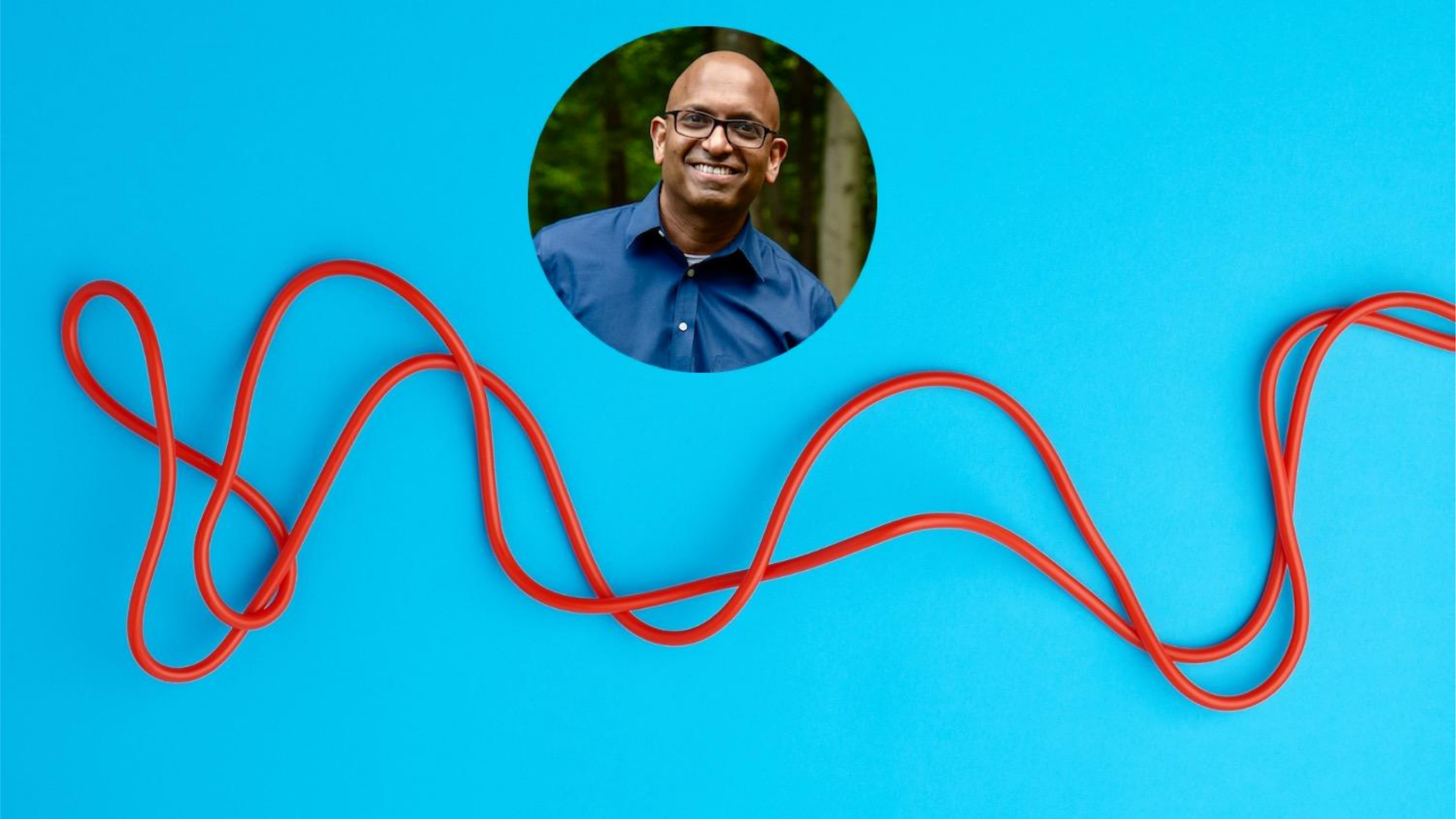
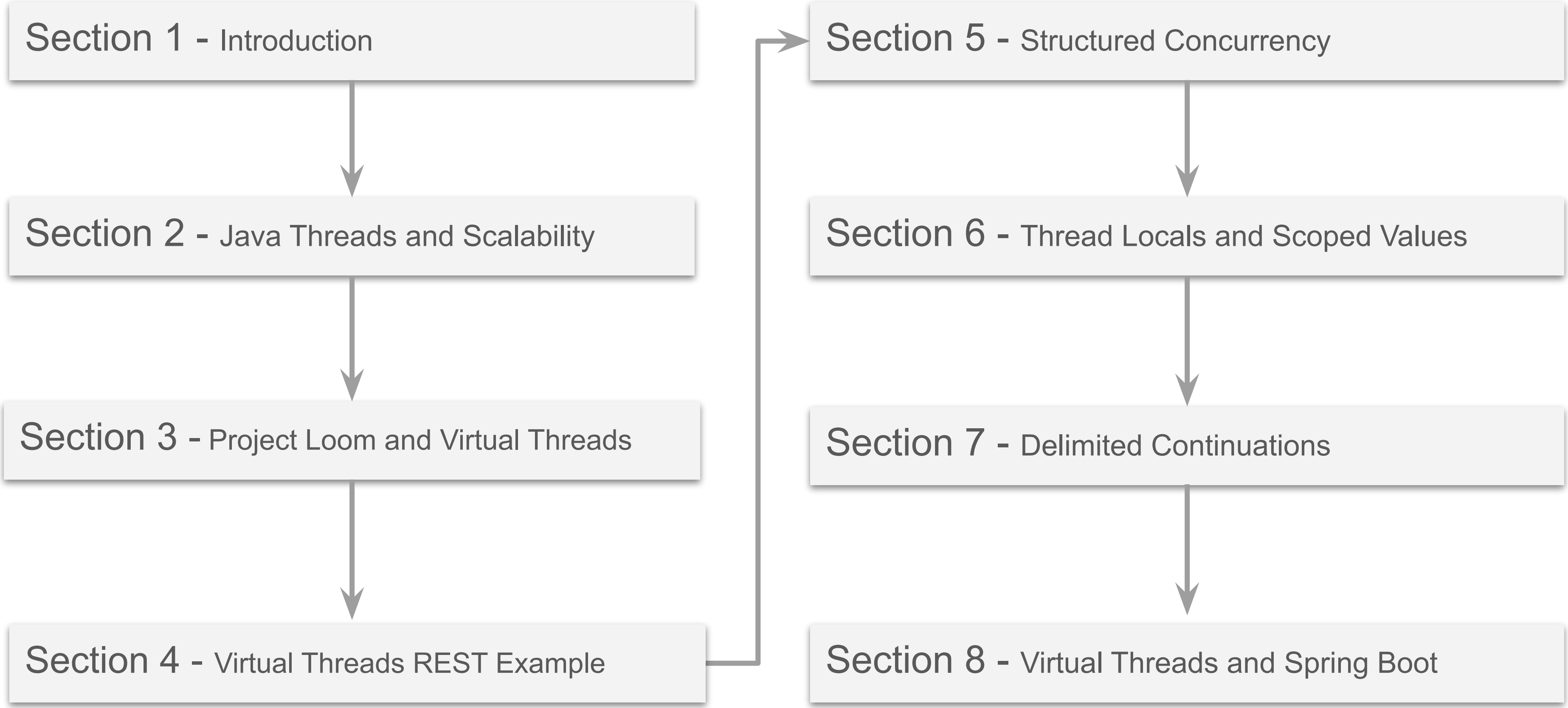


Course Content



Section 1 - Introduction

Section 2 - Java Threads and Scalability

Section 3 - Project Loom and Virtual Threads

Section 4 - Virtual Threads REST Example

Section 5 - Structured Concurrency

Section 6 - Thread Locals and Scoped Values

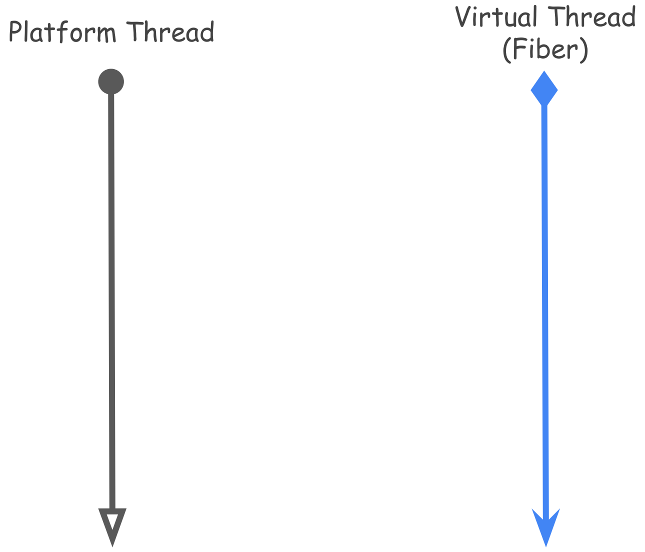
Section 7 - Delimited Continuations

Section 8 - Virtual Threads and Spring Boot

Bonus - Java Futures and Completable Futures Tutorial

Web Application (Process)

Deploy the **Web Application** in a more powerful machine, VM or Container



Platform Thread

Virtual Thread (Fiber)

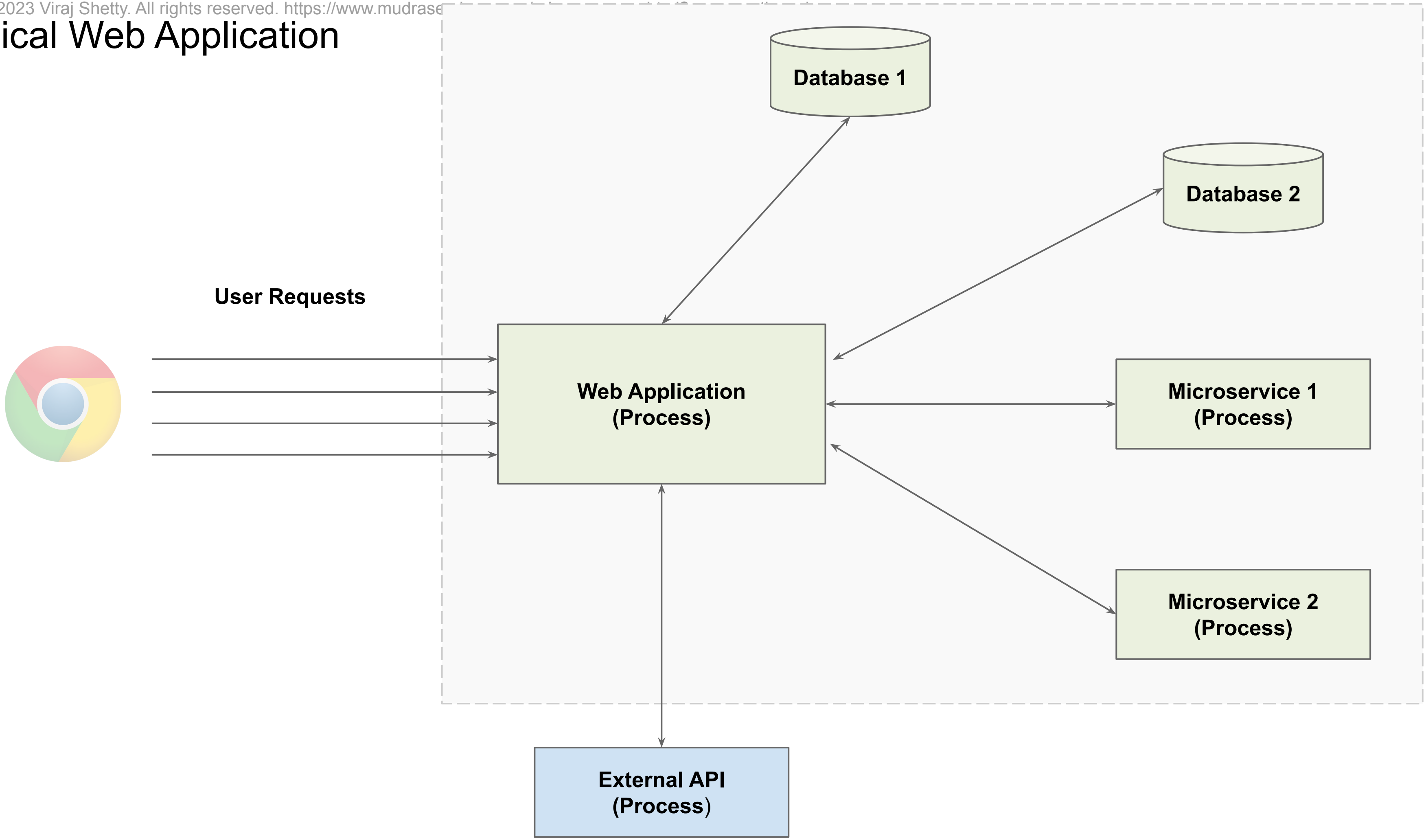
- Concurrent Users
- Scalability
- Loom Early Access Builds

Java Threads and Scalability

- ❖ Task Types
- ❖ Concurrency and Parallelism
- ❖ Non Blocking IO
- ❖ Introduce Project Loom

Task Types

Typical Web Application



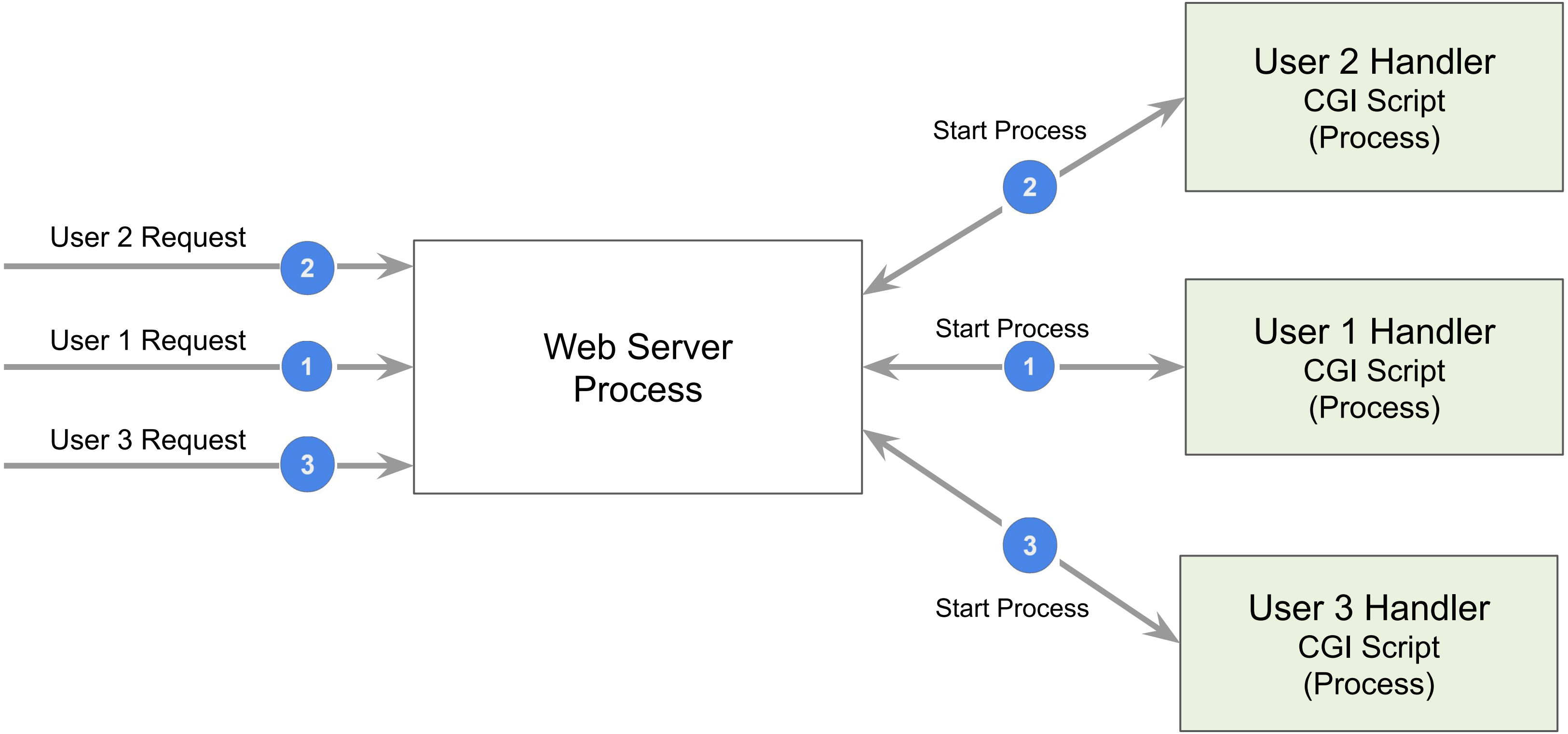
Task Types

```
// Pseudo code for handling User Request

// Fetch some data from DB
data1 = FetchDataFromDB(dbUrl)

// Fetch some data from a Microservice 1
data2 = FetchDataFromService1(url1)

// Fetch some data from a Microservice 2
data3 = FetchDataFromService2(url2)

// Process all data
combinedData = ProcessAndCombine(data1, data2, data3)

// send data to user
SendData(combinedData)
```

- Task Types
  - IO Bound
  - CPU Bound

Handling User Requests

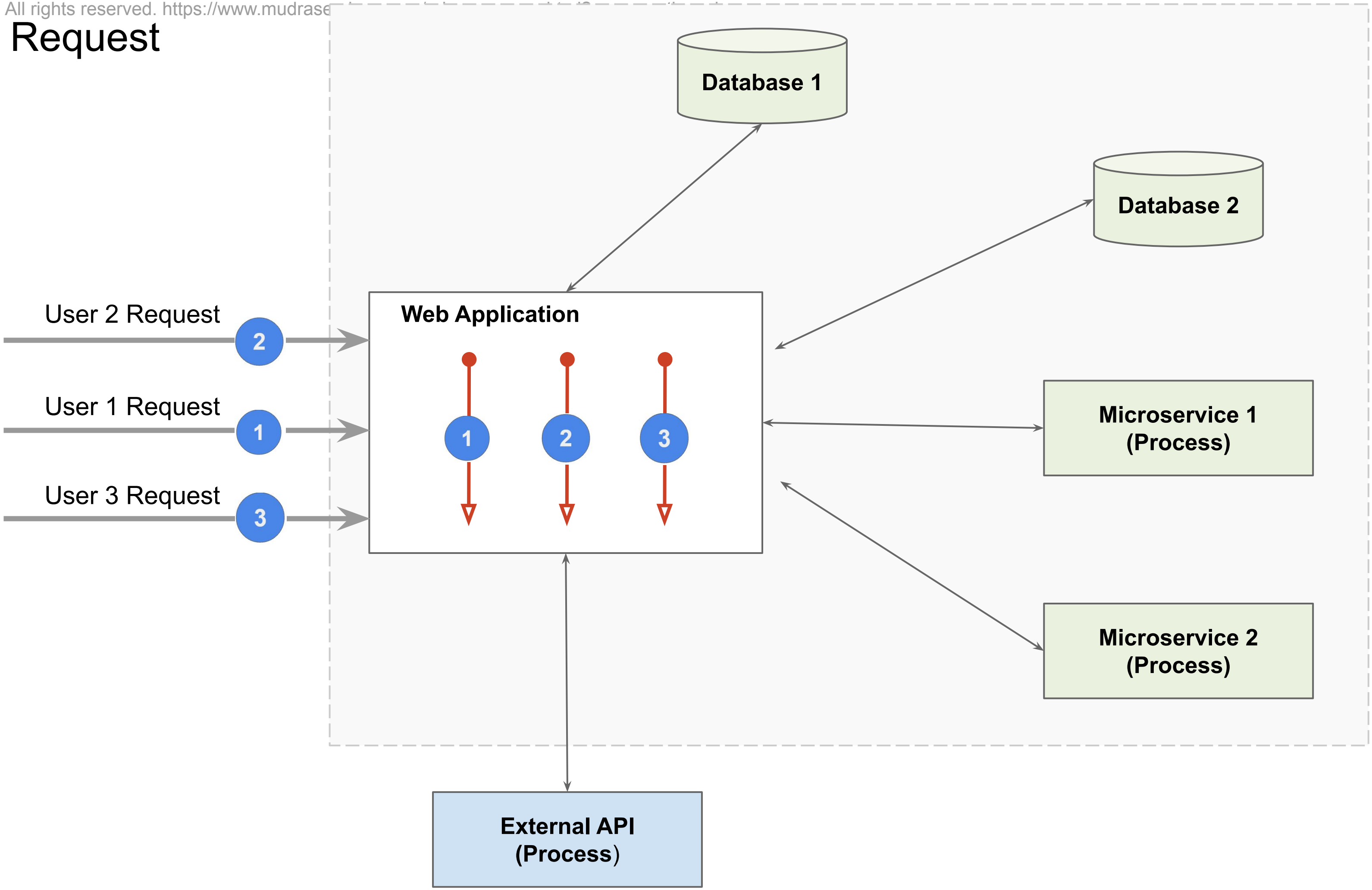
Process Per Request (CGI)



Process Per User Request

- Process is heavyweight
- Limited number of Processes per machine
  - Scalability issues
  - Cannot support large number of users
- Expensive Process startup and termination time
- Difficult to share data or communicate between Processes
- FastCGI
  - Pooling of Processes
  - CGI processes are started upfront for performance

Thread Per Request



Thread Per User Request

- Thread is lightweight
  - But has its own stack
- Can handle larger number of concurrent users
- Can share data or communicate between threads
- Improved Performance
  - No extra process to deal with
- Easy to understand
- Easy to debug

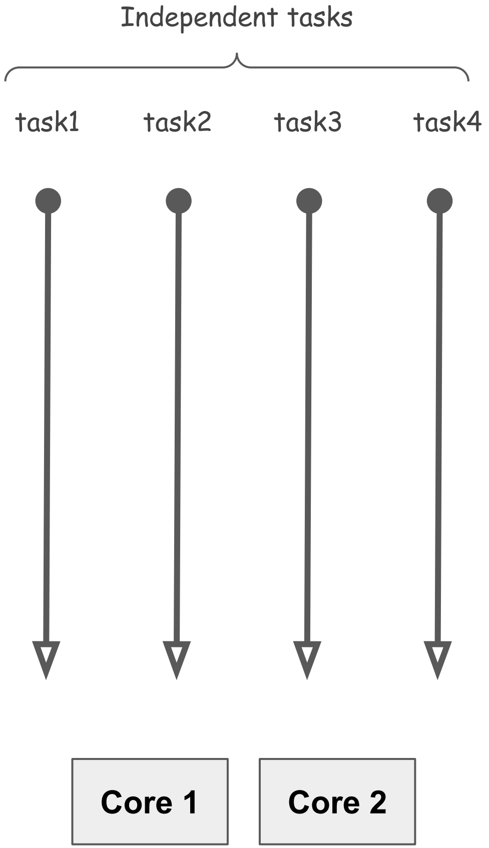
Concurrency Versus Parallelism

Concurrency

- Multiple independent tasks are making progress but may not execute at the ***same*** time

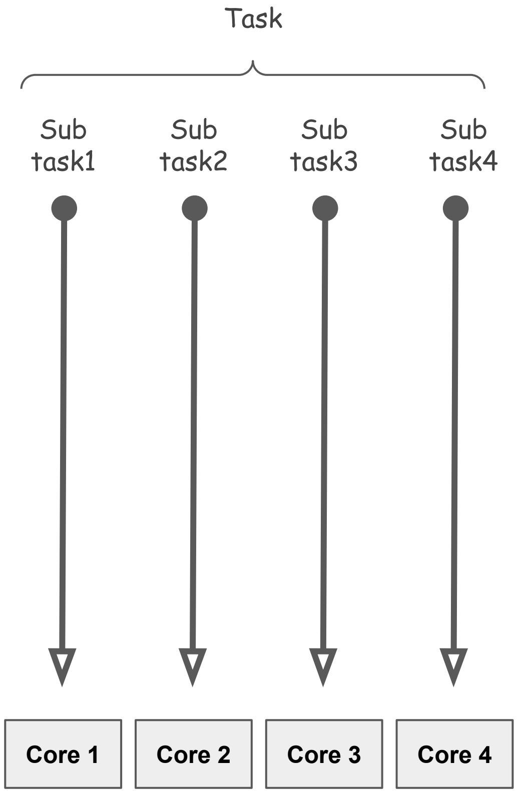
- Appearance of Parallelism

- CPU time slicing



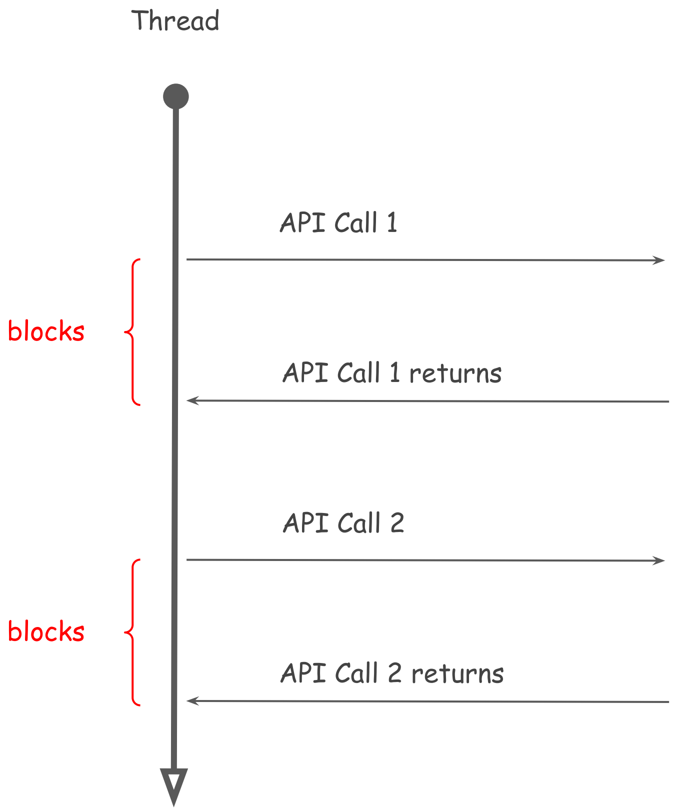
Parallelism

- Multiple dependent sub tasks are executing at the ***same*** time
- Multiple cores needed
- No parallelism in single core



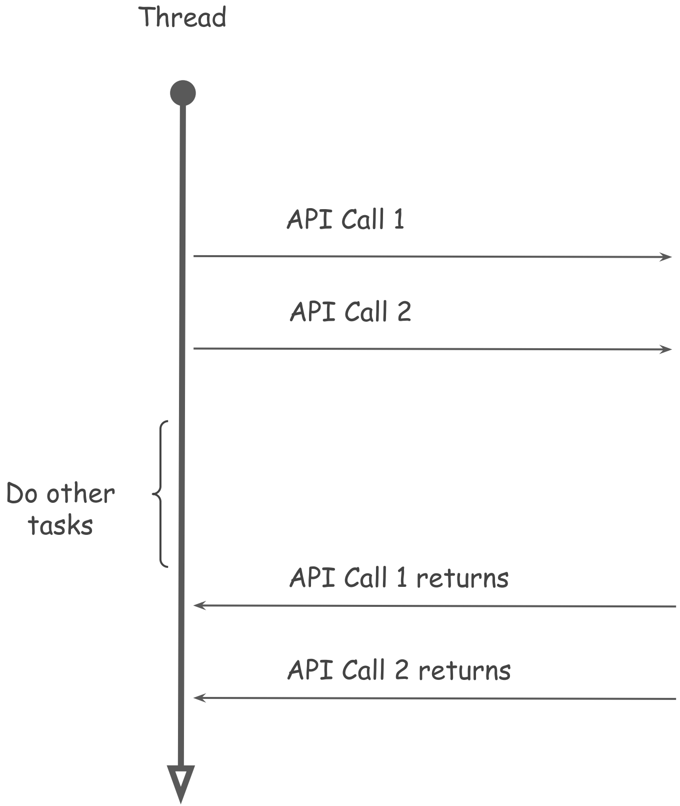
Synchronous Call

- Sequential execution of code
- Easy to understand
- Easy to debug



Asynchronous Call

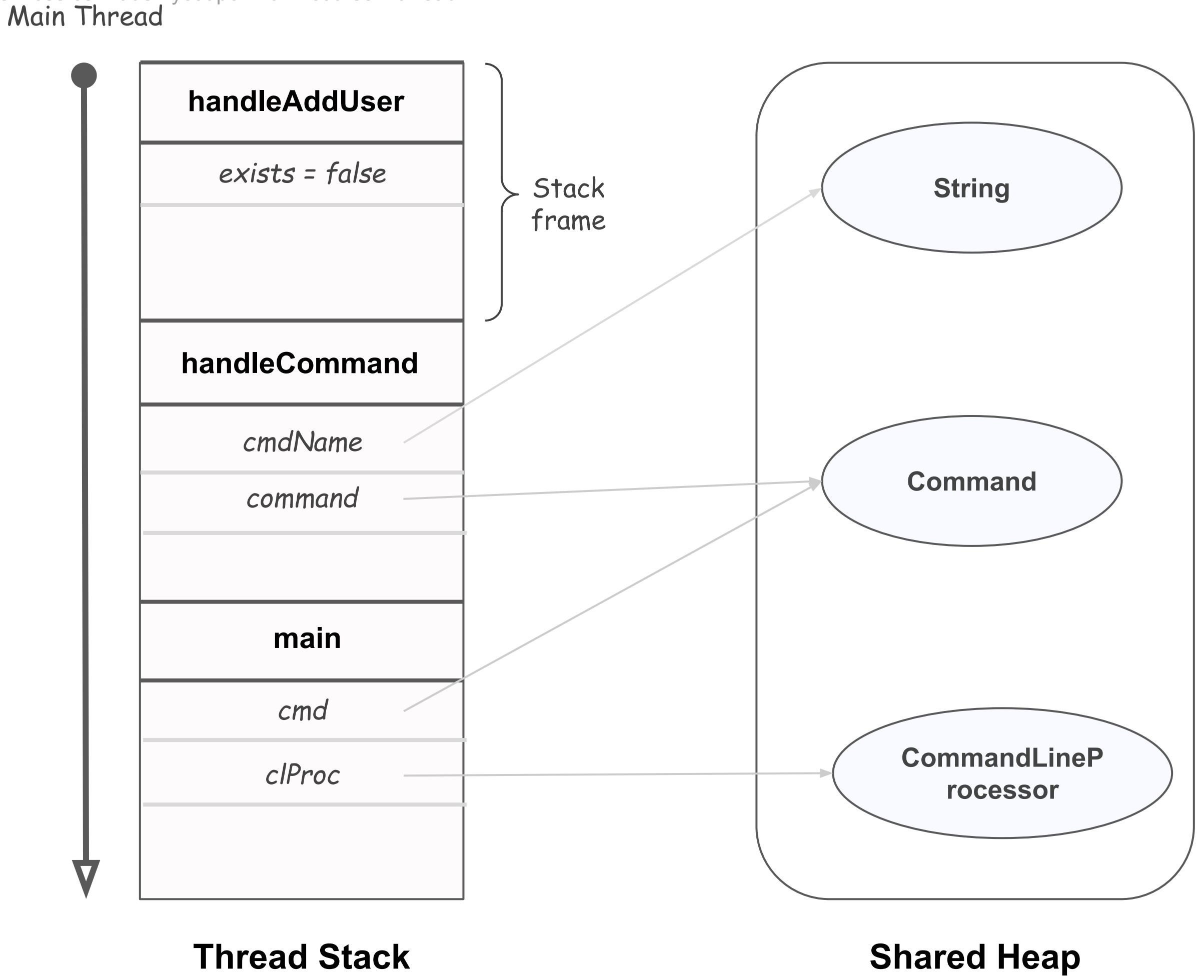
- Does not wait for call to complete
- Callbacks, futures
- More complex to understand
- In Java, user Threads



