Future
 Get value form CompletableFuture
 Waiting ...
 Waiting ...
 Obtained result: Hello Completable World!

© J&G Services Ltd, 2017

Using a CompletableFuture

- Create a Supplier and check for completion

- can supply a lambda expression
- which can simplify the code

```
...
CompletableFuture<String> future =
    CompletableFuture.supplyAsync(() -> {
        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return "Hello Completable World!";
    }, executor);
```

© J&G Services Ltd, 2017

Acting on Completion

- Can register a callback

- to be invoked once future is completed
- thenRun/thenRunAsync executes a Runnable
- thenAccept/thenAcceptAsync invokes a Consumer passing in a value
- typically the final stage in processing

```
System.out.println("Setting up Future");
CompletableFuture<String> future =
    CompletableFuture.supplyAsync(() -> {
        try {
            Thread.sleep(50);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return "Hello Completable World!";
    }, executor);
future.thenAccept(System.out::println);
```

Setting up Future
Hello Completable World!

© J&G Services Ltd, 2017

Combining Operations

- Can combine operations together

- chain of operations to be invoked once future completes

- thenApply/thenApplyAsync execute a Function

- supports both synchronous and asynchronous invocations

42

```
CompletableFuture<Integer> f1 =
    CompletableFuture.supplyAsync(() -> {
        try {
            Thread.sleep(50);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return 6;
    }, executor);
// Combine operations together
CompletableFuture<Integer> f2 = f1.thenApply(v -> v * 7);
CompletableFuture<String> f3 =
    f2.thenApply(v -> Integer.toString(v));
// Finally print result
f3.thenAccept(System.out::println);
```

© J&G Services Ltd, 2017

Composing CompletableFutures

- **CompletableFutures** are composable

- allows chaining of operations on the Future
- thenCompose / thenComposeAsync

```
public static Future<Integer> compose() {
    CompletableFuture<Integer> cf1 =
        NumberService.getIntegerAsync();

    CompletableFuture<Integer> cf2 = cf1.thenCompose(
        v -> CompletableFuture.supplyAsync(
            () -> NumberService.squareIt(v)));

    CompletableFuture<Integer> cf3 = cf2.thenCompose(
        v -> CompletableFuture.supplyAsync(
            () -> NumberService.cubeIt(v)));

    return cf3;
}
```

© J&G Services Ltd, 2017

Combining CompletableFuture together

- Can combine two independent futures

- thenCombine / thenCombineAsync
- results from two CompletableFuture combined

```
ExecutorService executor =
    Executors.newFixedThreadPool(3);

CompletableFuture<String> f1 =
    CompletableFuture.supplyAsync(() -> {
        return "Hello";
    } , executor);

CompletableFuture<String> f2 =
    CompletableFuture.supplyAsync(() -> {
        return "World";
    } , executor);

CompletableFuture<String> f3 =
    f1.thenCombine(f2, (x, y) -> x + " " + y);

f3.thenAccept(System.out::println);
```

Hello World

© J&G Services Ltd, 2017

Waiting for CompletableFuture to complete

- Can wait for two CompletableFuture to complete
 - thenAcceptBoth / runAfterBoth (and async versions)
- Can wait for first to finish
 - acceptEither / runAfterEither (and async versions)
 - applyToEither / applyToEitherAsync returns a future
 - anyOf / anyOfAsync handles any number of futures
- Combining multiple CompletableFuture futures
 - can wait for arbitrary number of futures
 - allOf / allOfAsync

© J&G Services Ltd, 2017

Error handling

- Can also handle exceptions asynchronously
 - exceptionally takes a function
 - invoked asynchronously when future throws an error

```
System.out.println("Setting up Future");
CompletableFuture<String> future =
    CompletableFuture.supplyAsync(..., executor);

CompletableFuture<String> safe =
    future.exceptionally(
        ex -> "We have a problem: " + ex.getMessage());

safe.thenAccept(System.out::println);
```

© J&G Services Ltd, 2017

Error handling

- Can handle exceptions when complete
 - whenComplete methods can all take an exception
 - can handle exception from anywhere in chain

```
CompletableFuture<String> f1 =
    CompletableFuture.supplyAsync(() -> {
        return "Hello";
    } , executor);
CompletableFuture<String> f2 = f1.thenApply(
    v -> {throw new RuntimeException("Noooo - " + v);});

f2.whenComplete((v, err) -> {
    if (v != null) {
        System.out.println(v);
    } else {
        System.err.println("Handling exception: " +
            err.getMessage());
    }
});
```

Handling exception: java.lang.RuntimeException: Noooo - Hello

© J&G Services Ltd, 2017

Explicit Completion

- CompletableFuture supports explicit completion
 - Success or failure
 - Allows for greater control over how tasks are executed

```
...
public static Future<Integer> fetchIntCF () {
    CompletableFuture<Integer> numF = new CompletableFuture<>();

    new Thread( () -> {
        int num = NumberService.getNumber();
        numF.complete(num);
    } ).start();
    return numF;
}

public static Future<Integer> fetchIntAsync () {
    return CompletableFuture.supplyAsync( () -> fetchInt() );
}
...

```

© J&G Services Ltd, 2017

Explicit Completion

- Task can complete but not successfully
 - boolean completeExceptionally(Throwable ex)
 - if not already completed, causes invocation of get and related methods to throw the given exception
 - return result indicates if future is now in completed state

```
boolean res = future.completeExceptionally(
    new RuntimeException("Opps"));
```

© J&G Services Ltd, 2017