Java Threads

Java Threads

```
package com.mudra.loom;
public class CommandLineProcessor {

    private static void handleAddUser() {
        boolean exists = false;

        // code to handle user creation
    }

    public void handleCommand(Command command) {

        String cmdName = command.name();
        if ("adduser".equalsIgnoreCase(cmdName)) {
            handleAddUser();
        }

        // rest of the code
    }

    public static void main(String[] args) {

        Command cmd = extractCommand(args);

        var clProc = new CommandLineProcessor();
        clProc.handleCommand(cmd);

    }

    private static Command extractCommand(String[] args) {
        // return the command object
    }

}
```



Java Threads

- Fundamental to the Java Platform
  - Debugging
  - Exceptions
- Every Java Thread is a wrapper around an OS Thread
  - OS Thread is an expensive resource
- Thread Pools
- Stack Memory Size can be set (-Xss)
- Heap memory can be set
  - -Xmx, -Xms

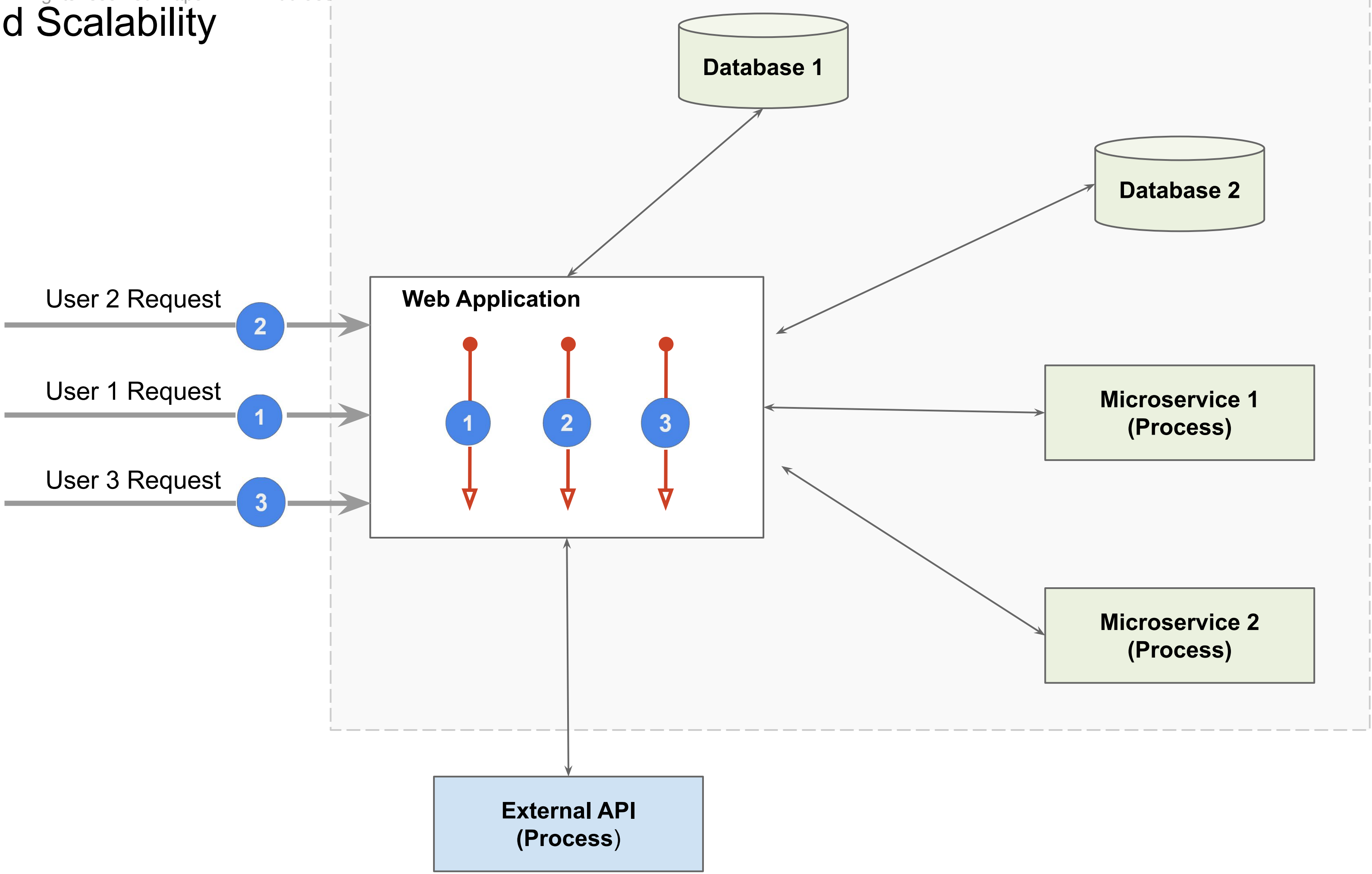
```
java -Xss512k -Xmx1G -Xms256k com.mudra.CommandLineProcessor <command> <arg1> <arg2>
```

Demonstration

- Mac Mini
  - 2 Cores
  - 8G RAM
  - Intel Core i5 2.6GHz
- Eclipse IDE 2021-12
- Application
  - JRE 17
  - Stack Size - 1M
  - Max Heap Size - 1G

Threads and Scalability

Threads and Scalability



Threads and Scalability

- Default stack size 1M
  - As number of users increase, memory usage increases

- There is a max limit to the max threads
  - Depends on VM or Machine Memory
  - Much more socket connections can be supported
  - This prevents optimum scalability

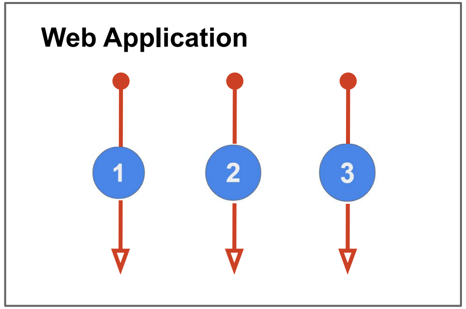
- IO bound tasks
  - Paralyzes the OS thread for a longer time than necessary

Scalability Solutions

Vertical Scaling

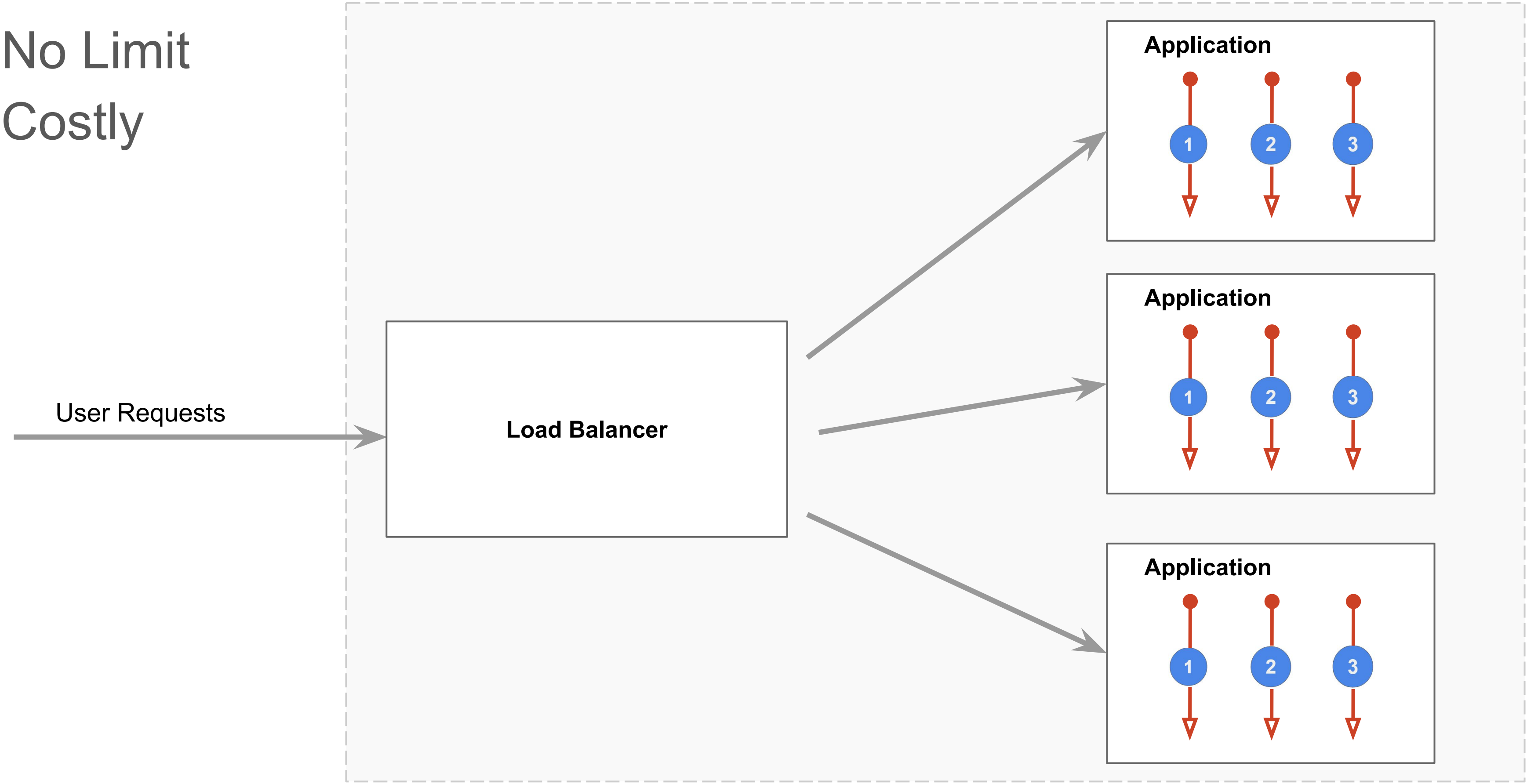
- Increase Resources
- CPU, Memory, Disk Space etc
- Limit to scaling
- Increases cost
- Cloud Environment

Deploy the Web Application in a more powerful machine, VM or Container



Horizontal Scaling (Increase number of Application nodes)

- No Limit
- Costly



Scalability Solution

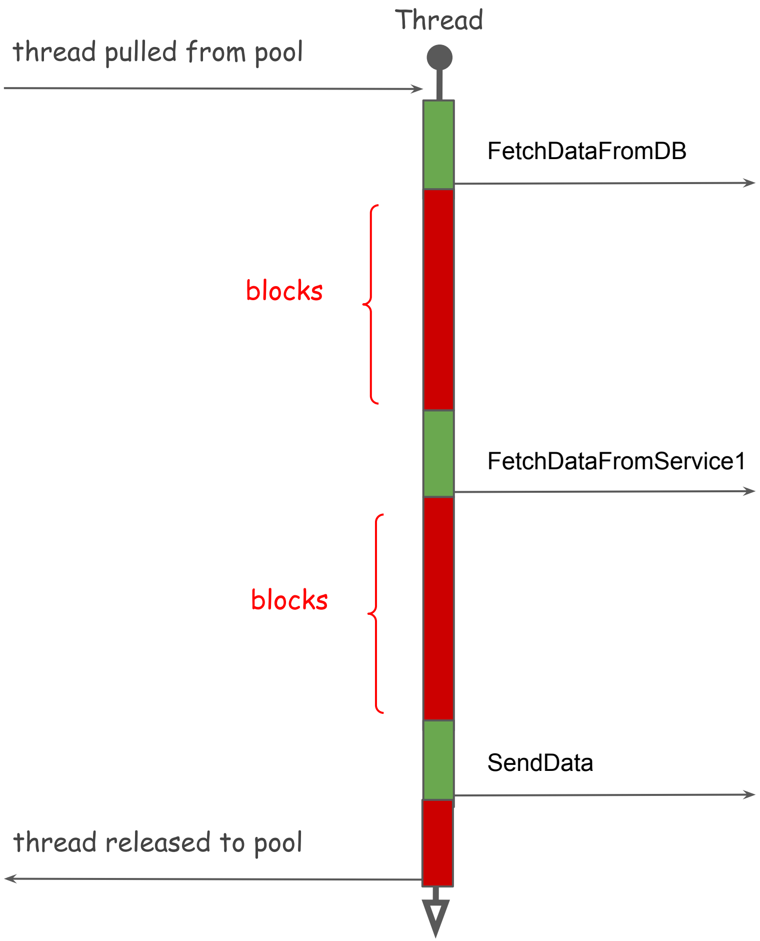
{ Optimized Scalable Application } + { Vertical Scaling } + { Horizontal Scaling }

Non Blocking IO

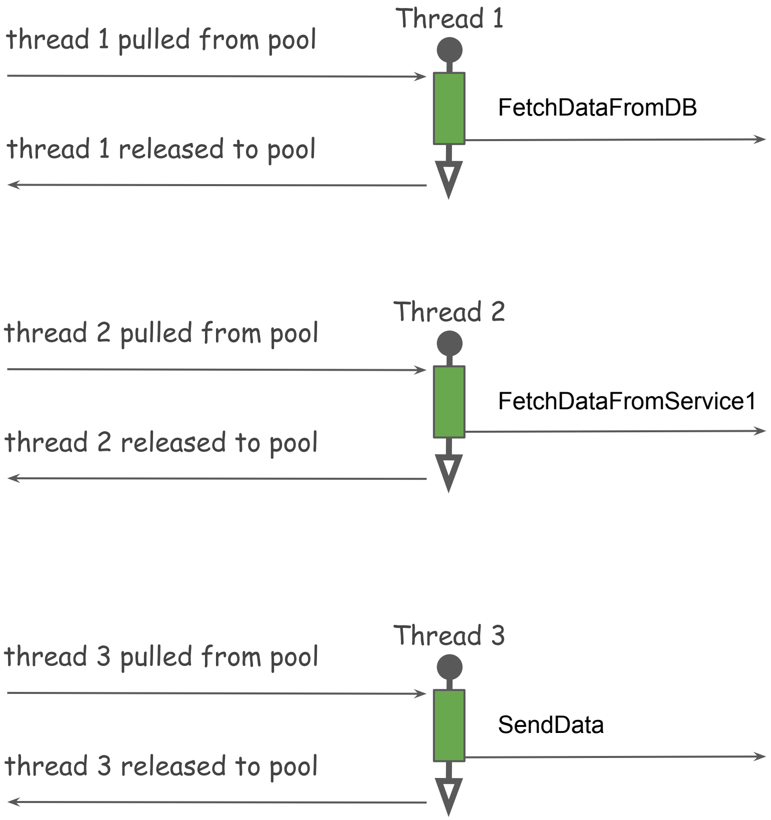
Pseudo Code for Blocking IO

```
// Pseudo code for handling User Request

// Fetch some data from DB
data1 = FetchDataFromDB(dbUrl)

// Fetch some data from a Microservice 1
data2 = FetchDataFromService1(url1)

// Process all data
combinedData = ProcessAndCombine(data1, data2)

// send data to user
SendData(combinedData)
```

Blocking IO



Non Blocking IO


Thread 1
thread 1 pulled from pool
FetchDataFromDB
thread 1 released to pool
Thread 2
thread 2 pulled from pool
FetchDataFromService1
thread 2 released to pool
Thread 3
thread 3 pulled from pool
SendData
thread 3 released to pool

Pseudo Code for Non Blocking IO (Callbacks)

```
    // Non Blocking : Fetch some data from DB
1   FetchDataFromDB(dbUrl, DBCallback(data1) {

        // Non Blocking : Fetch some data from a Microservice 1
        FetchDataFromService1(url1, RestCallback(data2) {

            // Process all data and send
            combinedData = ProcessAndCombine(data1, data2)
            SendData(combinedData)
        }
    }

    // Control reaches here before data is returned
2   // Thread is released
```

Non Blocking IO in Java

- Non Blocking IO
  - Java NIO (New IO) { July 2011 with JDK 7 }
    - Non Blocking File and Socket handling
  - Java CompletableFutures { March 2014 with JDK 8 }
  - Servlet 3.0 and 3.1 includes Non Blocking Servlet

- Reactive Programming
  - RxJava, Project Reactor
  - Spring WebFlux

- Disadvantages
  - High Complexity for Developers
  - Easy to make mistakes
  - End to End Non Blocking

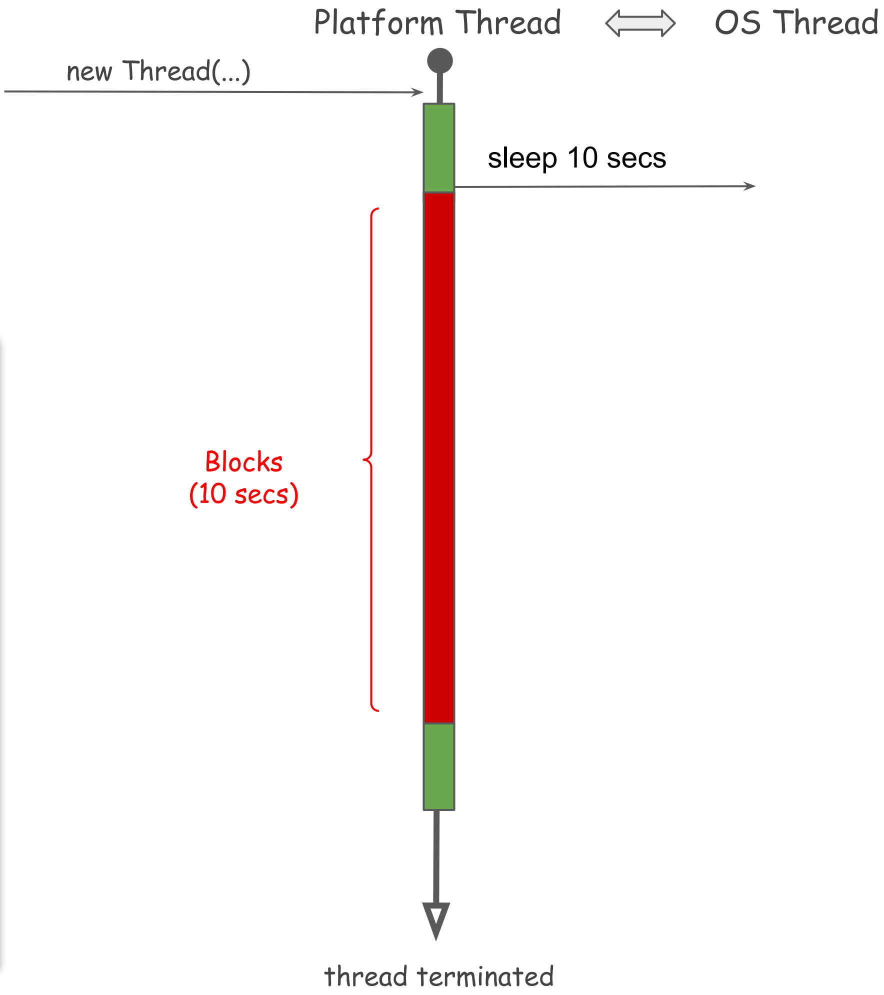
There is another solution ...

Virtual Threads

Platform Threads

```
new Thread(() -> handleUserRequest()).start();
```

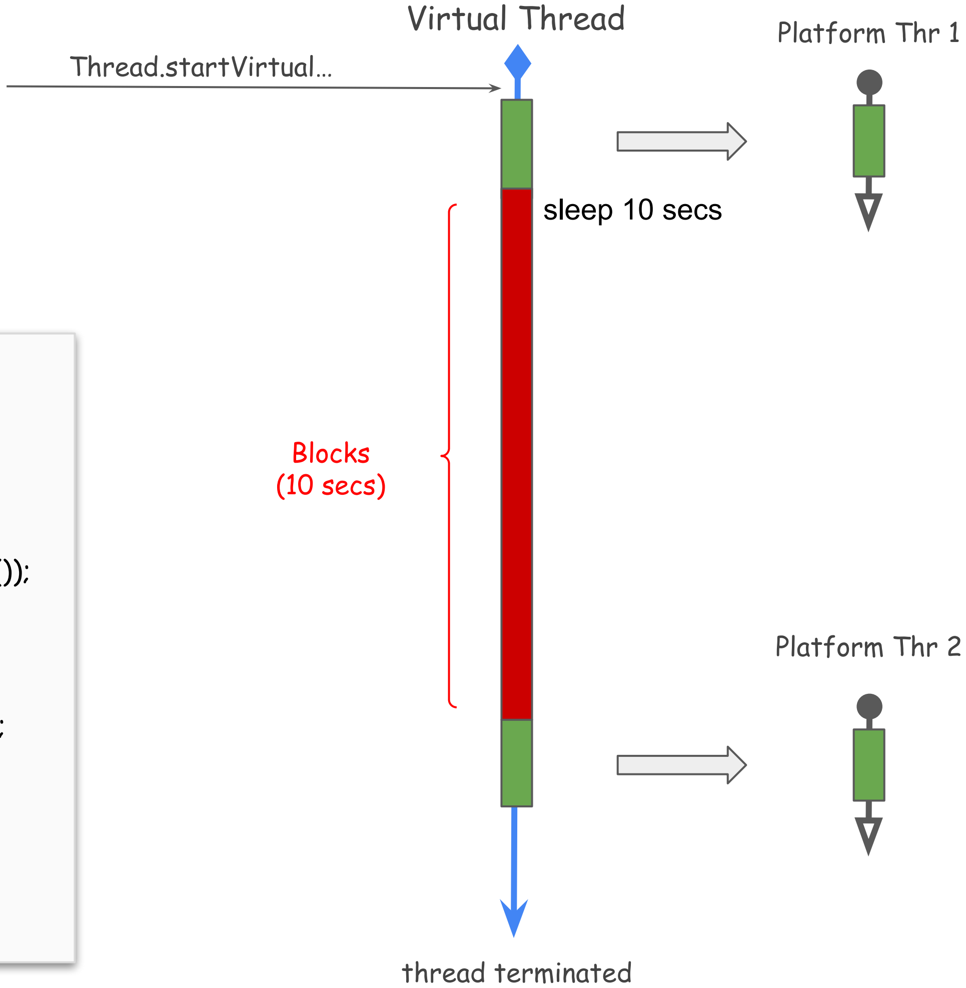
```
private static void handleUserRequest() {

    System.out.println("Starting thread " + Thread.currentThread());

    Thread.sleep(10000);

    System.out.println("Ending thread " + Thread.currentThread());

}
```

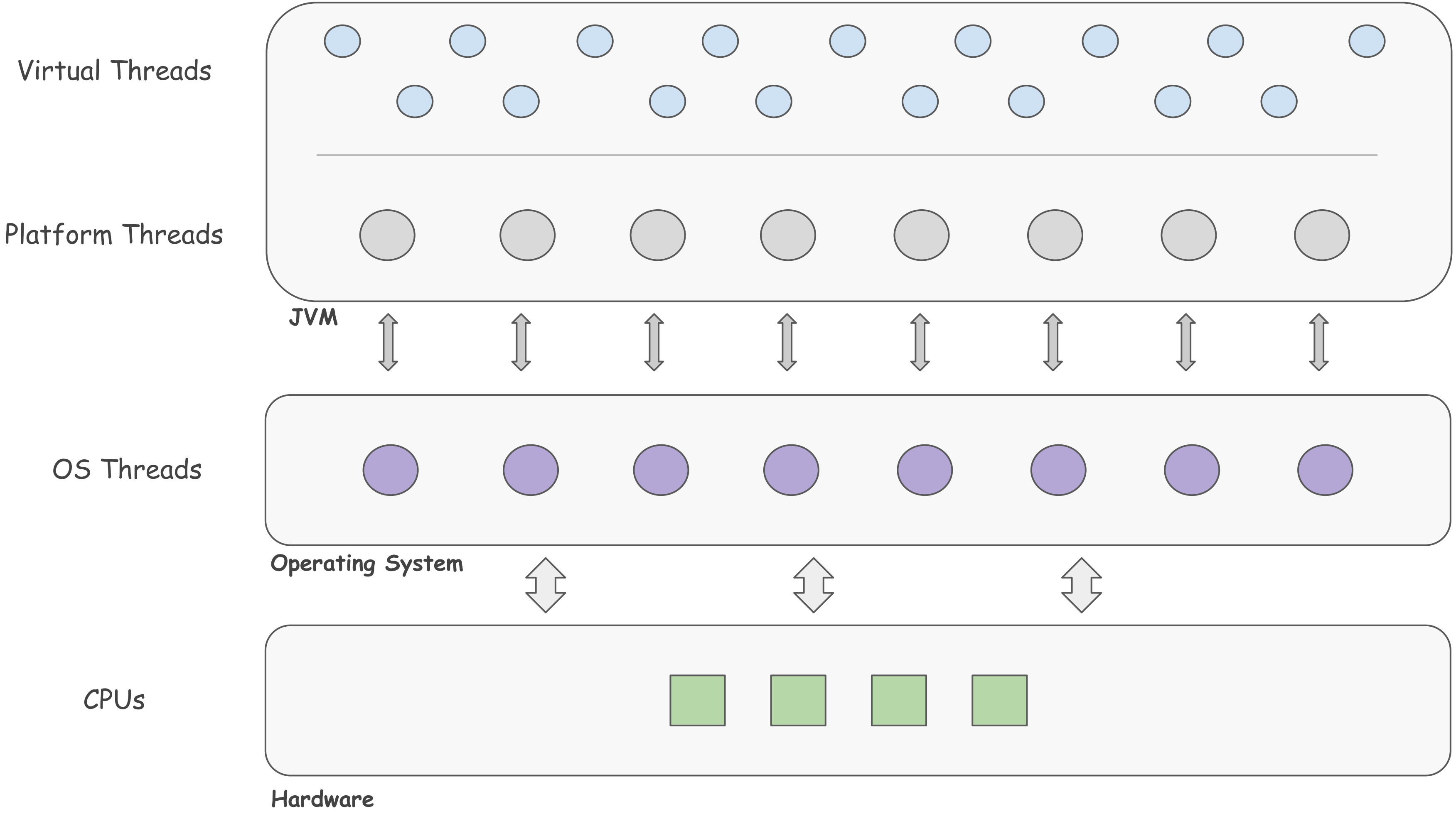


Virtual Threads

```
Thread.startVirtualThread(() -> handleUserRequest());
```

```
private static void handleUserRequest() {

    System.out.println("Starting thread " + Thread.currentThread());

    Thread.sleep(10000);

    System.out.println("Ending thread " + Thread.currentThread());

}
```




Virtual Threads
Platform Threads
JVM
OS Threads
Operating System
CPUs
Hardware

Creating Virtual Threads

Using a static Thread method

```
// Start a new Virtual thread. No name is associated with thread
Thread vThread1 = Thread.startVirtualThread(() -> handleUserRequest());

// Make sure the thread terminates
vThread1.join();

// Control reaches here once the virtual thread completes
```

Using the Virtual Thread Builder

```
// Create a Virtual Builder object with name and initial index
OfVirtual vBuilder = Thread.ofVirtual().name("userthread", 0);

// Start two virtual threads using the builder
Thread vThread1 = vBuilder.start(VirtualMethodsPlay::handleUserRequest);
Thread vThread2 = vBuilder.start(VirtualMethodsPlay::handleUserRequest);

// Make sure the threads terminate
vThread1.join();
vThread2.join();

// Control reaches here once the two virtual threads complete
```

Using the Thread Factory

```
// Create a Thread factory
ThreadFactory factory = Thread.ofVirtual().name("userthread", 0).factory();

// Start two virtual threads using the factory
Thread vThread1 = factory.newThread(VirtualMethodsPlay::handleUserRequest);
vThread1.start();

Thread vThread2 = factory.newThread(VirtualMethodsPlay::handleUserRequest);
vThread2.start();

// Make sure the threads terminate
vThread1.join();
vThread2.join();

// Control reaches here once the two virtual threads complete
```

Using the Virtual Thread Executor Service

```
// Create an Virtual Thread ExecutorService
// Note the try with resource which will make sure all Virtual threads
// are terminated
try (ExecutorService srv = Executors.newVirtualThreadPerTaskExecutor()) {

    // Submit two tasks to the Executor service
    srv.submit(VirtualMethodsPlay::handleUserRequest);
    srv.submit(VirtualMethodsPlay::handleUserRequest);

}

// Control reaches here once the two virtual threads complete
```

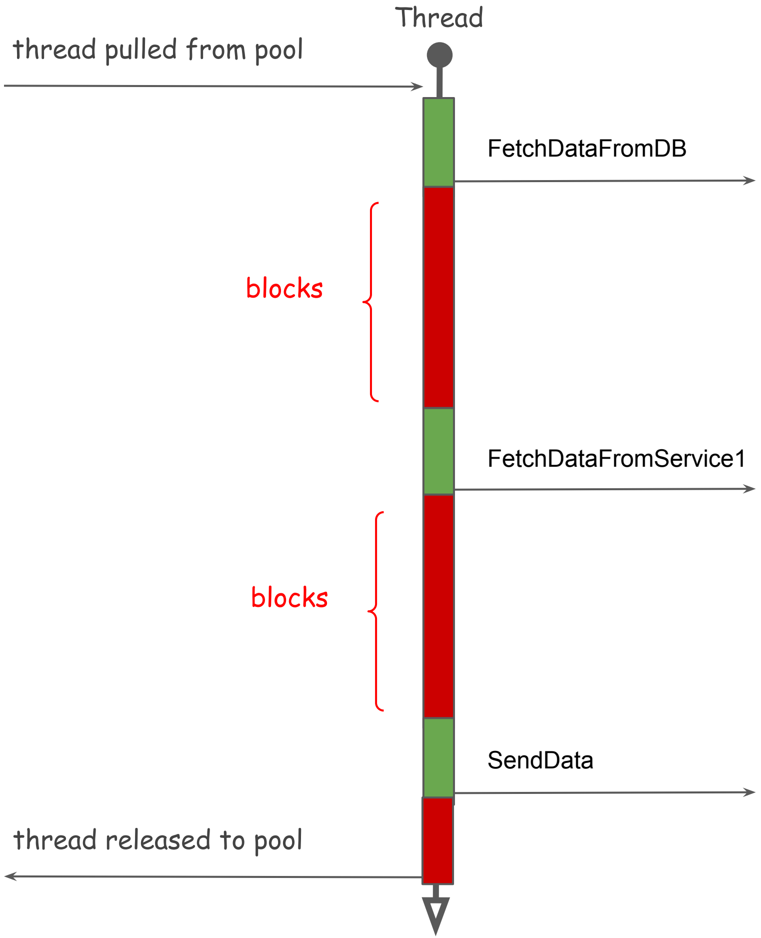
Using the Thread Executor Service

```java
// Create a Virtual Thread factory with custom name
ThreadFactory factory = Thread.ofVirtual().name("userthread", 0).factory();

// Create an ExecutorService for this factory
// Note the try with resource which will make sure all Virtual threads
// are terminated
try (ExecutorService srv = Executors.newThreadPerTaskExecutor(factory)) {

    // Submit two tasks to the Executor service
    srv.submit(VirtualMethodsPlay::handleUserRequest);
    srv.submit(VirtualMethodsPlay::handleUserRequest);
}

// Control reaches here once the two virtual threads complete
```

Virtual Threads - Advantages

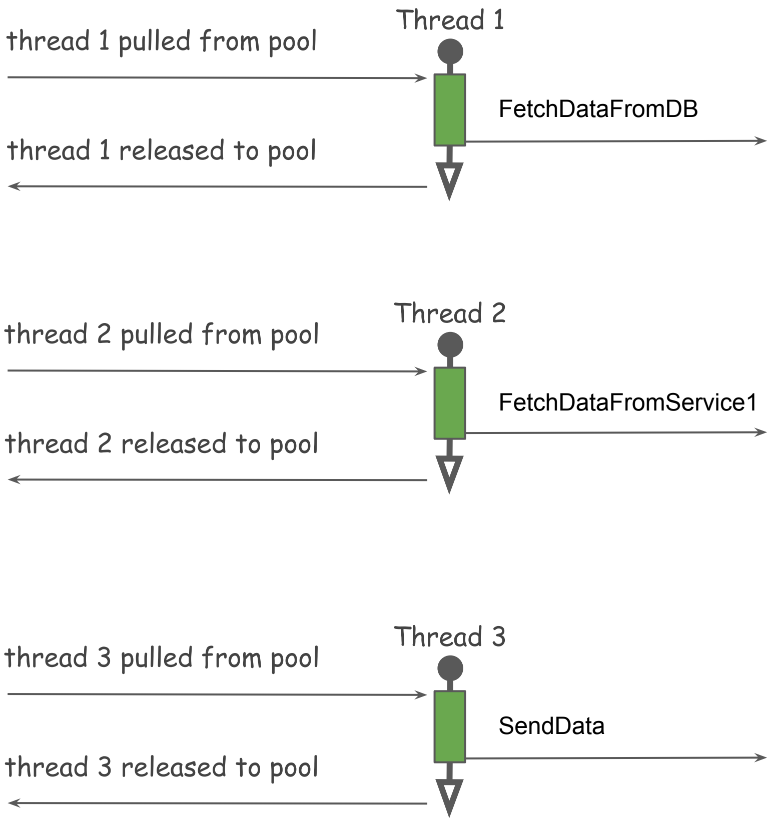
Pseudo Code for Blocking IO

```
// Pseudo code for handling User Request

// Fetch some data from DB
data1 = FetchDataFromDB(dbUrl)

// Fetch some data from a Microservice 1
data2 = FetchDataFromService1(url1)

// Process all data
combinedData = ProcessAndCombine(data1, data2)

// send data to user
SendData(combinedData)
```



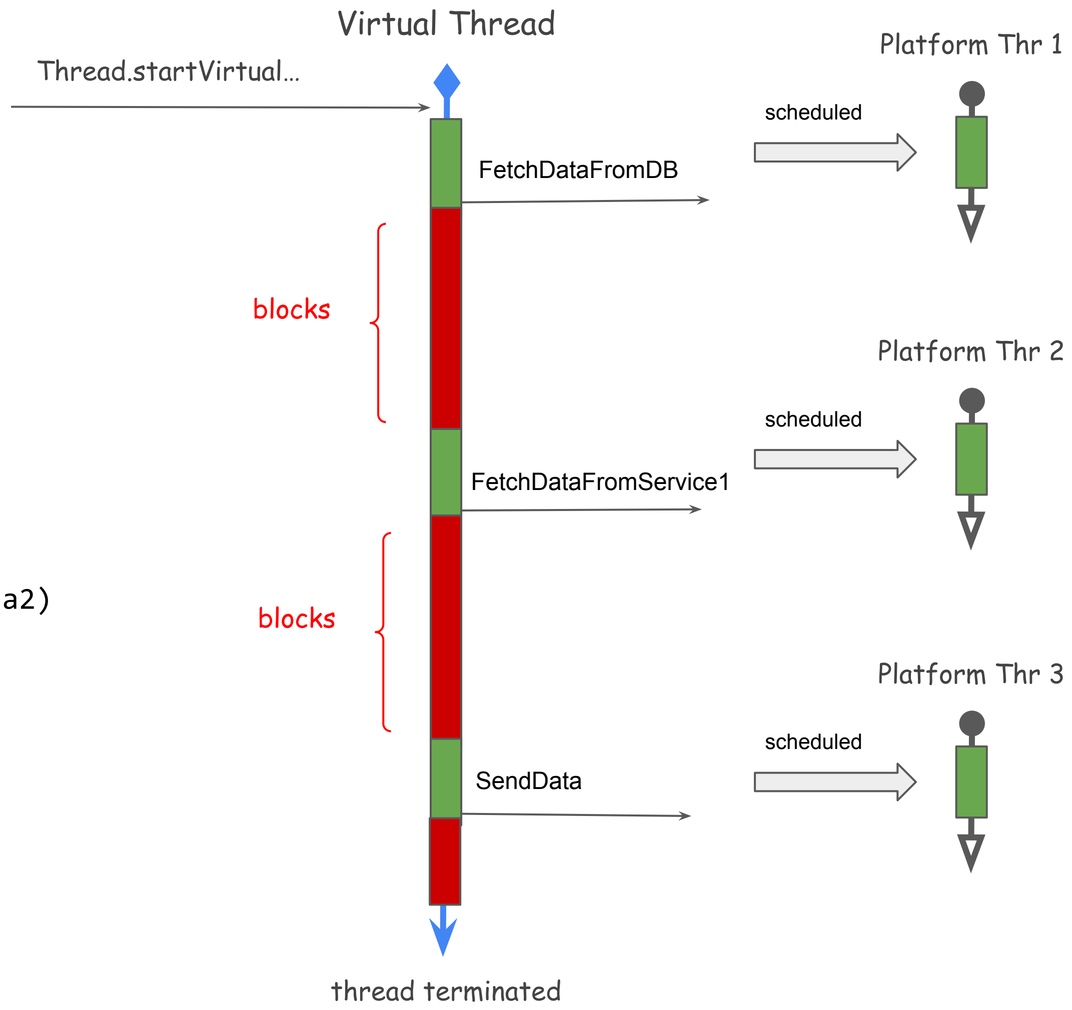
Pseudo Code for Non Blocking IO (Callbacks)

```
// Non Blocking : Fetch some data from DB
FetchDataFromDB(dbUrl, DBCallback(data1) {

    // Non Blocking : Fetch some data from a
Microservice
    FetchDataFromService1(url1, RestCallback(data2) {

        // Process all data and send
        combinedData = ProcessAndCombine(data1, data2)
        SendData(combinedData)
    }
}

// Control reaches here before data is returned
// Thread is released
```


Thread 1
thread 1 pulled from pool
FetchDataFromDB
thread 1 released to pool
Thread 2
thread 2 pulled from pool
FetchDataFromService1
thread 2 released to pool
Thread 3
thread 3 pulled from pool
SendData
thread 3 released to pool

Using Virtual Threads

```
// Pseudo code for handling User Request

// Fetch some data from DB
data1 = FetchDataFromDB(dbUrl)

// Fetch some data from a Microservice 1
data2 = FetchDataFromService1(url1)

// Process all data
combinedData = ProcessAndCombine(data1, data2)

// send data to user
SendData(combinedData)
```



Virtual Threads - Advantages

- Light Weight Thread (extends the Thread class)
  - Fast Creation time
  - Exhibits same behavior as Platform Threads
  - Scales to millions of instances

- Advantages
  - No need for Thread Pool
  - Can block on IO with no scalability issues
  - Optimal Concurrency
  - Code can still be Sequential
  - Existing code will benefit from using Virtual Thread
  - Combine with Futures and CompletableFuture

Virtual Threads - Limitations

Don’t use a Monitor

```java
public class MainJacket {

    private static void handleUserRequest() {
        System.out.println("Starting thread " + Thread.currentThread());

        synchronized (MainJacket.class) {
            try {
                Thread.sleep(Duration.ofMinutes(5));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        System.out.println("Ending thread " + Thread.currentThread());
    }

    @SuppressWarnings("preview")
    public static void main(String[] args) throws  Exception {
        Thread.startVirtualThread(MainJacket::handleUserRequest).join();
    }
}
```

Use Locks from java.util.concurrent

```
private static Lock ioLock = new ReentrantLock();

private static void handleUserRequest() {
        System.out.println("Starting thread " + Thread.currentThread());

                try {
                        ioLock.lock();

                        Thread.sleep(Duration.ofMinutes(1));
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
                finally {
                        ioLock.unlock();
                }

        System.out.println("Ending thread " + Thread.currentThread());
}
```

Other Limitations

- Blocking with native frames on Stack (JNI)
  - This is rare
- Control memory per stack
  - Reduce Thread Locals
  - No deep recursions
- Java Tools have not been updated
  - Debuggers, JConsole, VisualVM

Structured Concurrency

- Runtime behavior mirrors the structure of code, arranged in blocks

```
public String call() throws Exception {

    // Sequential coding ..
    String result1 = dbCall1();
    String result2 = dbCall2();

    // complicated parallel threads code in limited to the block below
    try (ExecutorService service = Executors.newVirtualThreadPerTaskExecutor()) {
        String result = CompletableFuture
                            .supplyAsync(this::restCall1, service)
                            .thenCombine(
                                    CompletableFuture.supplyAsync(this::restCall2, service)
                                   ,this::mergeResults)
                            .join(); // join blocks in a virtual thread. so its okay.

        String output = mergeResults(result1, result2, result);
        return output;
    }

    // Once block ends, we know for sure that  all tasks have terminated.
}
```

Virtual Threads - REST Example

- Non Blocking IO
  - Socket reads, writes
  - File reads, writes
  - Concurrent locks

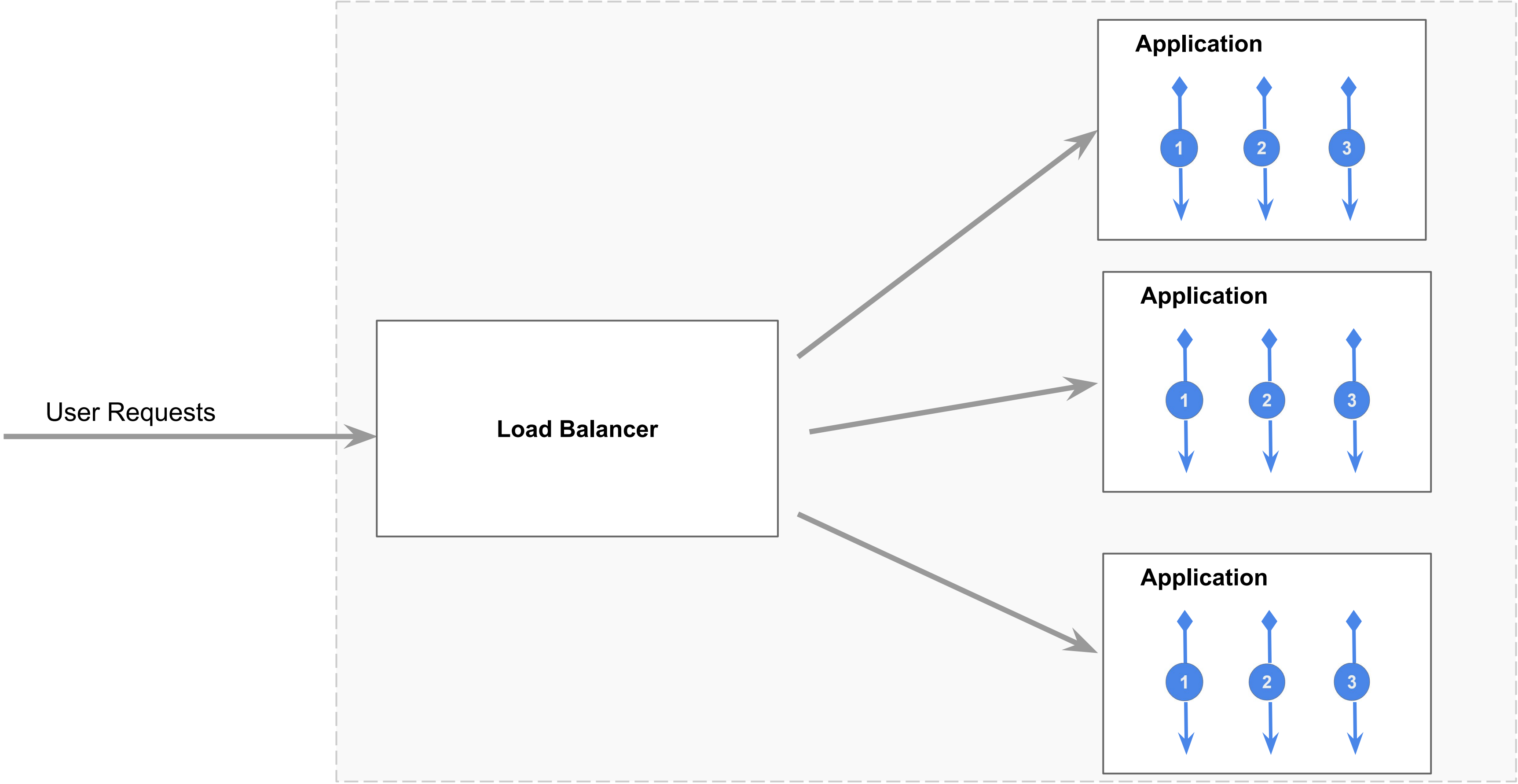
- REST Example
  - Socket handling
  - Will use httpbin.org

Virtual Threads - Scalability

Scalability Solution

{ Optimized Scalable Application } + { Vertical Scaling } + { Horizontal Scaling }

Enterprise Application using Virtual Threads - Dramatic Cost Reduction



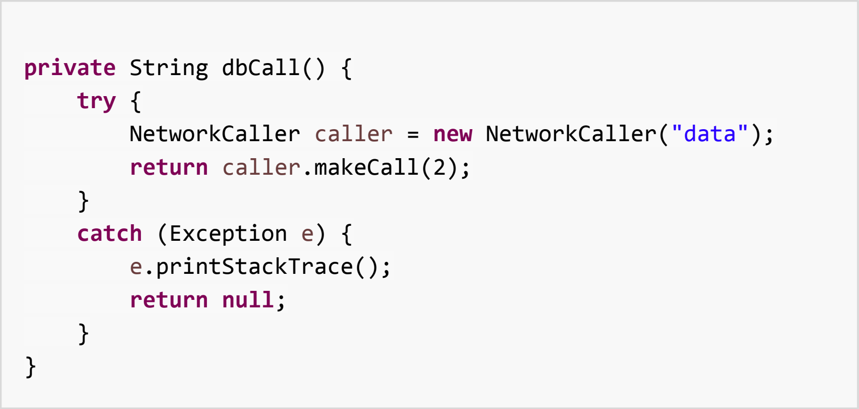
Structured Concurrency

```java
public class StructuredCodingExample {

    public static void main(String[] args) {
        List<Integer> result = getNumbersDivisibleBy5(51);
        System.out.println(result);
    }

    private static List<Integer> getNumbersDivisibleBy5(int num) {

        if (num < 1) {
            throw new RuntimeException("Invalid Input");
        }

        var result = new ArrayList<Integer>();
        for(int j=1; j <= num; j++) {

            if (j % 5 == 0) {
                result.add(j);
            }
        }

        return result;
    }

}
```

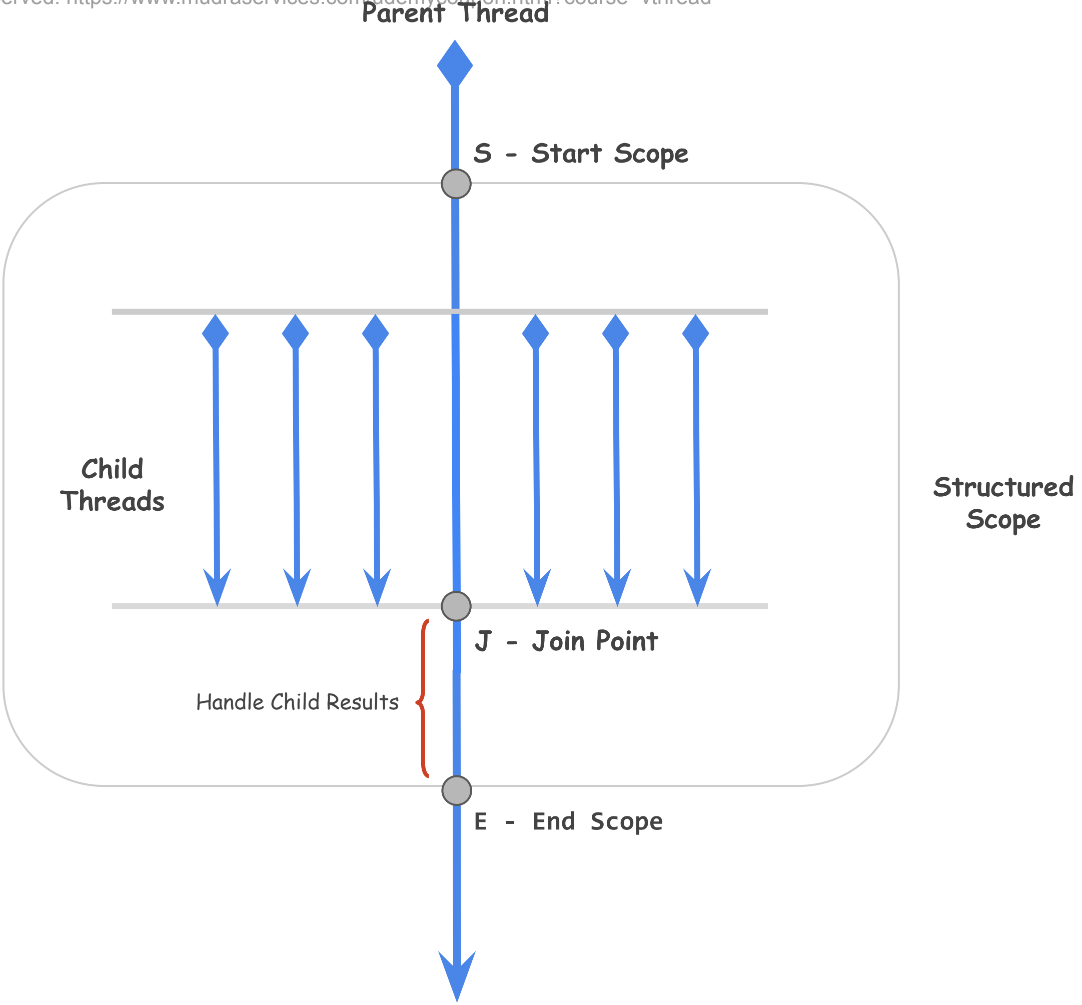
```
private void handleBusinessLogic() throws Exception {
    ExecutorService pool = ForkJoinPool.commonPool();
    Future<String> future = pool.submit(() -> {
        System.out.println(">> Starting worker thread .. ");

        doPartOfBusinessLogic();
        return "done";
    });

    // do other stuff ..
}

// The method ends but the submitted task may still be running
// The Worker Thread has leaked
```

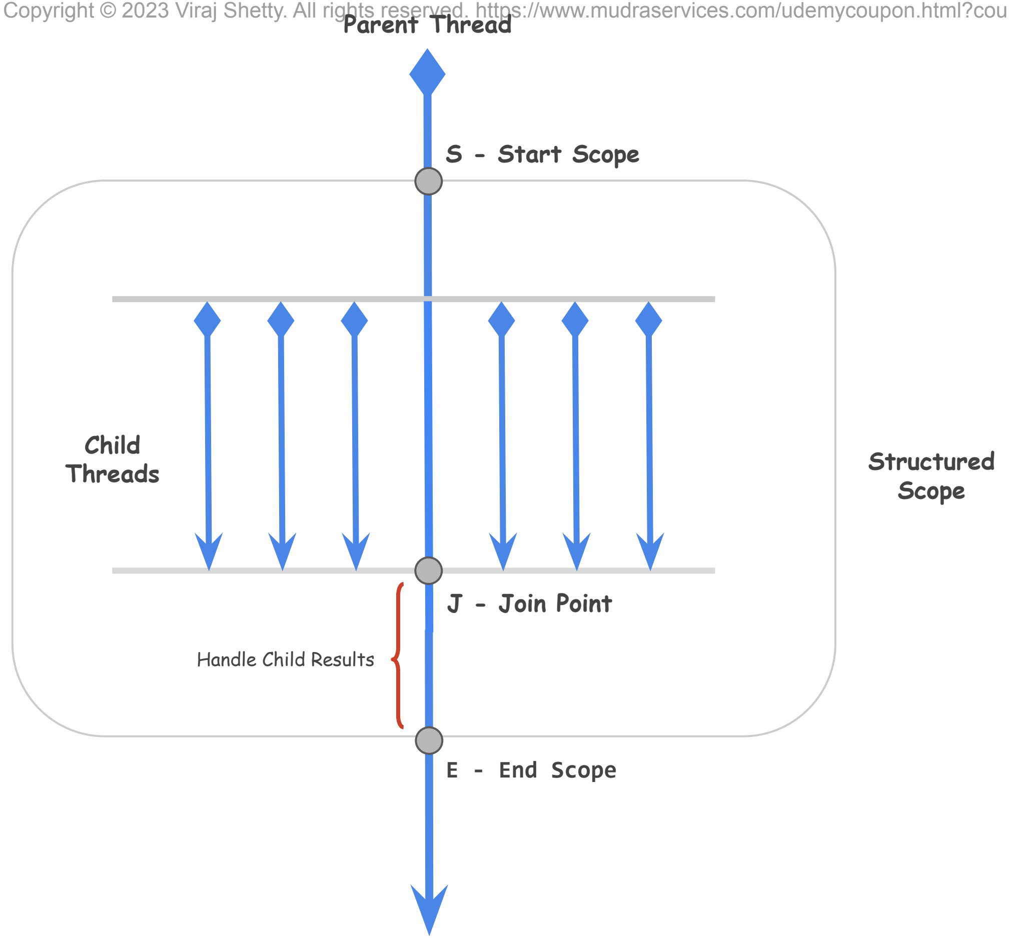
ExecutorService

```java
private String concurrentCallWithFutures() throws Exception {

    try (ExecutorService service = Executors.newVirtualThreadPerTaskExecutor()) {

        long start = System.currentTimeMillis();
        Future<String> dbFuture   = service.submit(this::dbCall);
        Future<String> restFuture = service.submit(this::restCall);

        String result = String.format("[%s,%s]", dbFuture.get(), restFuture.get());

        long end = System.currentTimeMillis();
        System.out.println("time = " + (end - start));

        System.out.println(result);
        return result;

    }

}
```



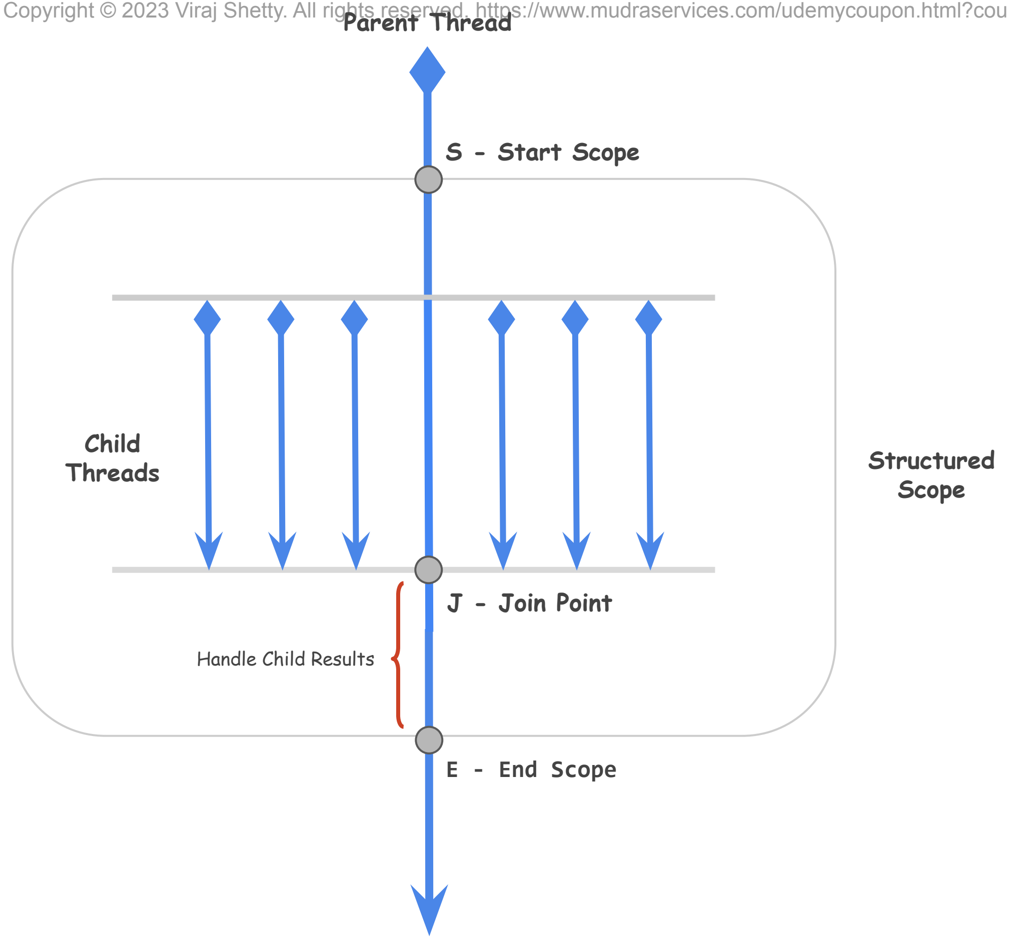
```java
private String dbCall() {
    try {
        NetworkCaller caller = new NetworkCaller("data");
        return caller.makeCall(2);
    }
    catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}
```


Parent Thread
S - Start Scope
Child
Threads
Structured
Scope
J - Join Point
Handle Child Results
E - End Scope



Use Cases

- Shutdown when all Child threads complete
  - *Example - Request Airfare prices from different travel sites*
- Shutdown when first Child Thread fails
  - *Example - Split an Enterprise use case into smaller parts and combine*
- Shutdown when first Child Thread succeeds
  - *Example - Request Weather information from multiple sites but choose first one*
- Custom



JDK Classes

- StructuredTaskScope
- Subtask

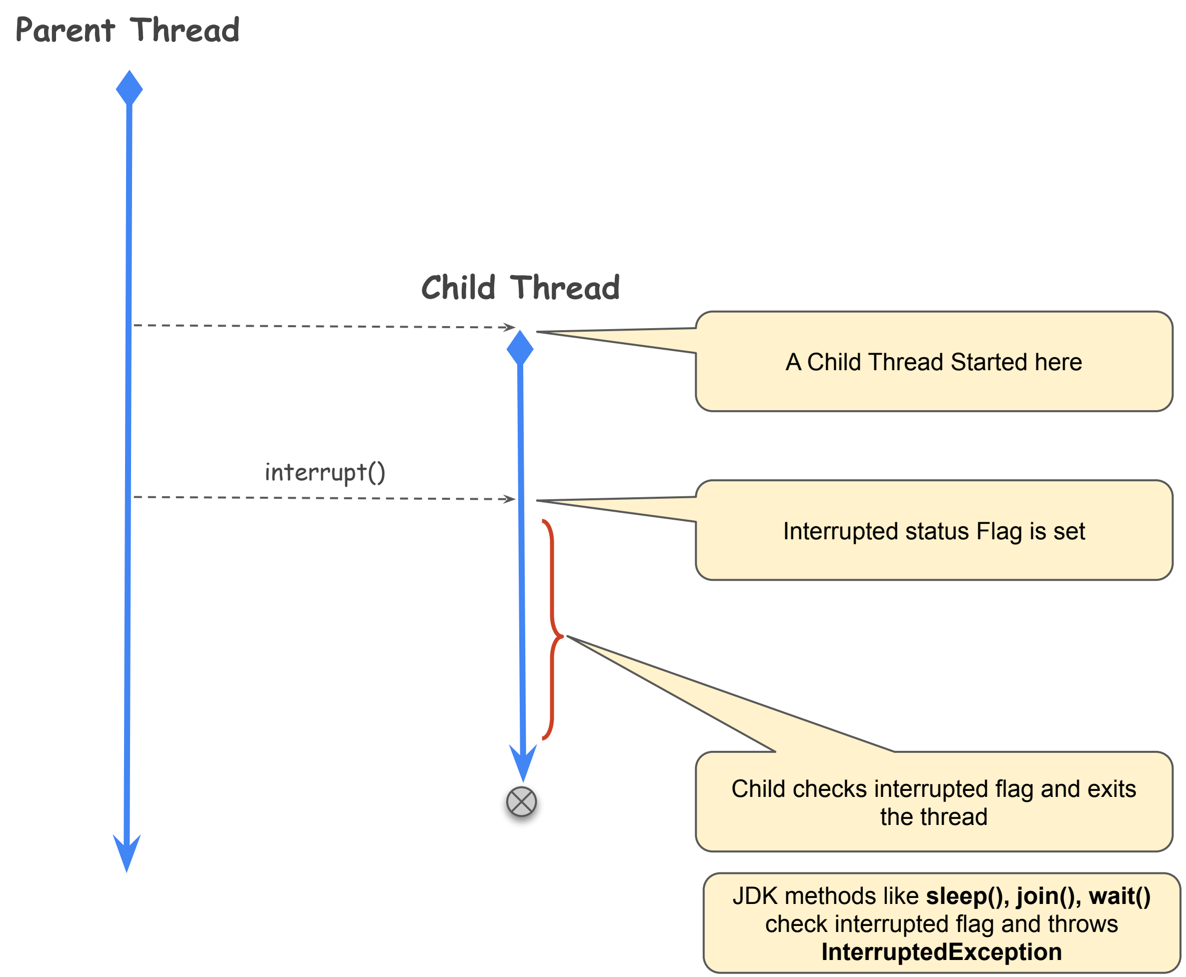
Thread Cancellation

Thread Cancellation

- Methods in the Thread Class

```
// Sets the ‘interrupted’ status flag for a thread to TRUE
public void interrupt()

// Checks the ‘interrupted’ status flag and if TRUE - clears it
public static boolean interrupted()

// Checks the ‘interrupted’ status flag but does not clear it
public boolean isInterrupted()
```

- Cooperative mechanism
- Both Platform Threads and Virtual Threads



A Child Thread Started here

Interrupted status Flag is set

Child checks interrupted flag and exits the thread

JDK methods like **sleep(), join(), wait()** check interrupted flag and throws **InterruptedException**

Thread Cancellation

- Interruptor must call `interrupt()` to set the flag
- Interrupted Thread must
  - May choose to ignore the interrupt
  - Check **interrupted** status periodically
  - JDK methods like `wait()`, `sleep()`, `join()` will check status automatically
    - Throws **InterruptedException**
    - Clears the **interrupted** status flag

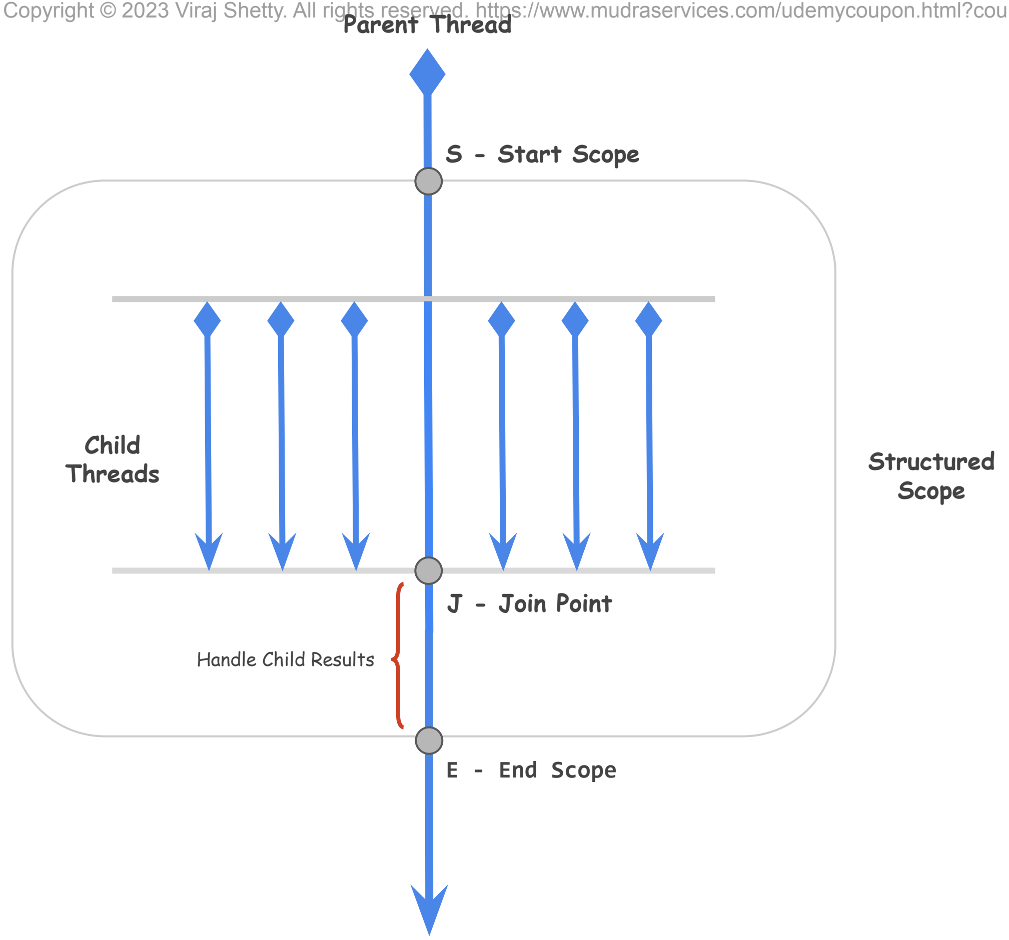
- Futures

```
// Sends an interrupt to the Child thread
Future<TaskResponse> taskFuture = exec.submit(callable);
taskFuture.cancel(true)
```

Writing a Long Running Task

Structured Concurrency Java Classes

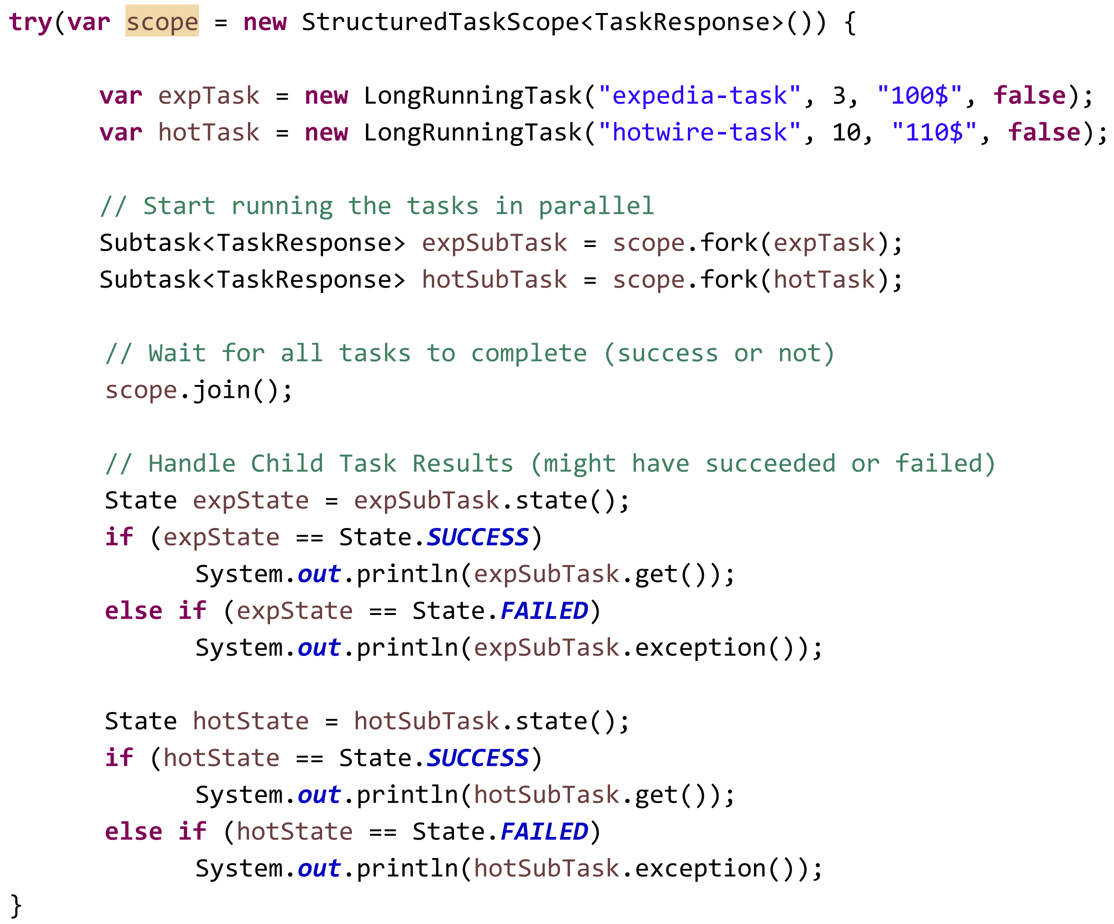
- StructuredTaskScope
- Subtask



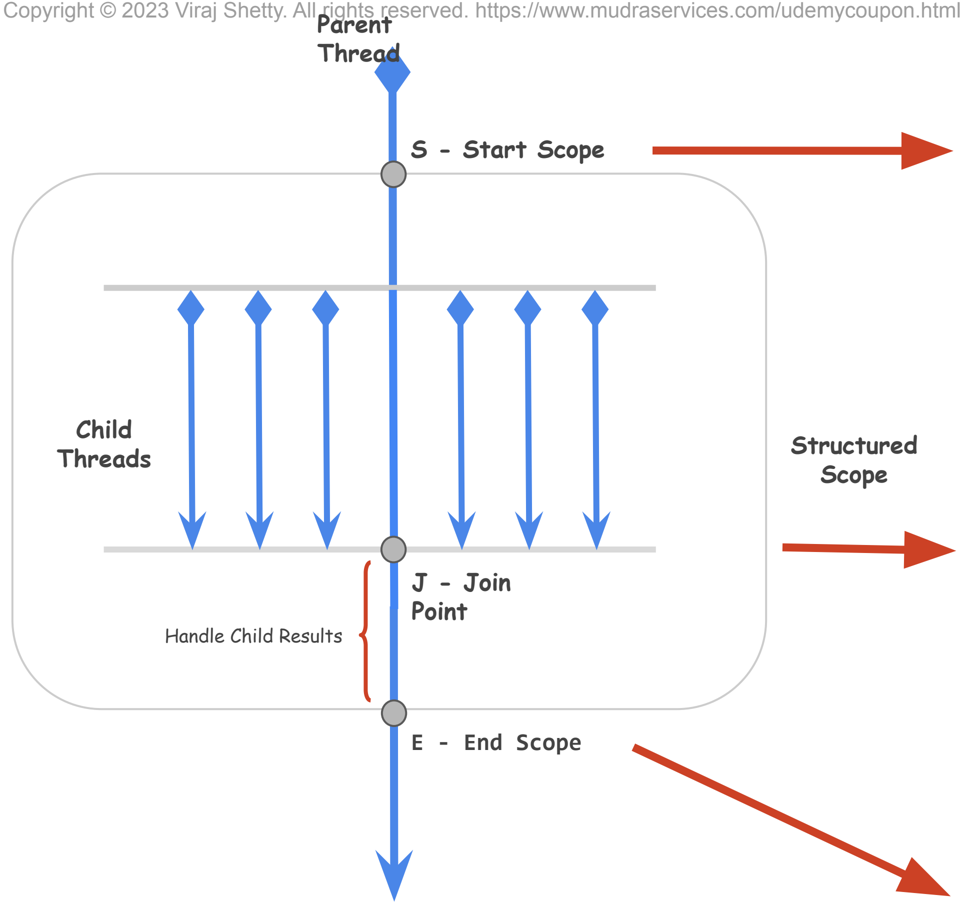
Use Cases

- Shutdown when all Child threads complete
  - *Example - Request Airfare prices from different travel sites*
- Shutdown when first Child Thread fails
  - *Example - Split an Enterprise use case into smaller parts and combine*
- Shutdown when first Child Thread succeeds
  - *Example - Request Weather information from multiple sites but choose first one*
- Custom

Shutdown when all Child Tasks complete (Default)



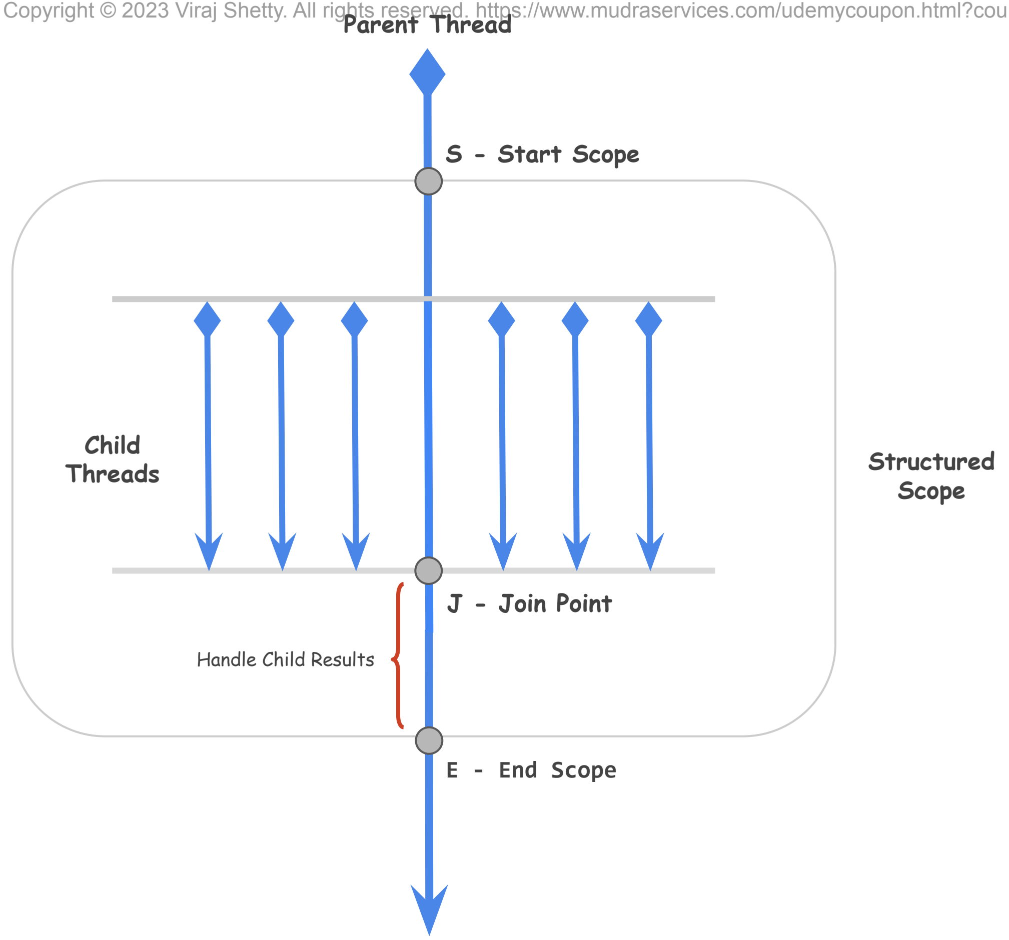
```java
try(var scope = new StructuredTaskScope<TaskResponse>()) {

      var expTask = new LongRunningTask("expedia-task", 3, "100$", false);
      var hotTask = new LongRunningTask("hotwire-task", 10, "110$", false);

      // Start running the tasks in parallel
      Subtask<TaskResponse> expSubTask = scope.fork(expTask);
      Subtask<TaskResponse> hotSubTask = scope.fork(hotTask);

      // Wait for all tasks to complete (success or not)
      scope.join();

      // Handle Child Task Results (might have succeeded or failed)
      State expState = expSubTask.state();
      if (expState == State.SUCCESS)
            System.out.println(expSubTask.get());
      else if (expState == State.FAILED)
            System.out.println(expSubTask.exception());

      State hotState = hotSubTask.state();
      if (hotState == State.SUCCESS)
            System.out.println(hotSubTask.get());
      else if (hotState == State.FAILED)
            System.out.println(hotSubTask.exception());
}
```



```java
try(var scope = new StructuredTaskScope<TaskResponse>()) {


    var expTask = new LongRunningTask("expedia-task", 3, "100$", true);
    var hotTask = new LongRunningTask("hotwire-task", 10, "110$", true);


    // Start running the tasks in parallel
    Subtask<TaskResponse> expSubTask = scope.fork(expTask);
    Subtask<TaskResponse> hotSubTask = scope.fork(hotTask);


    // Wait for all tasks to complete (success or not)
    scope.join();

    // Handle Child Task Results (might have succeeded or failed)
    State expState = expSubTask.state();
    if (expState == State.SUCCESS)
         System.out.println(expSubTask.get());
    else if (expState == State.FAILED)
         System.out.println(expSubTask.exception());

    State hotState = hotSubTask.state();
    if (hotState == State.SUCCESS)
         System.out.println(hotSubTask.get());
    else if (hotState == State.FAILED)
         System.out.println(hotSubTask.exception());
}
```

Demo StructuredTaskScope/Subtask

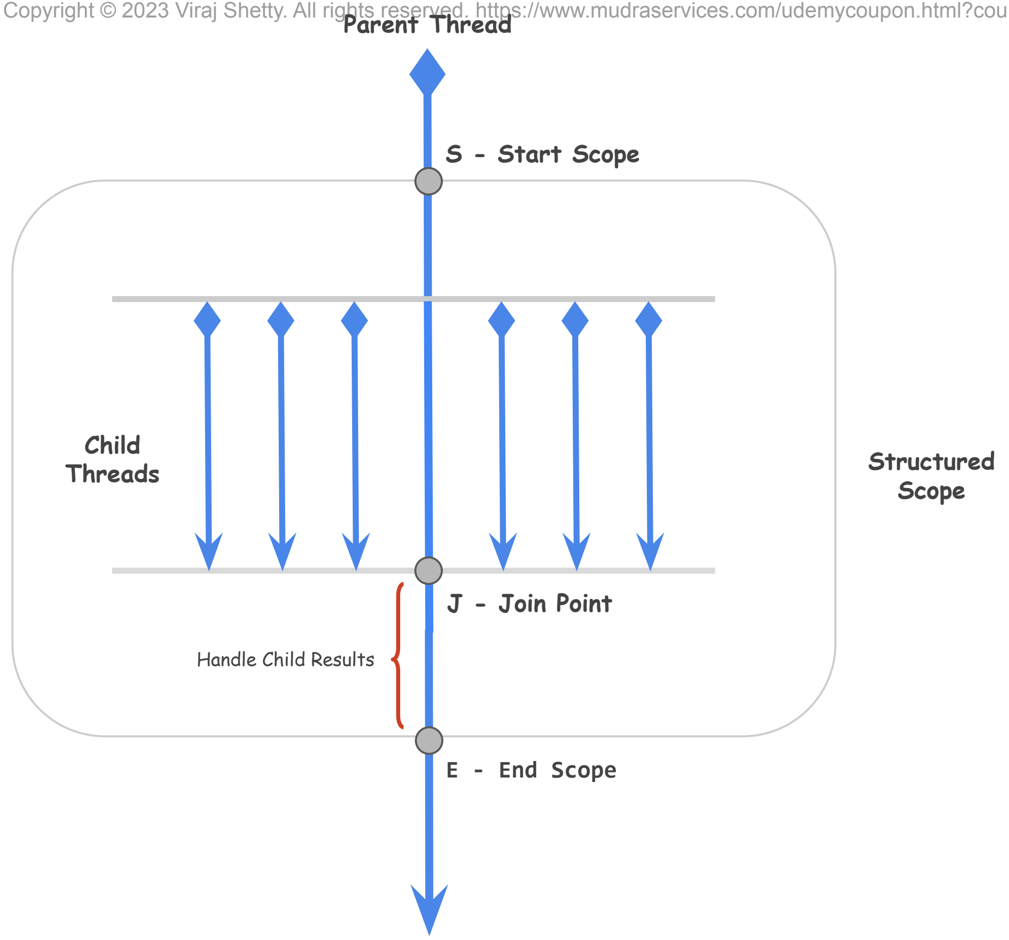
Use Cases



- Shutdown when all Child threads complete
  - *Example - Request Airfare prices from different travel sites*
- Shutdown when first Child Thread fails
  - *Example - Split an Enterprise use case into smaller parts and combine*
- Shutdown when first Child Thread succeeds
  - *Example - Request Weather information from multiple sites but choose first one*
- Custom

Shutdown when first Child Thread fails

```java
try(var scope = new StructuredTaskScope.ShutdownOnFailure()) {

    var dataTask = new LongRunningTask("dataTask", 3,  "row1", false);
    var restTask = new LongRunningTask("restTask", 10, "json2", false);

    // Start running the tasks in parallel
    Subtask<TaskResponse> dataSubTask = scope.fork(dataTask);
    Subtask<TaskResponse> restSubTask = scope.fork(restTask);

    // Wait till first Child Task fails. Send cancellation to
    // all other Child Tasks
    scope.join();
    scope.throwIfFailed();

    // Handle Success Child Task Results
    System.out.println(dataSubTask.get());
    System.out.println(restSubTask.get());
}
```

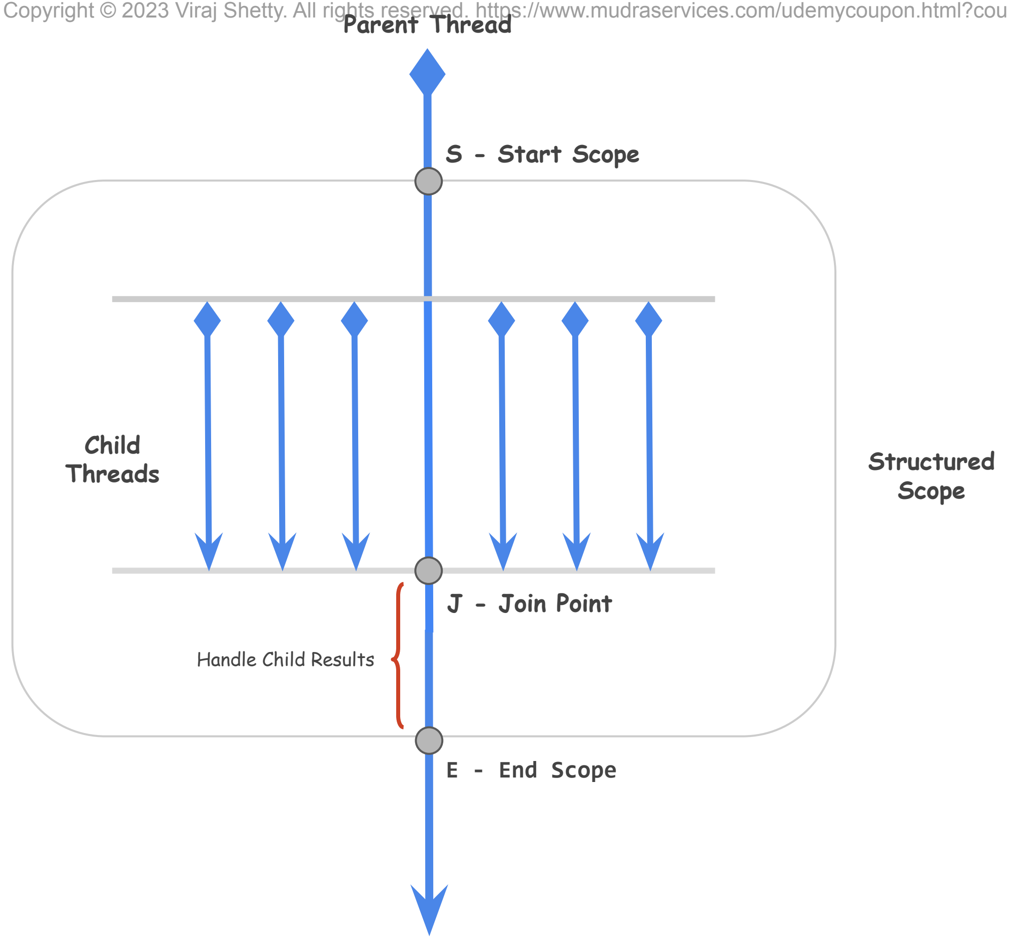


Use Cases

- Shutdown when all Child threads complete
  - *Example - Request Airfare prices from different travel sites*
- Shutdown when first Child Thread fails
  - *Example - Split an Enterprise use case into smaller parts and combine*
- Shutdown when first Child Thread succeeds
  - *Example - Request Weather information from multiple sites but choose first one*
- Custom

Shutdown when first Child Task succeeds

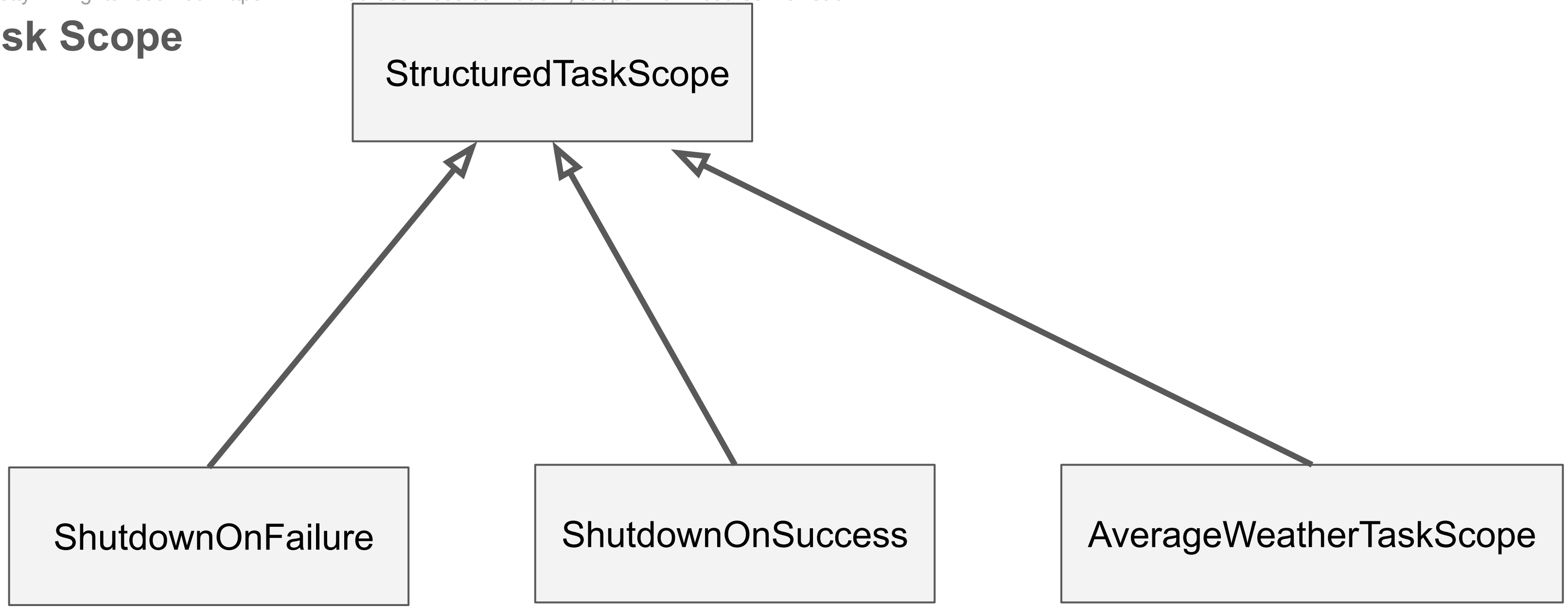
```java
try(var scope = new StructuredTaskScope.ShutdownOnSuccess<TaskResponse>()) {

    var wthr1Task = new LongRunningTask("Weather-1", 3,  "32", false);
    var wthr2Task = new LongRunningTask("Weather-2", 10, "30", false);

    // Start running the tasks in parallel
    Subtask<TaskResponse> subTask1 = scope.fork(wthr1Task);
    Subtask<TaskResponse> subTask2 = scope.fork(wthr2Task);

    // Wait till first Child Task Succeeds. Send Cancellation
    // to all other Child Tasks
    scope.join();

    // Handle Successful Child Task or throw ExecutionException
    TaskResponse result = scope.result();
    System.out.println(result);
}
```



Use Cases

- Shutdown when all Child threads complete
  - *Example - Request Airfare prices from different travel sites*
- Shutdown when first Child Thread fails
  - *Example - Split an Enterprise use case into smaller parts and combine*
- Shutdown when first Child Thread succeeds
  - *Example - Request Weather information from multiple sites but choose first one*
- Custom - Arbitrary rules

Custom Task Scope



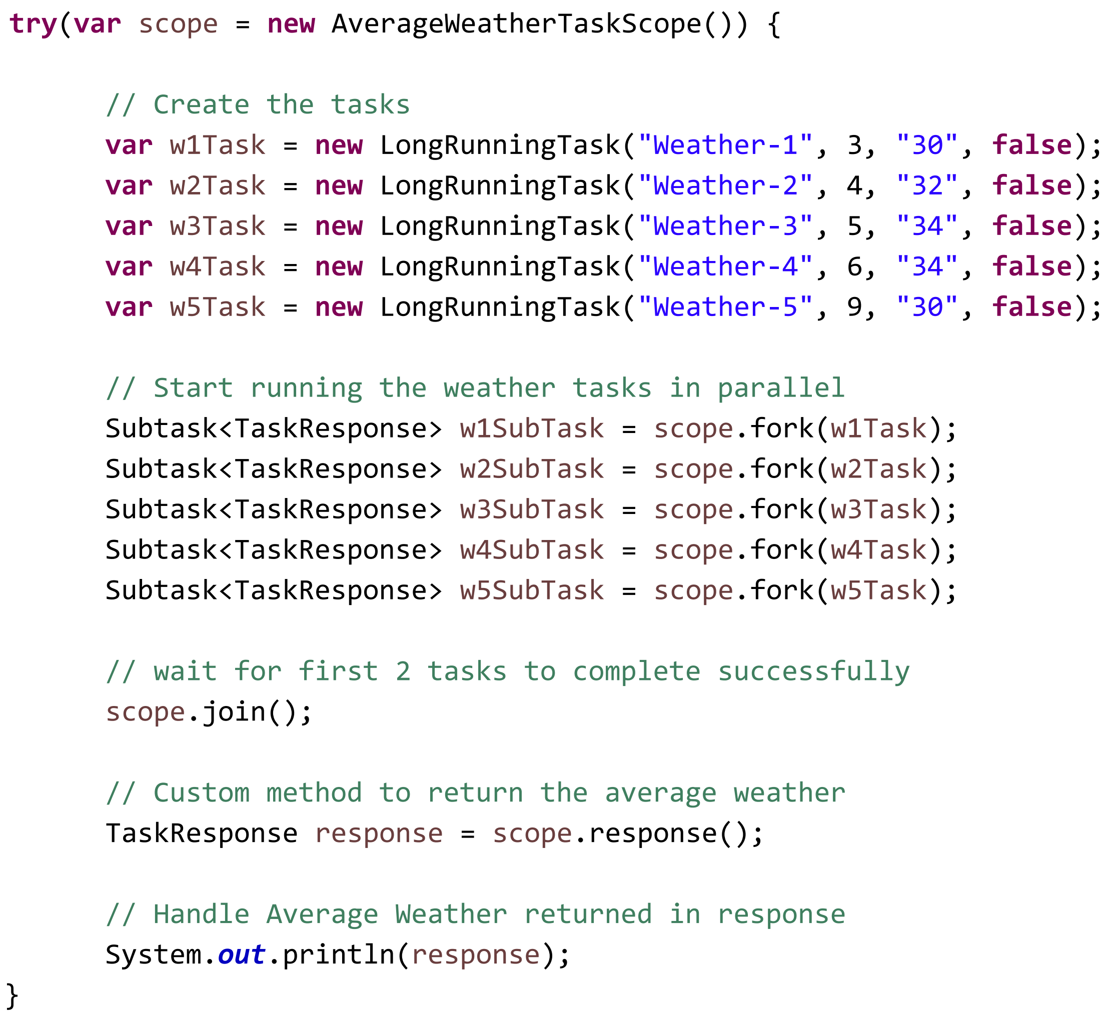
StructuredTaskScope

ShutdownOnFailure

ShutdownOnSuccess

AverageWeatherTaskScope

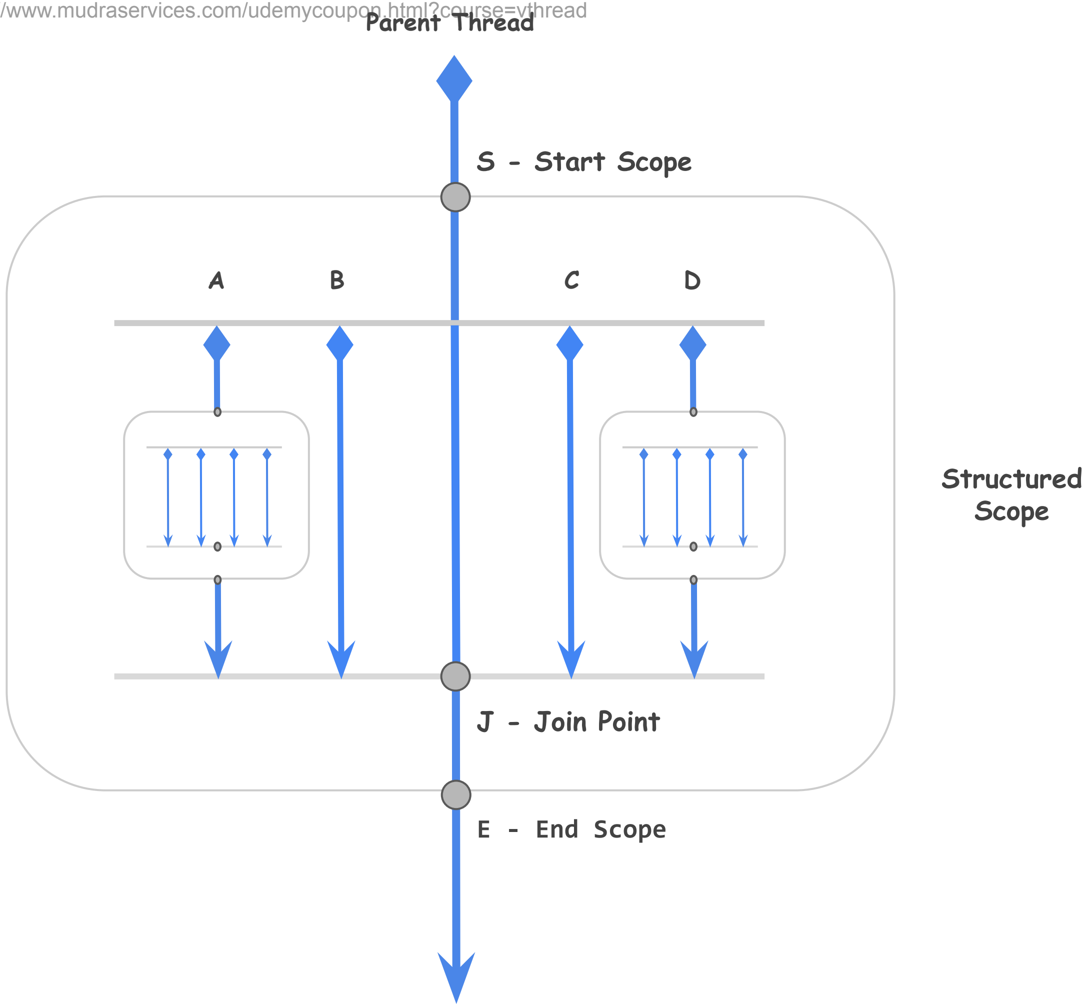
```
// Override this method
protected void handleComplete(Subtask<? extends T> subtask)

// Add any other Custom Methods in Custom Task Scope Class
```



```java
try(var scope = new AverageWeatherTaskScope()) {

      // Create the tasks
      var w1Task = new LongRunningTask("Weather-1", 3, "30", false);
      var w2Task = new LongRunningTask("Weather-2", 4, "32", false);
      var w3Task = new LongRunningTask("Weather-3", 5, "34", false);
      var w4Task = new LongRunningTask("Weather-4", 6, "34", false);
      var w5Task = new LongRunningTask("Weather-5", 9, "30", false);

      // Start running the weather tasks in parallel
      Subtask<TaskResponse> w1SubTask = scope.fork(w1Task);
      Subtask<TaskResponse> w2SubTask = scope.fork(w2Task);
      Subtask<TaskResponse> w3SubTask = scope.fork(w3Task);
      Subtask<TaskResponse> w4SubTask = scope.fork(w4Task);
      Subtask<TaskResponse> w5SubTask = scope.fork(w5Task);

      // wait for first 2 tasks to complete successfully
      scope.join();

      // Custom method to return the average weather
      TaskResponse response = scope.response();

      // Handle Average Weather returned in response
      System.out.println(response);
}
```

```java
public class AverageWeatherTaskScope extends StructuredTaskScope<TaskResponse> {
    private final List<Subtask<? extends TaskResponse>> successSubTasks
            = Collections.synchronizedList(new ArrayList<>());
    protected void handleComplete(Subtask<? extends TaskResponse> subtask) {
        if (subtask.state() == Subtask.State.SUCCESS)
            add(subtask);
    }
    private void add(Subtask<? extends TaskResponse> subtask) {
        int numSuccessful = 0;
        synchronized(successSubTasks) {
            successSubTasks.add(subtask);
            numSuccessful = successSubTasks.size();
        }

        if (numSuccessful == 2)
            this.shutdown();
    }
    public AverageWeatherTaskScope join() throws InterruptedException {
        super.join();
        return this;
    }
    public TaskResponse response() {
        super.ensureOwnerAndJoined();
        if (successSubTasks.size() != 2)
            throw new RuntimeException("Atleast two subtasks must be successful");

        TaskResponse r1 = successSubTasks.get(0).get(); TaskResponse r2 = successSubTasks.get(1).get();
        Integer temp1 = Integer.valueOf(r1.response());
        Integer temp2 = Integer.valueOf(r2.response());
        return new TaskResponse("Weather", "" + (temp1 + temp2)/2,  (r1.timeTaken() + r2.timeTaken())/2);
    }
}
```

Task Scope Hierarchy


Parent Thread
S - Start Scope
A
B
C
D
Structured Scope
J - Join Point
E - End Scope

Scoped Values

Recap

- Java Threads and Scalability Issues
- Virtual Threads
- Virtual Threads, Futures, Completable Futures
- Structured Concurrency

- Thread Locals
- Scoped Values

Java Scopes

```java
public class HttpCaller {

    public static final HttpClient client = HttpClient.newHttpClient();

    private final String callName;

    public HttpCaller(String callName) {
        this.callName = callName;
    }

    public String makeCall(int secs) throws InterruptedException {
        try {

            URI uri = new URI("http://httpbin.org/delay/" + secs);
            HttpRequest request = HttpRequest.newBuilder().GET().uri(uri).build();

            /* Rest of the code not shown */
        }
        catch (IOException | URISyntaxException exp) {
            throw new RuntimeException(exp);
        }
    }

}
```

Global Scope (**HttpCaller.client**)

Class Scope

Method Scope

Block Scope

Block Scope

ThreadLocal

```
// Declare a Thread Local Variable user
public static final ThreadLocal<User> user = new ThreadLocal<>();
```

```
// Sets the current thread’s value for user
user.set(new User("bob"));

// Gets the current thread’s value for user
User requestUser = user.get();
```

```
// Removes the current thread's value for user
user.remove();
```

```
// Declare a Thread Local Variable user with an Initializer
public static ThreadLocal<User> user = ThreadLocal.withInitial(() -> new User("anonymous"))

// Returns Anonymous but new user object per thread (Thread safe)
User requestUser = user.get();
```

```java
public class ThreadLocalSimplePlay {

    public static final ThreadLocal<User> user
                                  = new ThreadLocal<>();

    public static void main(String[] args) {

        print("User => " + user.get());

        // Main thread sets the user
        user.set(new User("anonymous"));
        print("User => " + user.get());

        handleUser();
    }

    private static void handleUser() {

        UserHandler handler = new UserHandler();
        handler.handle();
    }

    public static void print(String m) {
       System.out.printf("[%s] %s\n",
           Thread.currentThread().getName(), m);
    }
}
```

```java
public class UserHandler {

    public void handle() {

        User requestUser = ThreadLocalSimplePlay.user.get();
        print("handle - User => " + requestUser);

        // handle user 'requestUser'
    }

    public static void print(String m) {
        ThreadLocalSimplePlay.print(m);
    }

}
```

```java
public class User {
   private String id;

   // Constructor, getter, setter

   @Override
   public String toString() {
      return String.format("[%s, %s]",super.toString(), this.id);
   }
}
```

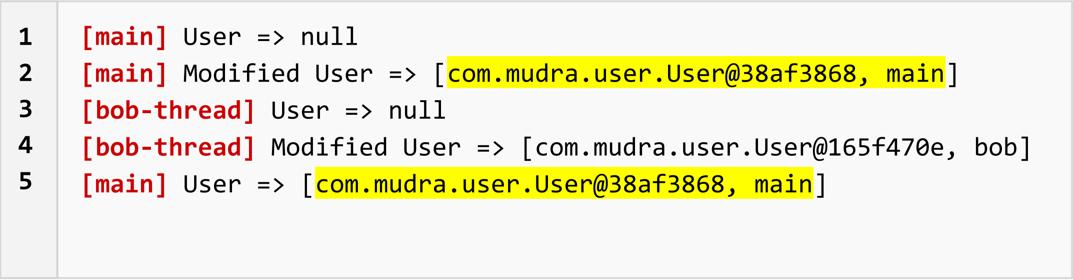
```
public class ThreadLocalSimplePlay {

   public static final ThreadLocal<User> user
                                 = new ThreadLocal<>();

   public static void main(String[] args) {

      print("User => " + user.get());

      // Main thread sets the user
      user.set(new User("anonymous"));
      print("User => " + user.get());

      handleUser();
   }

   private static void handleUser() {

      UserHandler handler = new UserHandler();
      handler.handle();
   }

   public static void print(String m) {
     System.out.printf("[%s] %s\n",
         Thread.currentThread().getName(), m);
   }
   // print method not shown
}
```

```
public class UserHandler {

   public void handle() {

      User requestUser = ThreadLocalSimplePlay.user.get();
      print("handle - User => " + requestUser);

      // handle user 'requestUser'
   }

   public static void print(String m) {
      ThreadLocalSimplePlay.print(m);
   }

}
```

```
[main] User => null
[main] User => [com.mudra.user.User@38af3868, anonymous]
[main] handle - User => [com.mudra.user.User@38af3868, anonymous]
```

```
    public class ThreadLocalPlay {

        public static final ThreadLocal<User> user = new ThreadLocal<User>();

        public static void main(String[] args) throws InterruptedException {

 1          print("User => " + user.get());

            // Main thread sets the user
            user.set(new User("main"));
 2          print("Modified User => " + user.get());

            // Start a Child Thread for "bob"
            Thread thread = Thread.ofVirtual().start(() -> {

                Thread.currentThread().setName("bob-thread");

 3              print("User => " + user.get());

                User.set(new User("bob"));
 4              print("Modified User => " + user.get());

            });

            thread.join();
 5          print("User => " + user.get());
        }

        // print method not shown

    }
```

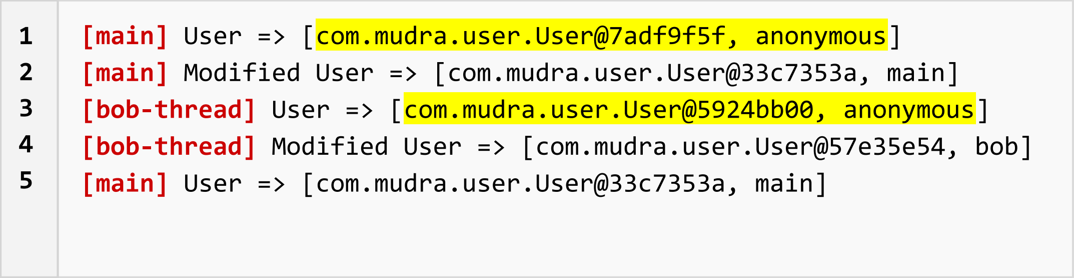
Run Output



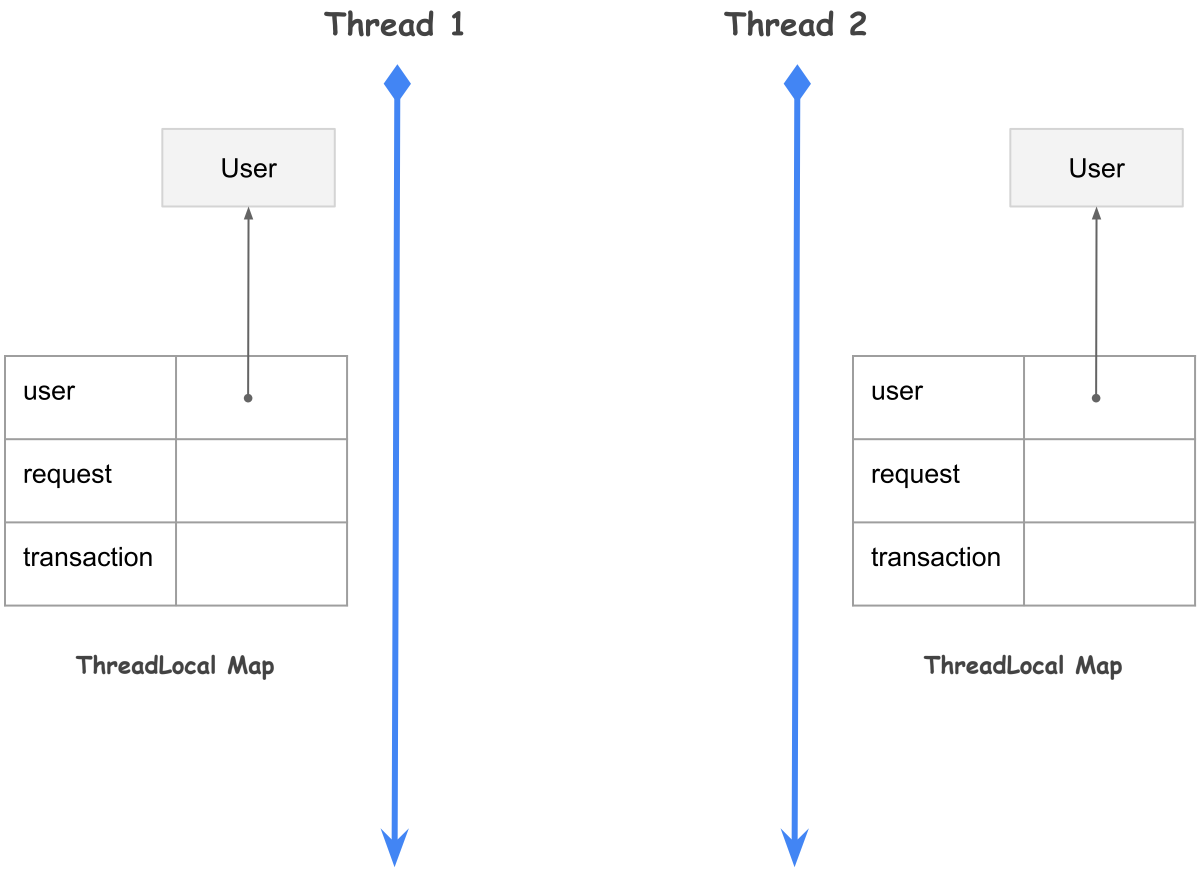
```
1   [main] User => null
2   [main] Modified User => [com.mudra.user.User@38af3868, main]
3   [bob-thread] User => null
4   [bob-thread] Modified User => [com.mudra.user.User@165f470e, bob]
5   [main] User => [com.mudra.user.User@38af3868, main]
```

```
  public class ThreadLocalInitializerPlay {

     public static final ThreadLocal<User> user = ThreadLocal.withInitial(() -> new User("anonymous"));

     public static void main(String[] args) throws InterruptedException {

1       print("User => " + user.get());

        // Main thread sets the user
        user.set(new User("main"));
2       print("Modified User => " + user.get());

        // Start a Child Thread for "bob"
        Thread thread = Thread.ofVirtual().start(() -> {

           Thread.currentThread().setName("bob-thread");

3          print("User => " + user.get());

           User.set(new User("bob"));
4          print("Modified User => " + user.get());

        });

        thread.join();
5       print("User => " + user.get());
     }

    // print method not shown

  }
```

Run Output



```
1  [main] User => [com.mudra.user.User@7adf9f5f, anonymous]
2  [main] Modified User => [com.mudra.user.User@33c7353a, main]
3  [bob-thread] User => [com.mudra.user.User@5924bb00, anonymous]
4  [bob-thread] Modified User => [com.mudra.user.User@57e35e54, bob]
5  [main] User => [com.mudra.user.User@33c7353a, main]
```



Thread 1

User

user	
request	
transaction	

ThreadLocal Map

Thread 2

User

user	
request	
transaction	

ThreadLocal Map

Inheritable Thread Local

InheritableThreadLocal

- Copied From Parent to Child Thread
  - By default, Child Value is identical to Parent Value (Thread-safety)
  - Child Value as function of Parent Value

```
// Declare a Inherited Thread Local Variable user
public static final InheritableThreadLocal<User> user = new InheritableThreadLocal<>();
```

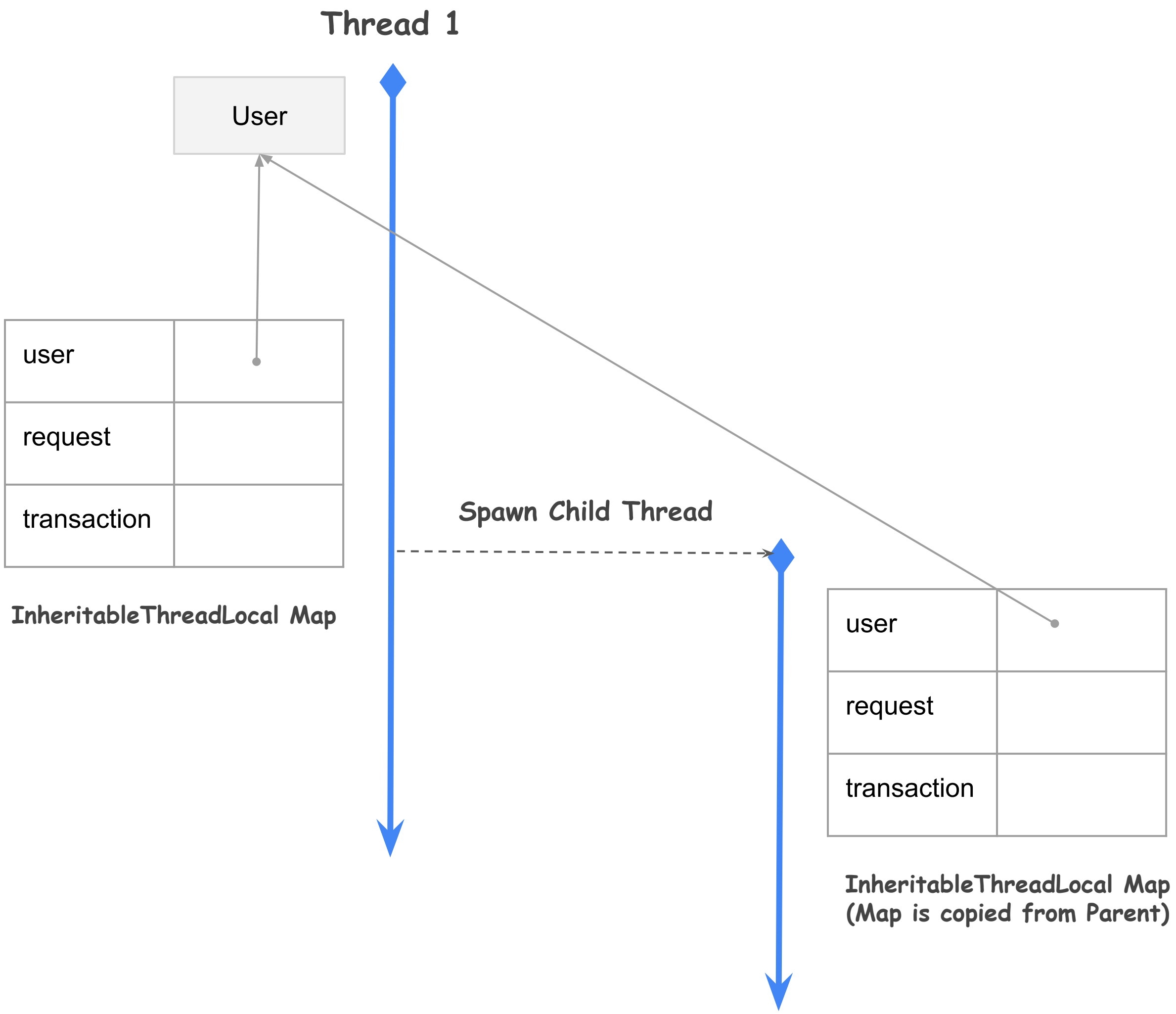
```
// Sets the current thread’s value for user
user.set(new User("bob"));

// Gets the current thread’s value for user
User requestUser = user.get();
```

```
// Removes the current thread's value for user
user.remove();
```

```
public class InheritableThreadLocalPlay {

    public static final InheritableThreadLocal<User> user = new InheritableThreadLocal<>();

    public static void main(String[] args) throws InterruptedException {

1       print("User => " + user.get());

        // Main thread sets the user
        user.set(new User("main"));
2       print("Modified User => " + user.get());

        // Start a Child Thread for "bob"
        Thread thread = Thread.ofVirtual().start(() -> {

            Thread.currentThread().setName("bob-thread");

3           print("User => " + user.get());

            user.get().setId("bobby");
4           print("Modified User => " + user.get());
        });

        thread.join();
5       print("User => " + user.get());
    }

    // print method not shown

}
```

Run Output

```
1  [main] User => null
2  [main] Modified User => [com.mudra.user.User@77459877, main]
3  [bob-thread] User => [com.mudra.user.User@77459877, main]
4  [bob-thread] Modified User => [com.mudra.user.User@77459877, bobby]
5  [main] User => [com.mudra.user.User@77459877, bobby]
```



Thread 1

User

user	
request	
transaction	

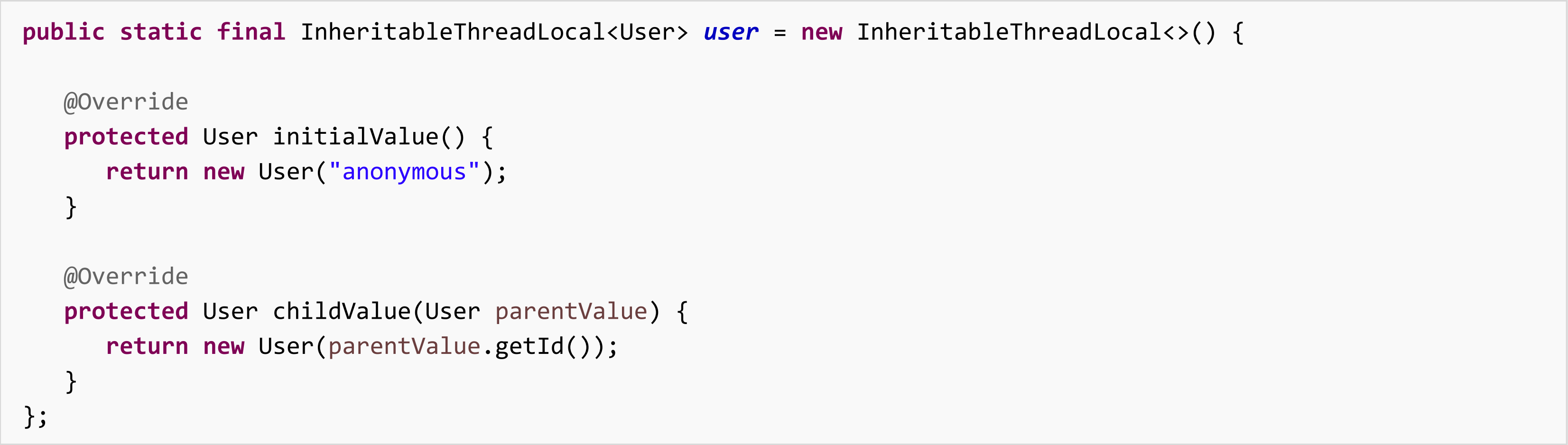
InheritableThreadLocal Map

Spawn Child Thread

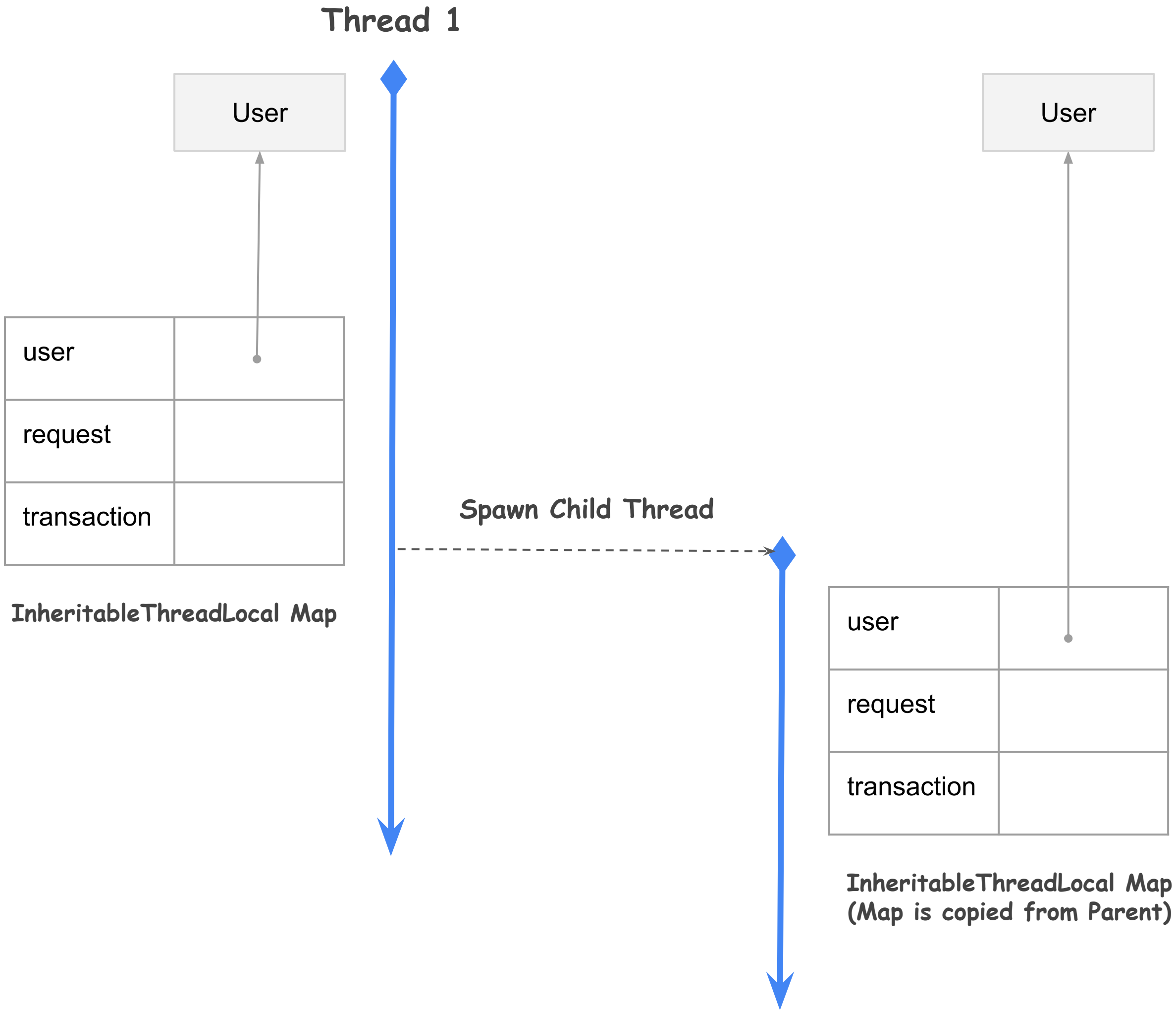
user	
request	
transaction	

InheritableThreadLocal Map
(Map is copied from Parent)

InheritableThreadLocal - Customize Child Value



```java
public static final InheritableThreadLocal<User> user = new InheritableThreadLocal<>() {

   @Override
   protected User initialValue() {
      return new User("anonymous");
   }

   @Override
   protected User childValue(User parentValue) {
      return new User(parentValue.getId());
   }
};
```



Thread 1

User

user	
request	
transaction	

InheritableThreadLocal Map

Spawn Child Thread

User

user	
request	
transaction	

InheritableThreadLocal Map
(Map is copied from Parent)

```
public class InheritableThreadLocalPlay {

    public static final InheritableThreadLocal<User> user = new InheritableThreadLocal<>() { <See Prev Slide> };

    public static void main(String[] args) throws InterruptedException {

1       print("User => " + user.get());

        // Main thread sets the user
        user.set(new User("main"));
2       print("Modified User => " + user.get());

        // Start a Child Thread for "bob"
        Thread thread = Thread.ofVirtual().start(() -> {

            Thread.currentThread().setName("bob-thread");

3           print("User => " + user.get());

            user.get().setId("bobby");
4           print("Modified User => " + user.get());
        });

        thread.join();
5       print("User => " + user.get());
    }

    // print method not shown

}
```

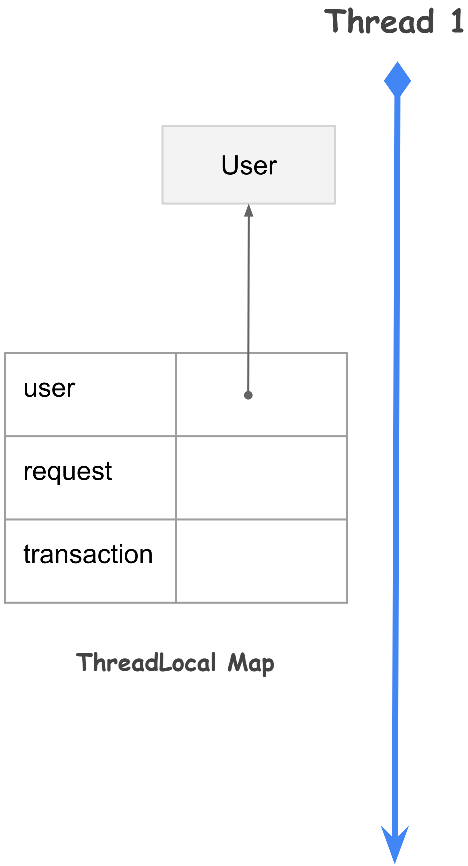
Run Output

```
1  [main] User => [com.mudra.user.User@2f92e0f4, anonymous]
2  [main] Modified User => [com.mudra.user.User@72ea2f77, main]
3  [bob-thread] User => [com.mudra.user.User@7a664ba2, main]
4  [bob-thread] Modified User => [com.mudra.user.User@7a664ba2, bobby]
5  [main] User => [com.mudra.user.User@72ea2f77, main]
```

Problems with Thread Locals

Unconstrained Mutability

- Leads to Spaghetti-like Data Flow

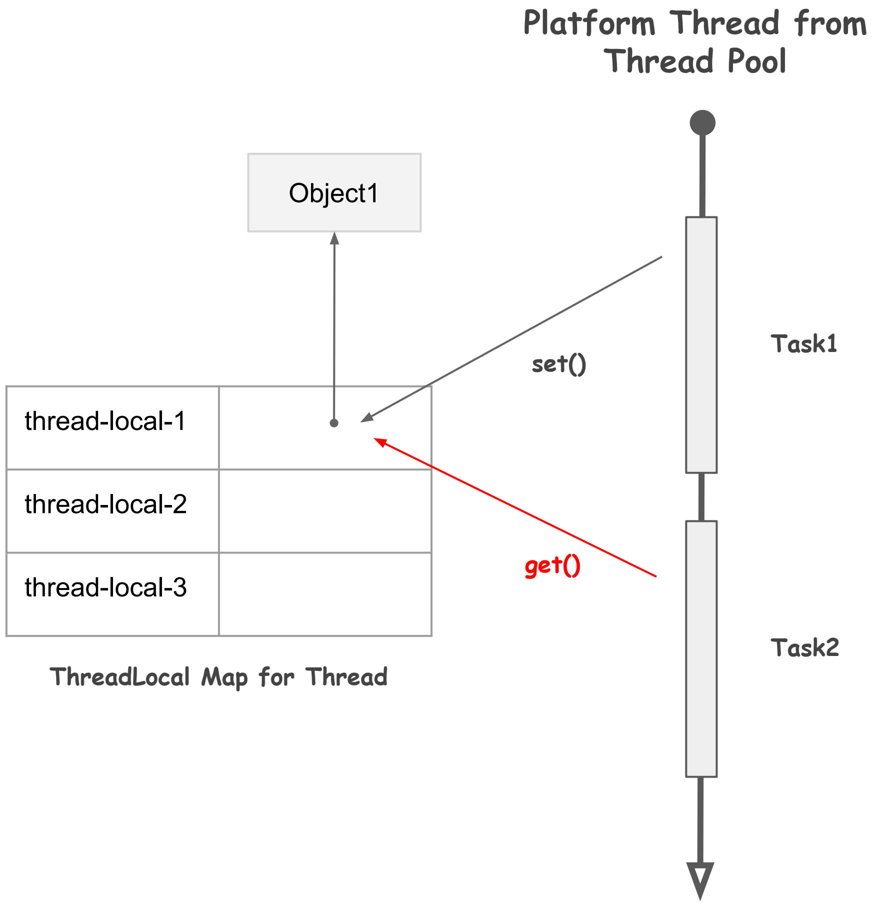


user	
request	
transaction	

ThreadLocal Map

Unbounded Lifetime

- remove() has to be called
- Beware of memory leaks



Expensive Inheritance

- Map needs to be copied to Child
- Possible Memory issues for large number of Virtual Threads

Scoped Values

- Can be accessed from within “dynamic” scope of a method
- Immutable

Thread Local

- Unconstrained Mutability
- Unconstrained Scope

Scoped Value

ScopedValue

```
// Creates a scoped value that is initially unbound for all threads
public static final ScopedValue<User> user = ScopedValue.newInstance();
```

```
// Binds the ‘user’ to bob within the scope of Callable method ‘handleUser’
User bob = new User("bob");
boolean result = ScopedValue.callWhere(user, bob, ScopedValuePlay::handleUser);

// Signature
public static <T, R> R callWhere(ScopedValue<T> key, T value, Callable<? extends R> op)
```

```
// Returns the value of the scoped value if bound in the current thread, otherwise Exception
User requestUser = user.get();
```

```
// Returns true if this scoped value is bound in the current thread
boolean bound = user.isBound();
```

```
// METHODS NOT AVAILABLE FOR SCOPED VALUE
user.set()
user.remove()
```

```java
public class ScopedValuePlay {

    public static final ScopedValue<User> user =
                                ScopedValue.newInstance();

    public static void main(String[] args) throws Exception {

        print("user is Bound => " + user.isBound());

        User bob = new User("bob");
        boolean result = ScopedValue.callWhere(user, bob,
                                ScopedValuePlay::handleUser);

        print("Result => " + result);
        print("user is Bound => " + user.isBound());
    }

    private static boolean handleUser() {
      ScopedUserHandler handler = new ScopedUserHandler();
      return handler.handle();
    }

    public static void print(String m) {
        System.out.printf("[%s] %s\n",
                    Thread.currentThread().getName(), m);
    }

}
```

```java
public class ScopedUserHandler {

    public boolean handle() {
        boolean bound = ScopedValuePlay.user.isBound();

        print("handle - user is Bound => " + bound);
        if (bound) {
            User requestUser = ScopedValuePlay.user.get();
            print("handle - User => " + requestUser);

            // handle user 'requestUser'
        }

        return bound;
    }
}
```

```java
public class User {
   private String id;

   // Constructor, getter, setter

   @Override
   public String toString() {
      return String.format("[%s, %s]",super.toString(), this.id);
   }
}
```

```java
public class ScopedValuePlay {

    public static final ScopedValue<User> user =
                               ScopedValue.newInstance();

    public static void main(String[] args) throws Exception {

1       print("user is Bound => " + user.isBound());

        User bob = new User("bob");
        boolean result = ScopedValue.callWhere(user, bob,
                                 ScopedValuePlay::handleUser);

4       print("Result => " + result);
5       print("user is Bound => " + user.isBound());
    }

    private static boolean handleUser() {
      ScopedUserHandler handler = new ScopedUserHandler();
      return handler.handle();
    }

    public static void print(String m) {
        System.out.printf("[%s] %s\n",
                    Thread.currentThread().getName(), m);
    }

}
```

```java
public class ScopedUserHandler {

    public boolean handle() {
        boolean bound = ScopedValuePlay.user.isBound();

2       print("handle - user is Bound => " + bound);
        if (bound) {
            User requestUser = ScopedValuePlay.user.get();
3           print("handle - User => " + requestUser);

            // handle user 'requestUser'
        }

        return bound;
    }
}
```

Run Output

```
1   [main] user is Bound => false
2   [main] handle - user is Bound => true
3   [main] handle - User => [com.mudra.user.User@816f27d, bob]
4   [main] Result => true
5   [main] user is Bound => false
```

ScopedValue

```java
// Binds the ‘user’ to bob within the scope of Runnable method ‘handleUser’
ScopedValue.runWhere(user, bob, ScopedValuePlay::methodWithNoReturn);
```

```java
// Binds the ‘user’ to bob within the scope of Supplier method ‘handleUser’
User bob = new User("bob");
boolean result = ScopedValue.getWhere(user, bob, ScopedValuePlay::handleUser);
```

```java
// Declare the two Scoped Values
public static final ScopedValue<User> user = ScopedValue.newInstance();
public static final ScopedValue<Request> request = ScopedValue.newInstance();

// Binds the user to bob, request to HttpRequest within the scope of Callable method ‘handleUser’
boolean result = ScopedValue.where(user, bob)
                            .where(request, httpRequest)
                            .call(ScopedValuePlay::handleUser);
```

ScopedValue

```
// Return the value of the user if bound; otherwise return an anonymous user
User requestUser = user.orElse(new User("anonymous"));
```

```
// Return the value of the user if bound; otherwise throw a RuntimeException
User requestUser = user.orElseThrow(() -> new RuntimeException("No User bound"));
```

Rebinding Scoped Values

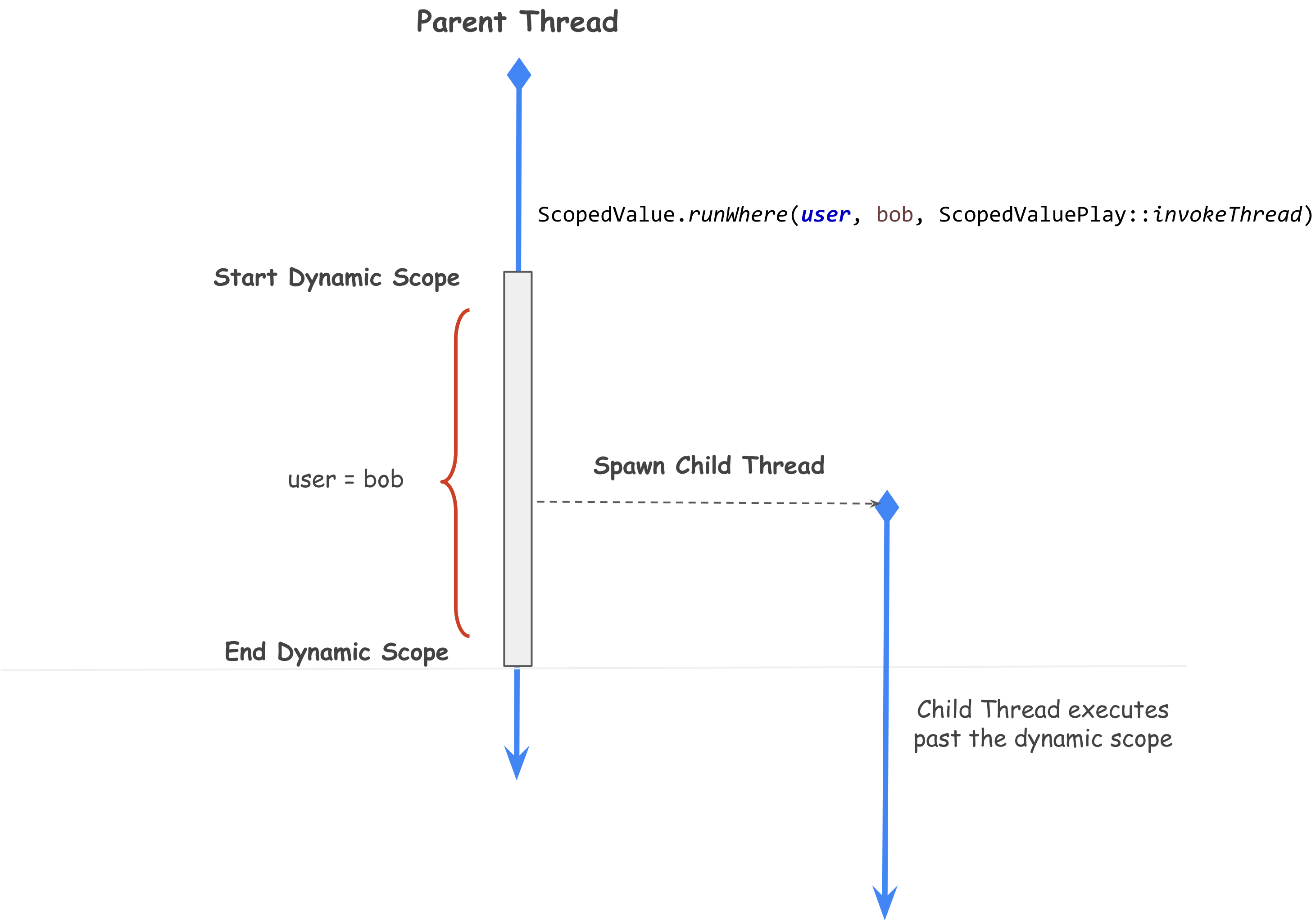
```java
public class ScopedValueRebindPlay {
    public static final ScopedValue<User> user = ScopedValue.newInstance();

    public static void main(String[] args) throws Exception {

        print("user is Bound => " + user.isBound());
        User bob = new User("bob");
        ScopedValue.runWhere(user, bob, ScopedValueRebindPlay::handleUser);
        print("user is Bound => " + user.isBound());
    }

    private static void handleUser() {

        print("handleUser - " + user.get());

        ScopedValue.runWhere(user, new User("anonymous"),
                        ScopedValueRebindPlay::callAsAnonymous);

        print("handleUser - " + user.get());
    }

    private static void callAsAnonymous() {
        print("callAsAnonymous - " + user.get());
    }

    public static void print(String m) {
        System.out.printf("[%s] %s\n", Thread.currentThread().getName(), m);
    }
}
```

```
    public class ScopedValueRebindPlay {
        public static final ScopedValue<User> user = ScopedValue.newInstance();

        public static void main(String[] args) throws Exception {

1           print("user is Bound => " + user.isBound());
            User bob = new User("bob");
            ScopedValue.runWhere(user, bob, ScopedValueRebindPlay::handleUser);
5           print("user is Bound => " + user.isBound());
        }

        private static void handleUser() {

2           print("handleUser - " + user.get());

            ScopedValue.runWhere(user, new User("anonymous"),
                                ScopedValueRebindPlay::callAsAnonymous);

4           print("handleUser - " + user.get());
        }

        private static void callAsAnonymous() {
3           print("callAsAnonymous - " + user.get());
        }

        public static void print(String m) {
            System.out.printf("[%s] %s\n", Thread.currentThread().getName(), m);
        }
    }
```

Run Output

```
1  [main] user is Bound => false
2  [main] handleUser - [com.mudra.user.User@1fb3ebeb, bob]
3  [main] callAsAnonymous - [com.mudra.user.User@1218025c, anonymous]
4  [main] handleUser - [com.mudra.user.User@1fb3ebeb, bob]
5  [main] user is Bound => false
```

Inheriting Scoped Values



Parent Thread

```
ScopedValue.runWhere(user, bob, ScopedValuePlay::invokeThread)
```

Start Dynamic Scope

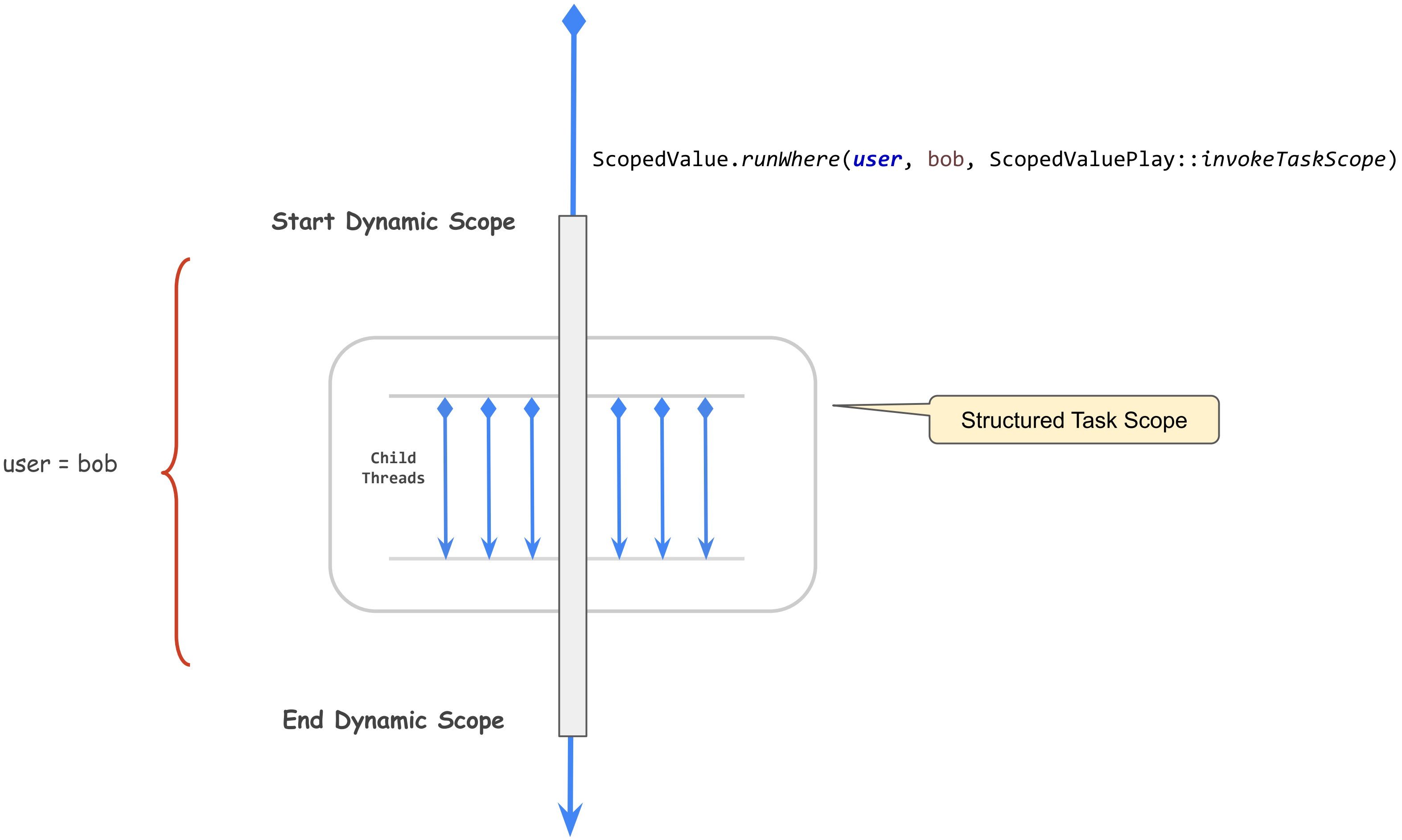
user = bob

Spawn Child Thread

End Dynamic Scope

Child Thread executes past the dynamic scope

Parent Thread



ScopedValue.*runWhere*(***user***, bob, ScopedValuePlay::*invokeTaskScope*)

Start Dynamic Scope

user = bob

Child Threads

Structured Task Scope

End Dynamic Scope

```
public class ScopedValueThreadPlay {
    public static final ScopedValue<User> user = ScopedValue.newInstance();

    public static void main(String[] args) throws Exception {
        ScopedValue.where(user, new User("sally")).run(ScopedValueThreadPlay::invokeThread);
    }

    private static void invokeThread() {
        try {
            print("user is Bound => " + user.isBound());

            User reqUser = user.get();
            String thrName = reqUser.getId() + "-thread";
            Thread thr = Thread.ofVirtual().name(thrName).start(() -> {
                print("user is Bound => " + user.isBound());
                User requestUser = user.orElse(new User("anonymous"));
                print("invokeThread - user " + requestUser);
            });

            thr.join();

        } catch (InterruptedException exp) {
            /* do something */
        }
    }

    public static void print(String m) {
        System.out.printf("[%s] %s\n", Thread.currentThread().getName(), m);
    }
}
```

```
    public class ScopedValueThreadPlay {
        public static final ScopedValue<User> user = ScopedValue.newInstance();

        public static void main(String[] args) throws Exception {
            ScopedValue.where(user, new User("sally")).run(ScopedValueThreadPlay::invokeThread);
        }

        private static void invokeThread() {
            try {
1               print("user is Bound => " + user.isBound());

                User reqUser = user.get();
                String thrName = reqUser.getId() + "-thread";
                Thread thr = Thread.ofVirtual().name(thrName).start(() -> {
2                   print("user is Bound => " + user.isBound());
                    User requestUser = user.orElse(new User("anonymous"));
3                   print("invokeThread - user " + requestUser);
                });

                thr.join();

            } catch (InterruptedException exp) {
                /* do something */
            }
        }

        public static void print(String m) {
            System.out.printf("[%s] %s\n", Thread.currentThread().getName(), m);
        }
    }
```

Run Output

```
1  [main] user is Bound => true
2  [sally-thread] user is Bound => false
3  [sally-thread] invokeThread - user [com.mudra.user.., anonymous]
```

```java
public class ScopedValueTaskScopePlay {
   public static final ScopedValue<User> user = ScopedValue.newInstance();

   public static void main(String[] args) throws Exception {
      ScopedValue.where(user, new User("sally")).call(ScopedValueTaskScopePlay::invokeTaskScope);
   }

   private static String invokeTaskScope() throws Exception {

      ThreadFactory factory = Thread.ofVirtual().name("test-",0).factory();
      try (var scope = new StructuredTaskScope<String>("test-scope", factory)) {
         scope.fork(() -> {

            User reqUser = user.orElse(new User("anonymous"));
            print("invokeTaskScope - user " + reqUser);

            // set the Id for the user
            reqUser.setId("bob");
            return "done";
         });

         scope.join();
      }

      User reqUser = user.orElse(new User("anonymous"));
      print("invokeTaskScope - user " + reqUser);

      return "done";
   }
}
```

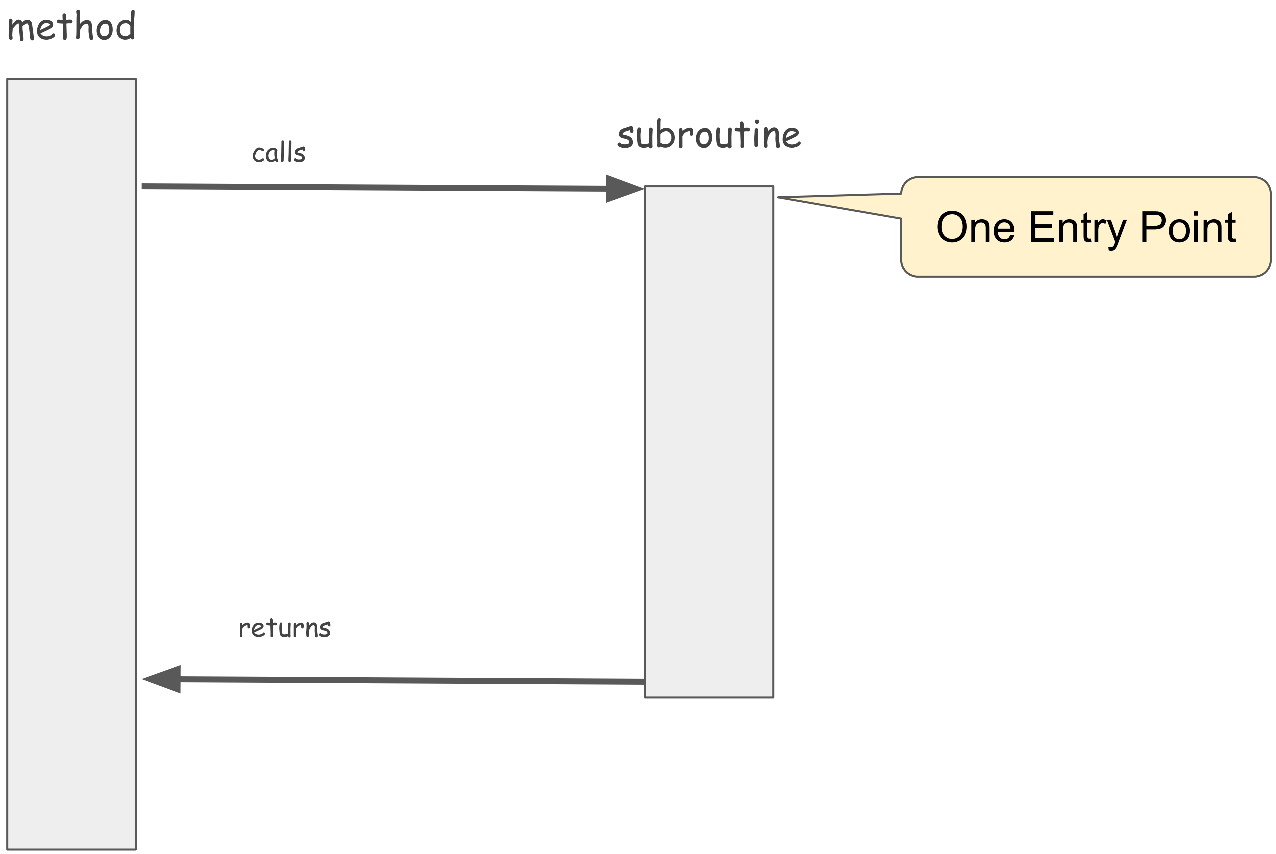
```
  public class ScopedValueTaskScopePlay {
      public static final ScopedValue<User> user = ScopedValue.newInstance();

      public static void main(String[] args) throws Exception {
          ScopedValue.where(user, new User("sally")).call(ScopedValueTaskScopePlay::invokeTaskScope);
      }

      private static String invokeTaskScope() throws Exception {

          ThreadFactory factory = Thread.ofVirtual().name("test-",0).factory();
          try (var scope = new StructuredTaskScope<String>("test-scope", factory)) {
              scope.fork(() -> {

                  User reqUser = user.orElse(new User("anonymous"));
1                 print("invokeTaskScope - user " + reqUser);

                  // set the Id for the user
                  reqUser.setId("bob");
                  return "done";
              });

              scope.join();
          }

          User reqUser = user.orElse(new User("anonymous"));
2         print("invokeTaskScope - user " + reqUser);

          return "done";
      }
  }
```

Run Output

```
1   [test-0] invokeTaskScope - user [com.mudra.user.User@52220b24, sally]
2   [main] invokeTaskScope - user [com.mudra.user.User@52220b24, bob]
```

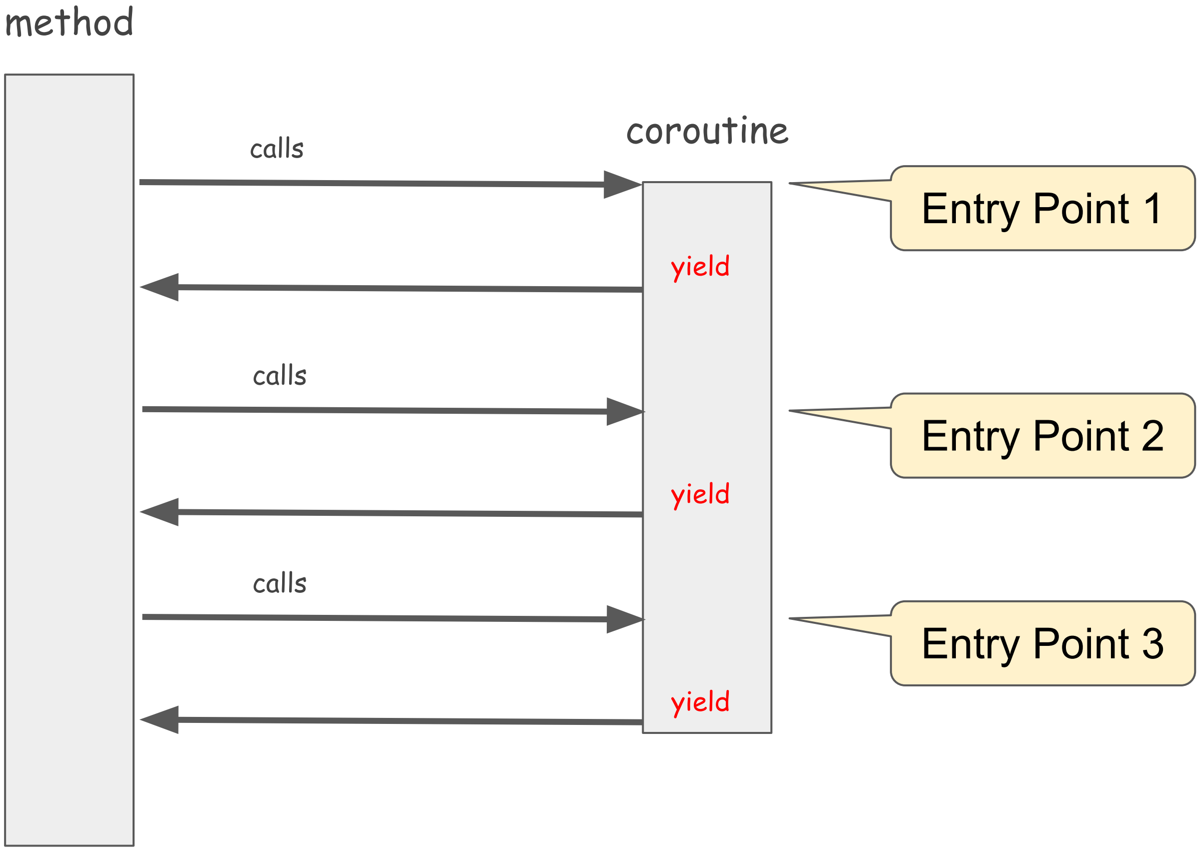
Coroutines and Continuations

Subroutine



One Entry Point

Coroutine

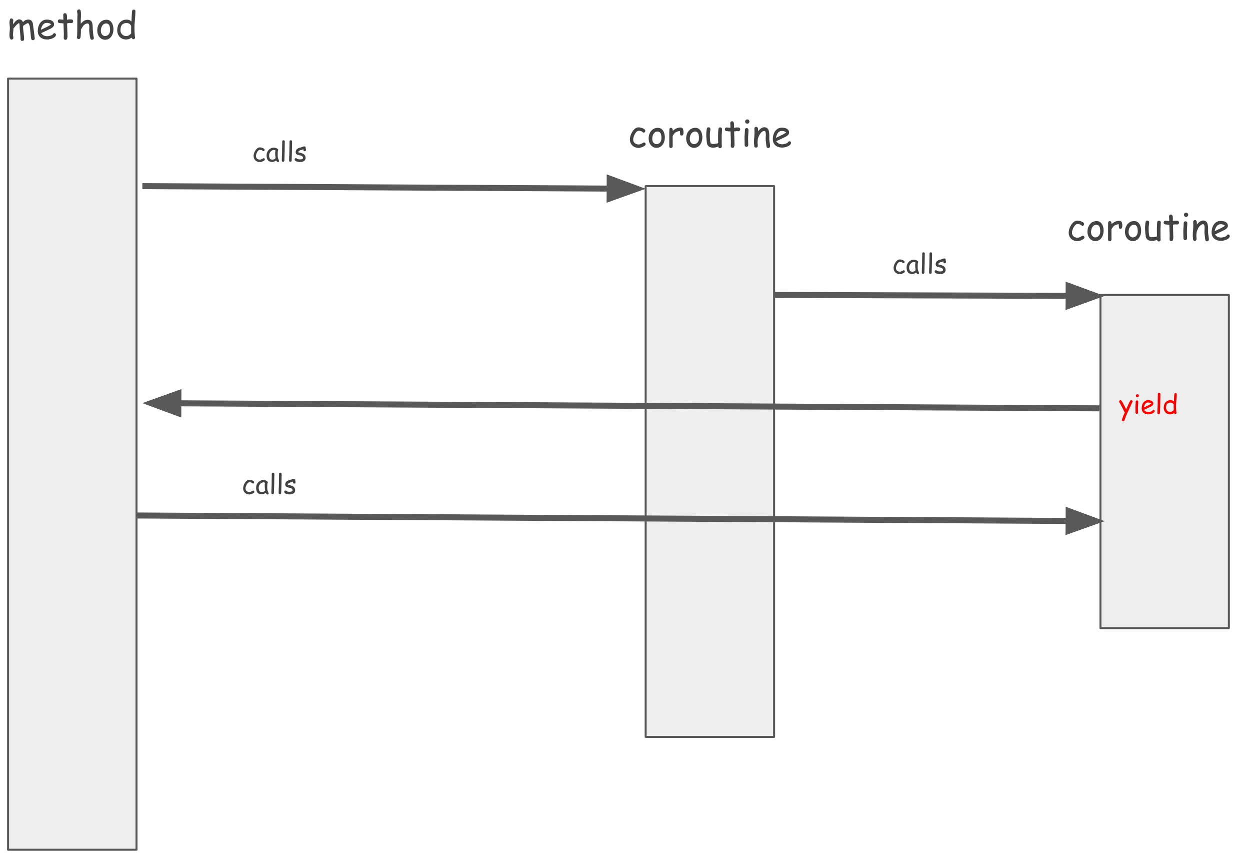


Entry Point 1

Entry Point 2

Entry Point 3

Coroutine



Coroutine

- Available in major programming languages
  - JavaScript
  - Kotlin
  - Go
  - Python

- With Project Loom, Java will have Continuation
  - Delimited Continuation
  - Unclear if it will be exposed as API

- Exceptions

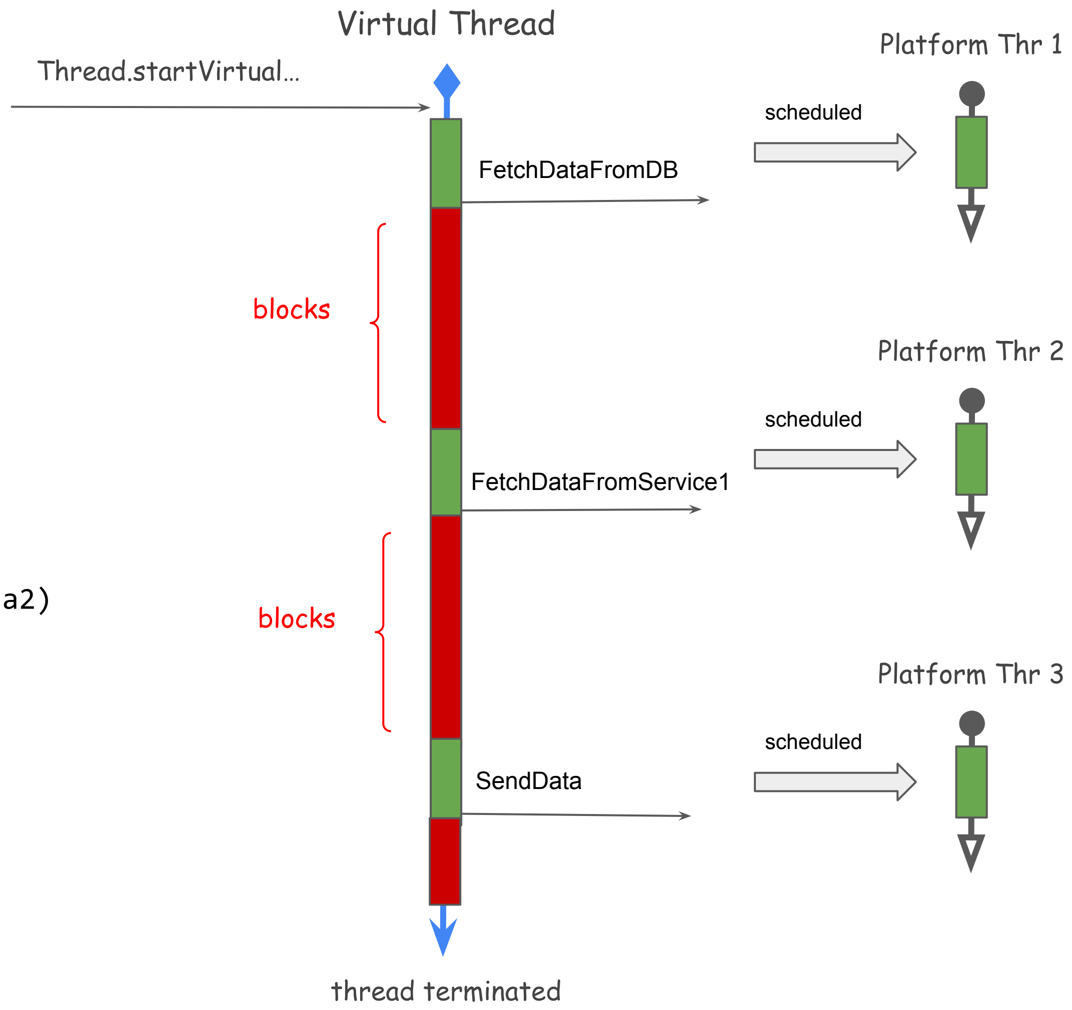
Demo of

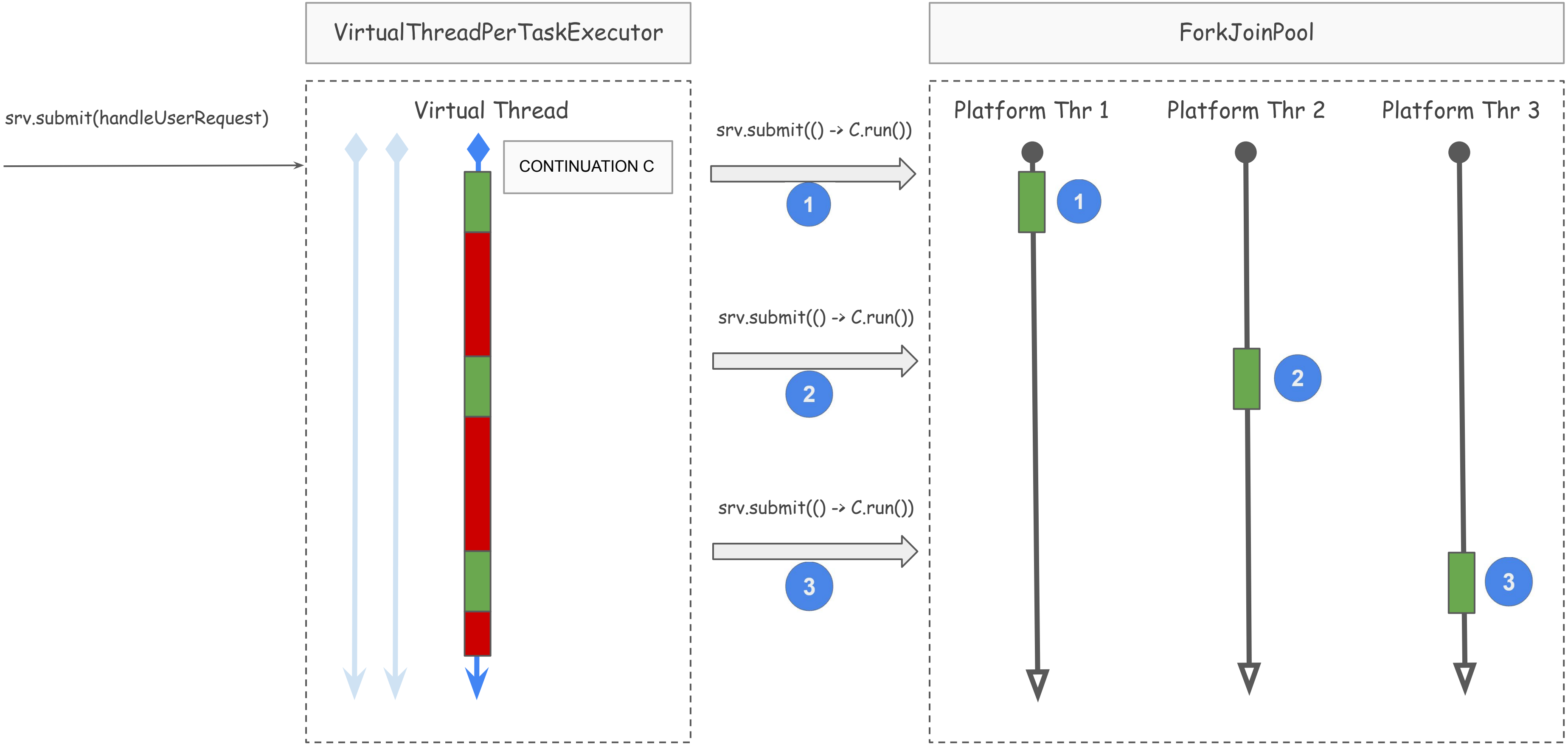
Delimited Continuations

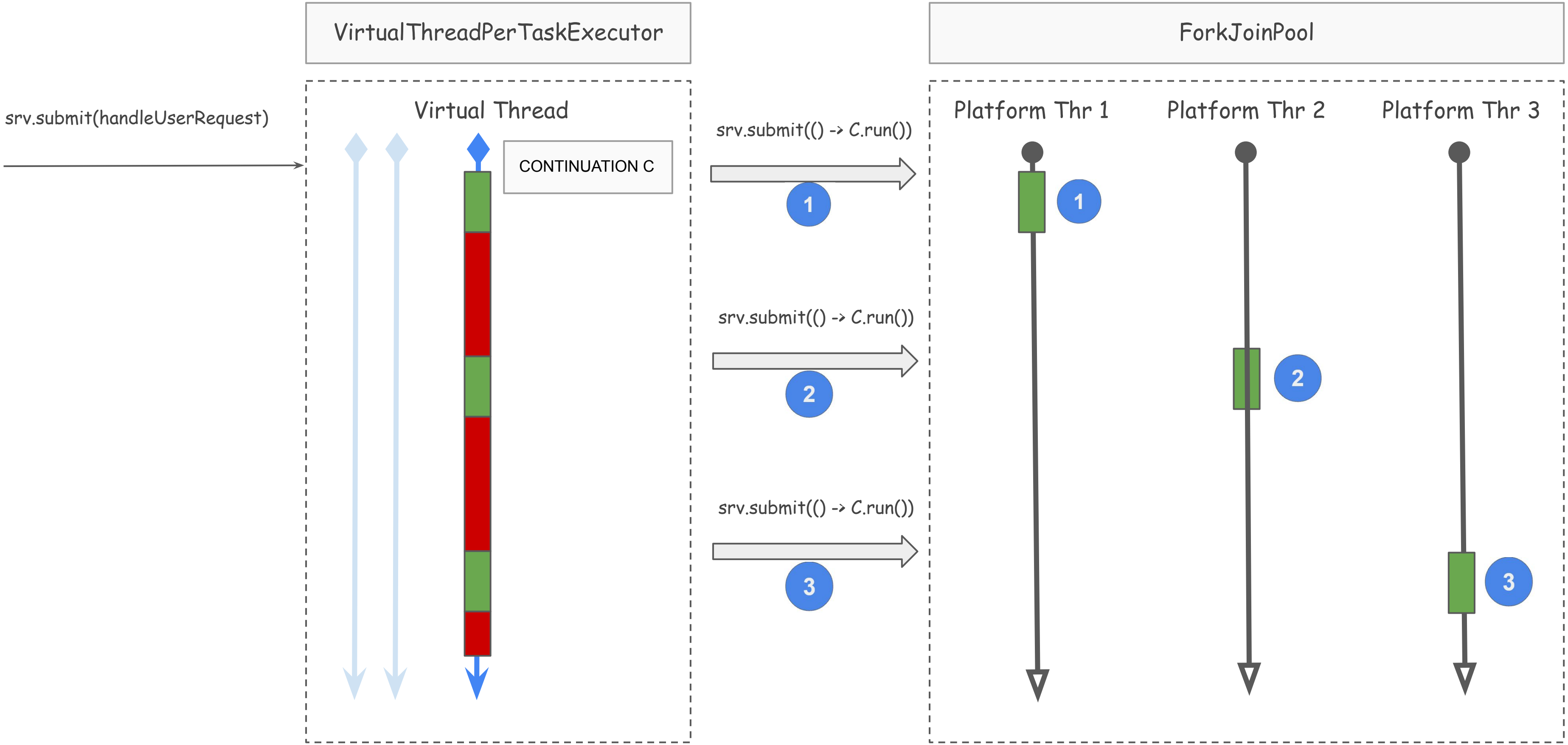
Virtual Threads using Continuations

Using Virtual Threads

```
// Pseudo code for handling User Request

// Fetch some data from DB
data1 = FetchDataFromDB(dbUrl)

// Fetch some data from a Microservice 1
data2 = FetchDataFromService1(url1)

// Process all data
combinedData = ProcessAndCombine(data1, data2)

// send data to user
SendData(combinedData)
```



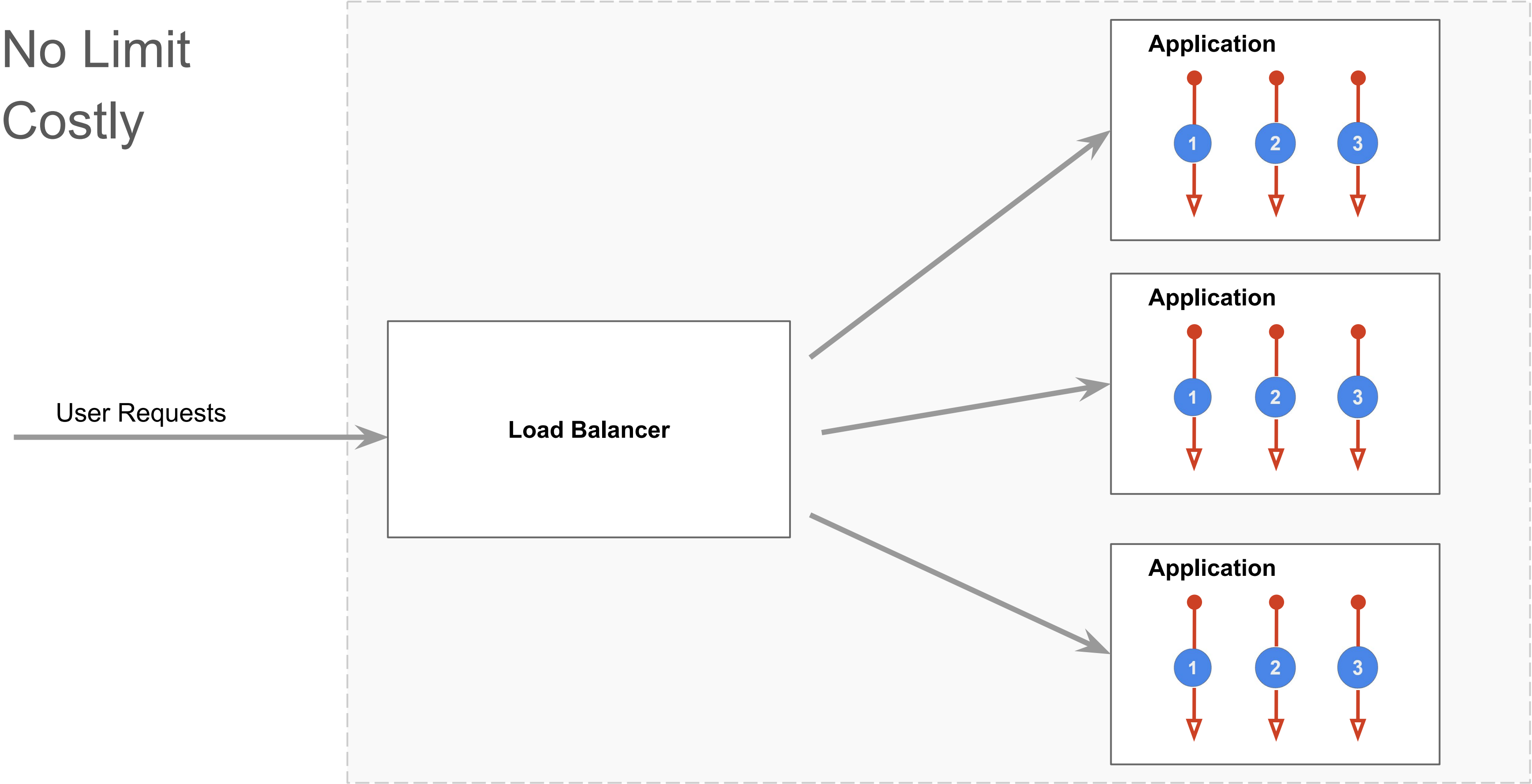

VirtualThreadPerTaskExecutor
ForkJoinPool
srv.submit(handleUserRequest)
Virtual Thread
CONTINUATION C
srv.submit(() -> C.run())
1
srv.submit(() -> C.run())
2
srv.submit(() -> C.run())
3
Platform Thr 1
Platform Thr 2
Platform Thr 3
1
2
3


VirtualThreadPerTaskExecutor
ForkJoinPool
srv.submit(handleUserRequest)
Virtual Thread
CONTINUATION C
srv.submit(() -> C.run())
1
srv.submit(() -> C.run())
2
srv.submit(() -> C.run())
3
Platform Thr 1
Platform Thr 2
Platform Thr 3
1
2
3

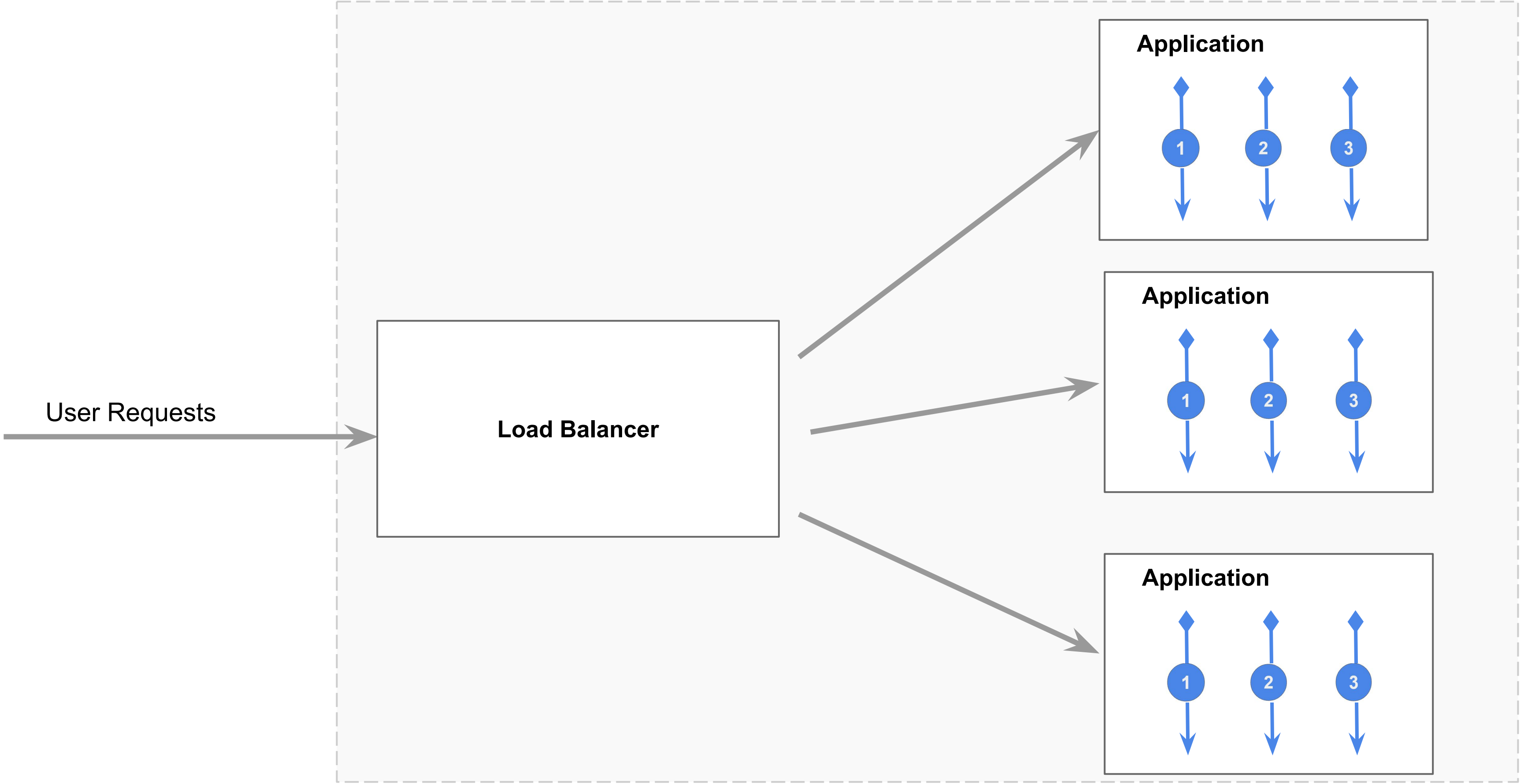
Spring Boot and Virtual Threads

Horizontal Scaling (Increase number of Application nodes)

- No Limit
- Costly



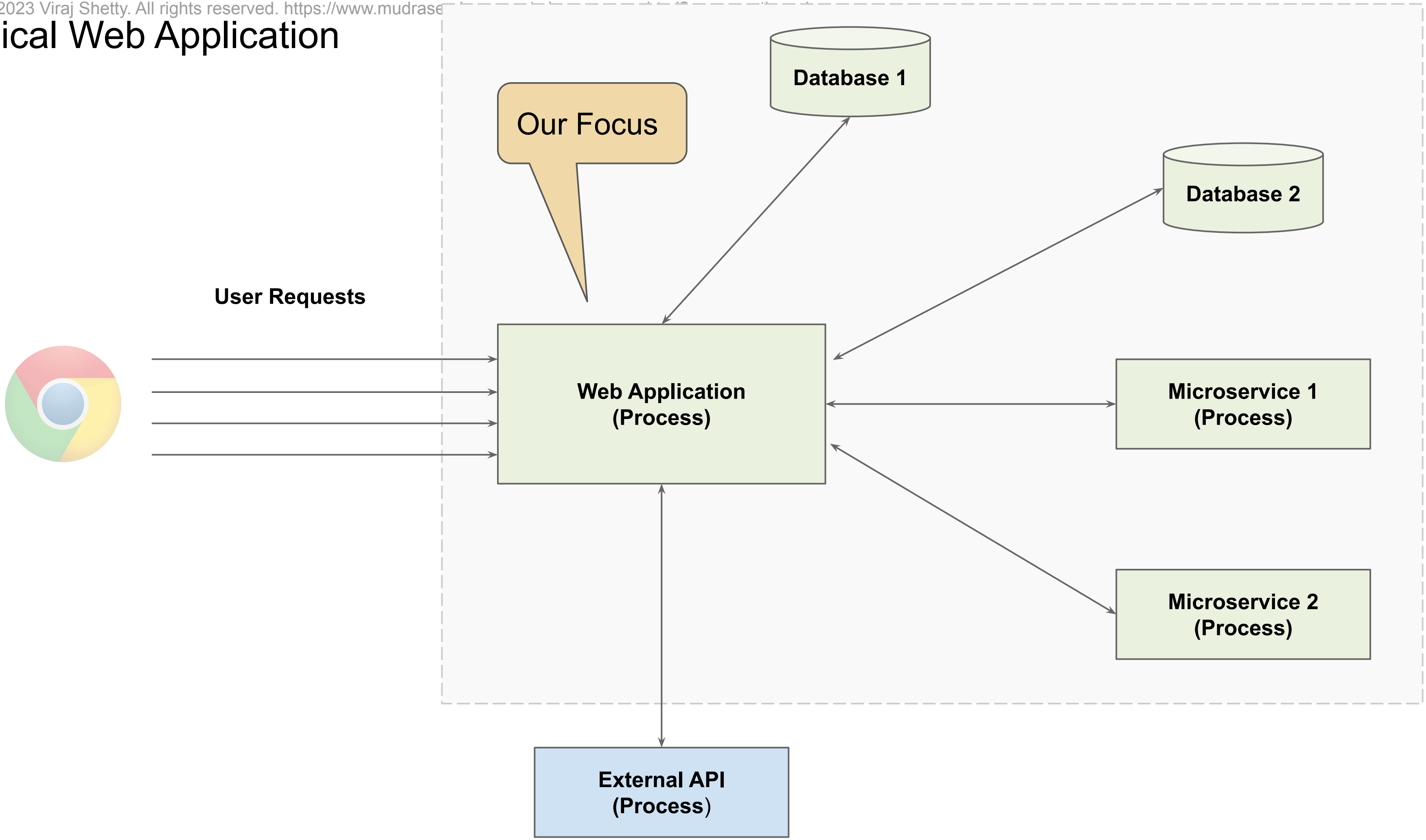
Enterprise Application using Virtual Threads - Dramatic Cost Reduction



Simple Spring Boot Application

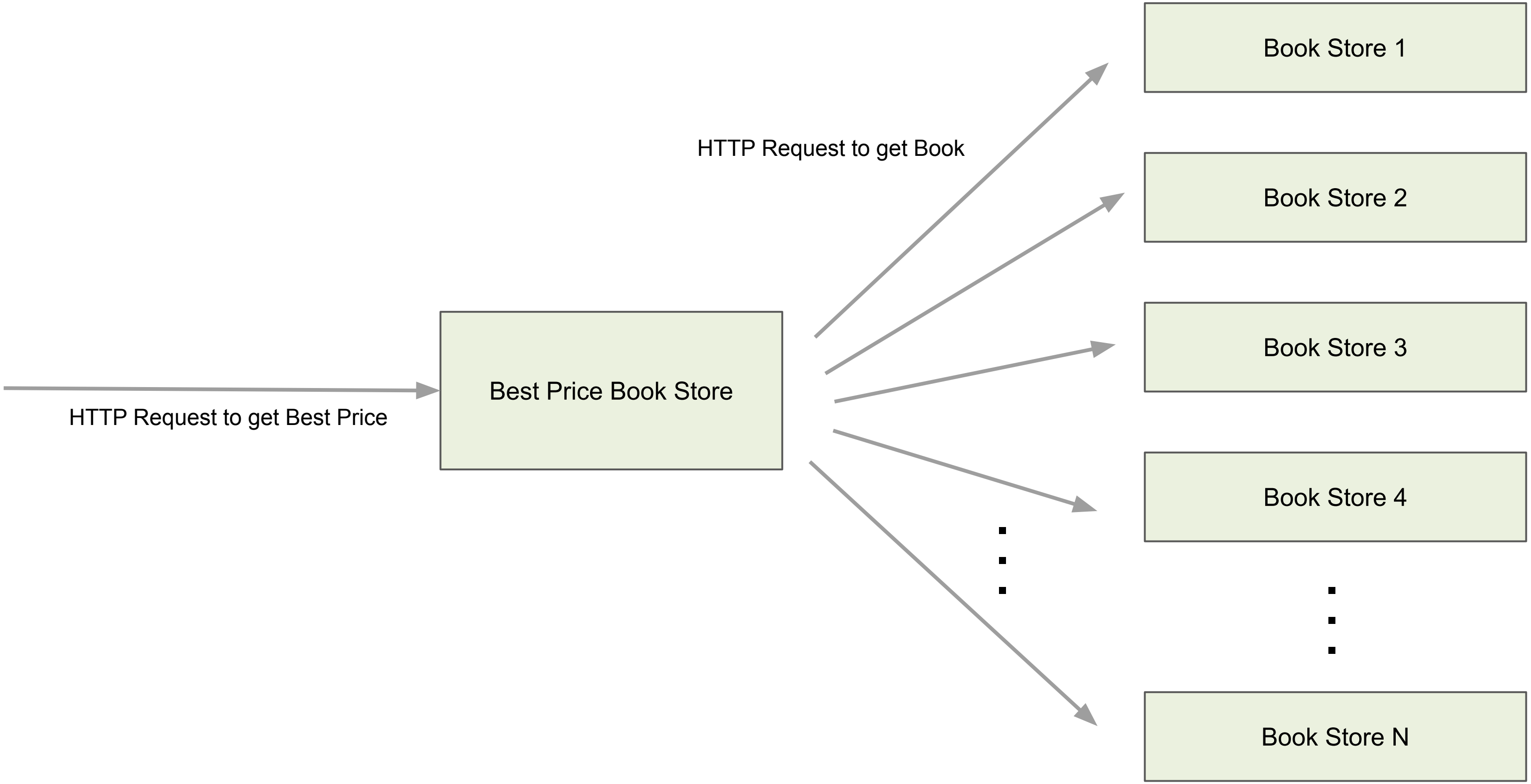
Virtual Threads in Spring Boot

Typical Web Application

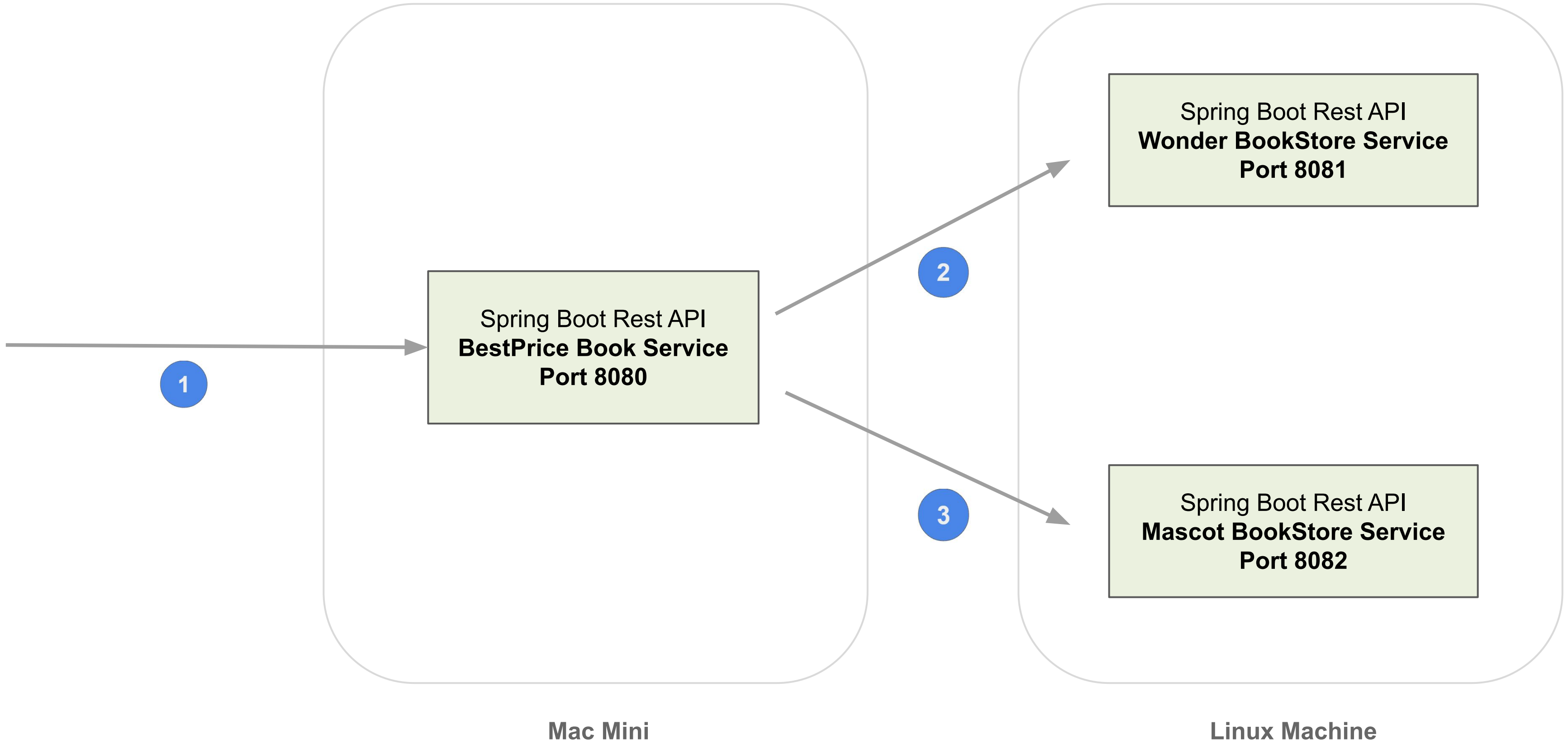


Project : Best Price Book Service

Best Price Book Service



Best Price Book Service



```
GET http://vshettys-Mini:8080/virtualstore/book?name=the+poet
```

```
{
    "callStatistics": {
        "timeMap": {
            "Mascot Book Store": 5214,
            "Best Price Store": 5217,
            "Wonder Book Store": 5214
        }
    },
    "bestPriceDeal": {
        "bookStore": "Wonder Book Store",
        "bookName": "The Poet",
        "author": "Michael Connelly",
        "cost": 15,
        "numPages": 528,
        "link": "http://wonder:8081/store/book?name=The+Poet"
    },
    "allDeals": [
        {
            "bookStore": "Wonder Book Store",
            "bookName": "The Poet",
            "author": "Michael Connelly",
            "cost": 15,
            "numPages": 528,
            "link": "http://wonder:8081/store/book?name=The+Poet"
        },
        {
            "bookStore": "Mascot Book Store",
            "bookName": "The Poet",
            "author": "Michael Connelly",
            "cost": 16,
            "numPages": 528,
            "link": "http://mascot:8082/store/book?name=The+Poet"
        }
    ]
}
```

Project : Best Price Book Store

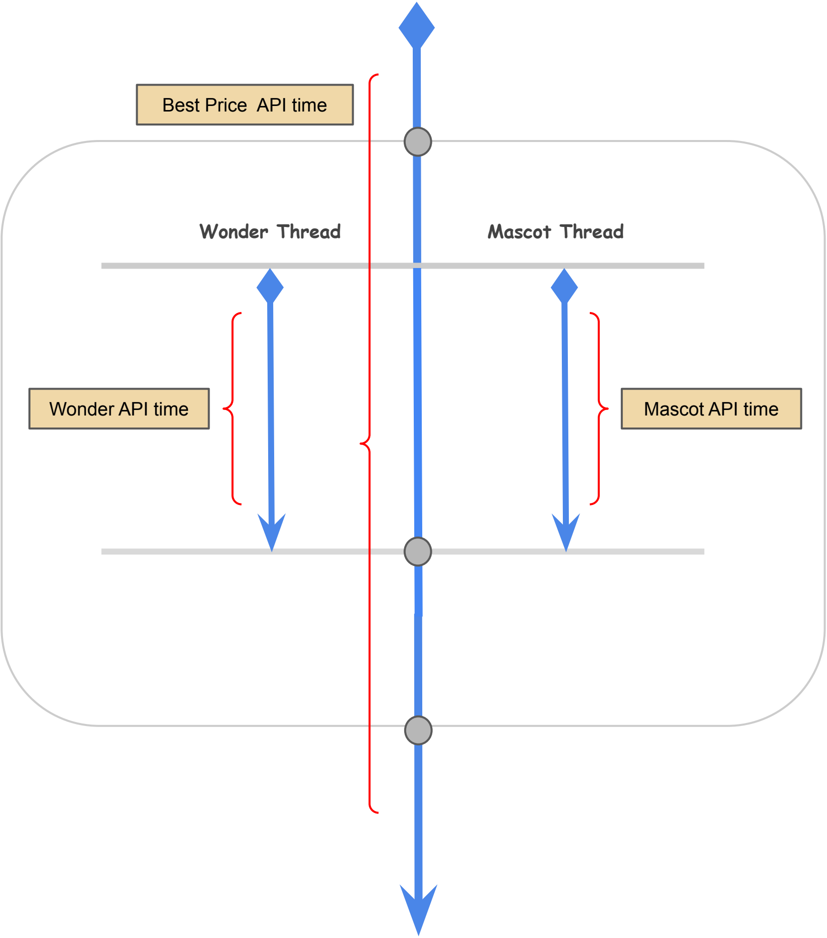
- Create REST API for the **Wonder** and **Mascot** Book Stores
  - Virtual Threads
- Test this with **Apache Benchmark (ab)** tool
- Create REST API for the **Best Price Book Store**
  - Virtual Threads
  - StructuredTaskScope
- Add **Timing** information using **Scoped Values**
- Test this with **Apache Benchmark (ab)** tool

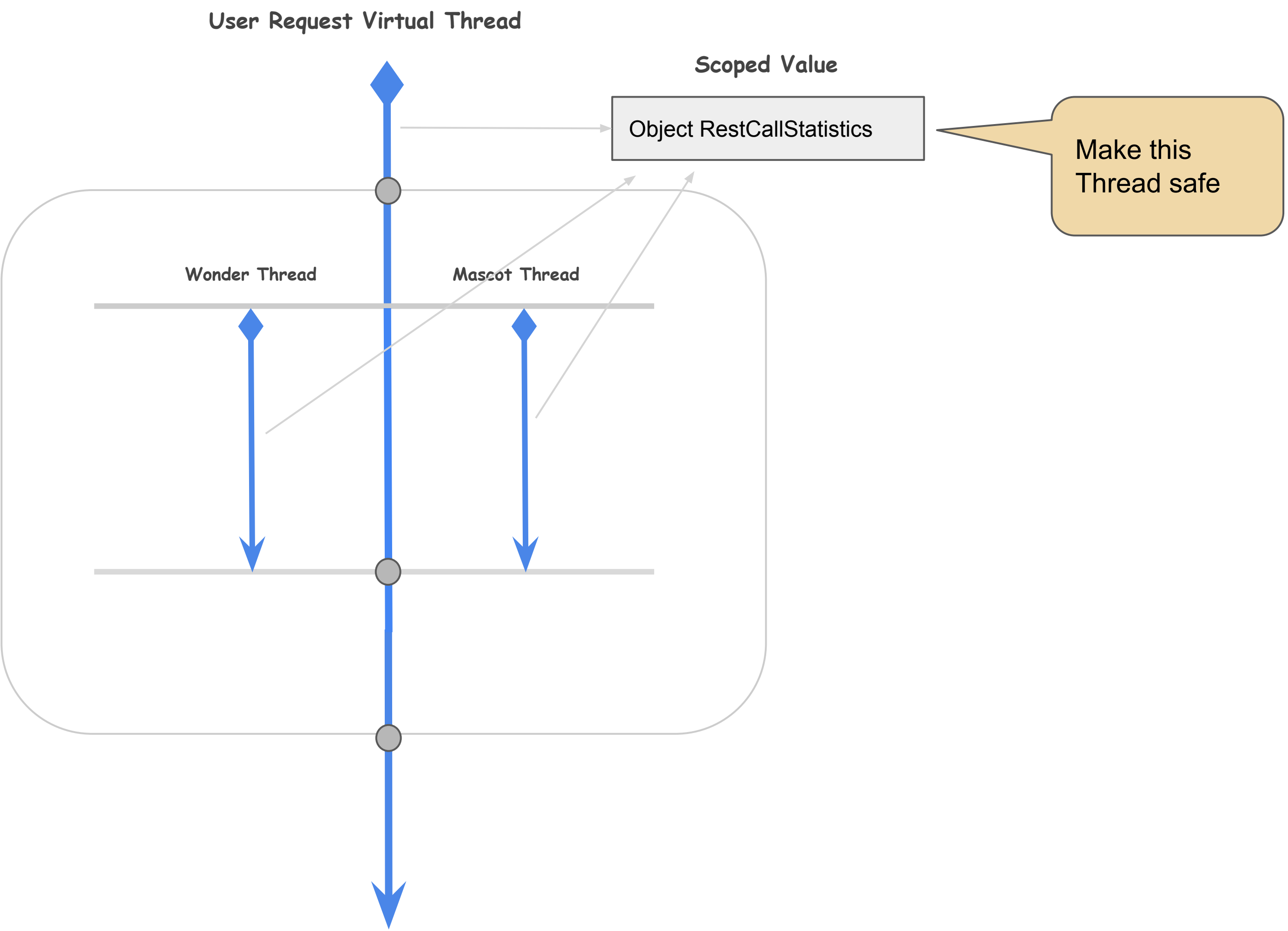
Coding : BookStore Service

Coding : BestPrice BookStore Service

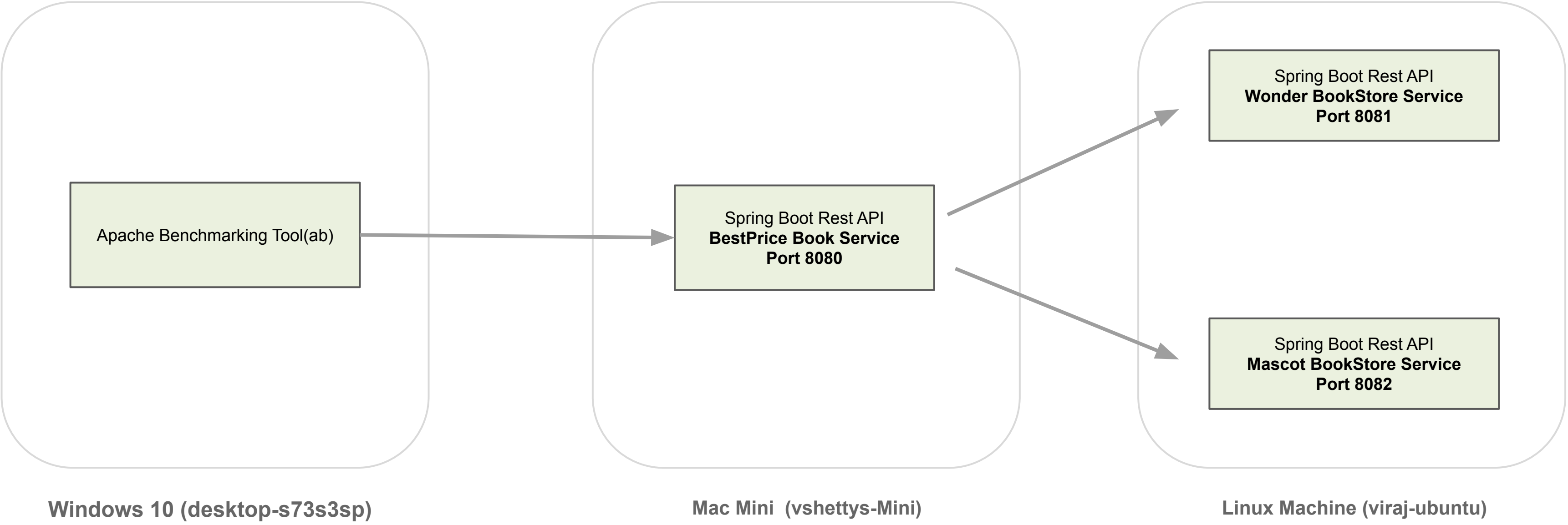
Coding : Adding Timing Statistics

User Request Virtual Thread




User Request Virtual Thread
Scoped Value
Object RestCallStatistics
Make this Thread safe
Wonder Thread
Mascot Thread

Scalability


Apache Benchmarking Tool(ab)
Spring Boot Rest API
BestPrice Book Service
Port 8080
Spring Boot Rest API
Wonder BookStore Service
Port 8081
Spring Boot Rest API
Mascot BookStore Service
Port 8082
Windows 10 (desktop-s73s3sp)
Mac Mini (vshettys-Mini)
Linux Machine (viraj-ubuntu)

- Will use Apache Benchmarking tool to create 40,000 concurrent connections
- Spring Boot set up for 50,000 max connections in application.properties

```
server.tomcat.max-connections=50000
```

- ulimit command on Linux and MacOS to set to 100,000

```
ulimit -n 100000
```

- Windows Maximum Dynamic TCP port = 50,000

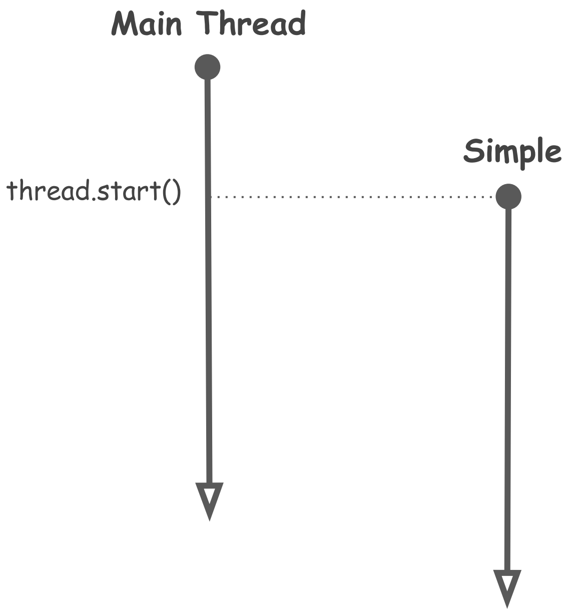
```
netsh int ipv4 set dynamicport tcp 10000 50000
```

Conclusion

Creating New Threads

Extend Thread Class

```java
// start a thread from Main Thread
Thread thread = new SimpleThread("Simple", 2);
thread.start();
```



```java
class SimpleThread extends Thread {

   private final int secs;

   SimpleThread(String name, int secs) {
       this.secs = secs;
       this.setName(name);
   }

   @Override
   public void run() {
       System.out.printf("%s : Starting Simple Thread\n",
                                                     this.getName());

       try {
           TimeUnit.SECONDS.sleep(this.secs);
       } catch (InterruptedException e) {
           System.out.println("Interrupted");
       }

       System.out.printf("%s : Ending Simple Thread\n",
                                                     this.getName());
   }

}
```

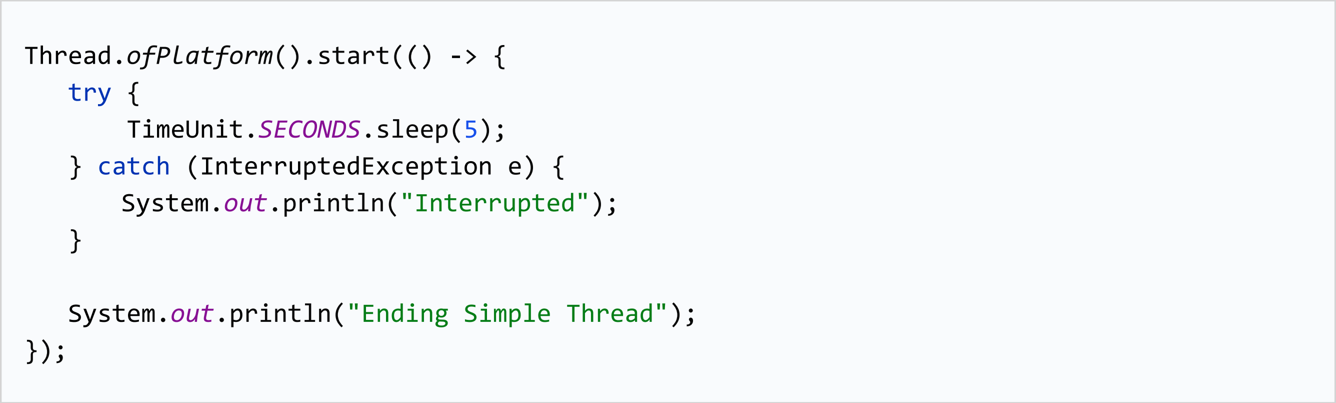
Implement Runnable Interface

```java
// start a thread from Runnable
Runnable r = new SimpleRunnable();
Thread thread = new Thread(r);
thread.start();
```

```java
// start a thread from Runnable. A more fluent way
Runnable r = new SimpleRunnable();
Thread thread =
       Thread.ofPlatform().name("Simple").daemon(true).start(r);
```

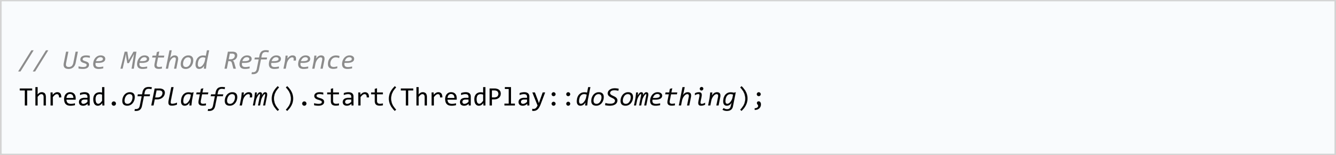
```java
class SimpleRunnable implements Runnable {

   @Override
   public void run() {

       try {
           TimeUnit.SECONDS.sleep(5);
       } catch (InterruptedException e) {
           System.out.println("Interrupted");
       }

       System.out.println("Ending Simple Thread");
   }

}
```

Using Lambda Functions

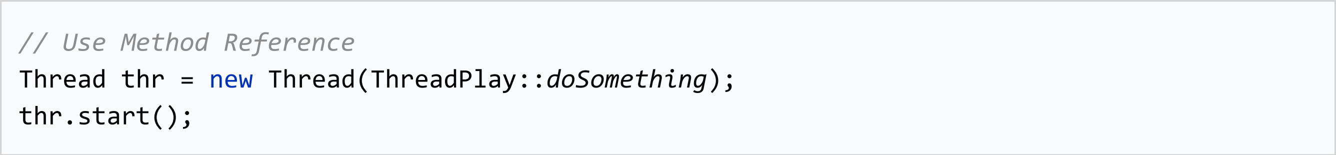


```
Thread.ofPlatform().start(() -> {
    try {
        TimeUnit.SECONDS.sleep(5);
    } catch (InterruptedException e) {
        System.out.println("Interrupted");
    }

    System.out.println("Ending Simple Thread");
});
```

Using Method Reference



```
// Use Method Reference
Thread.ofPlatform().start(ThreadPlay::doSomething);
```



```
// Use Method Reference
Thread thr = new Thread(ThreadPlay::doSomething);
thr.start();
```

Get Current Thread

```
Thread thread = Thread.currentThread();
System.out.println(thread.getName());
```

Interrupt another thread

```
thread.interrupt();
boolean interrupted = thread.isInterrupted();
```

Join

```
Thread thread = Thread.ofPlatform().start(ThreadPlay::doSomething);
thread.join()
```

sleep

```
Thread.sleep(Duration.ofSeconds(5));
```

Set Daemon status

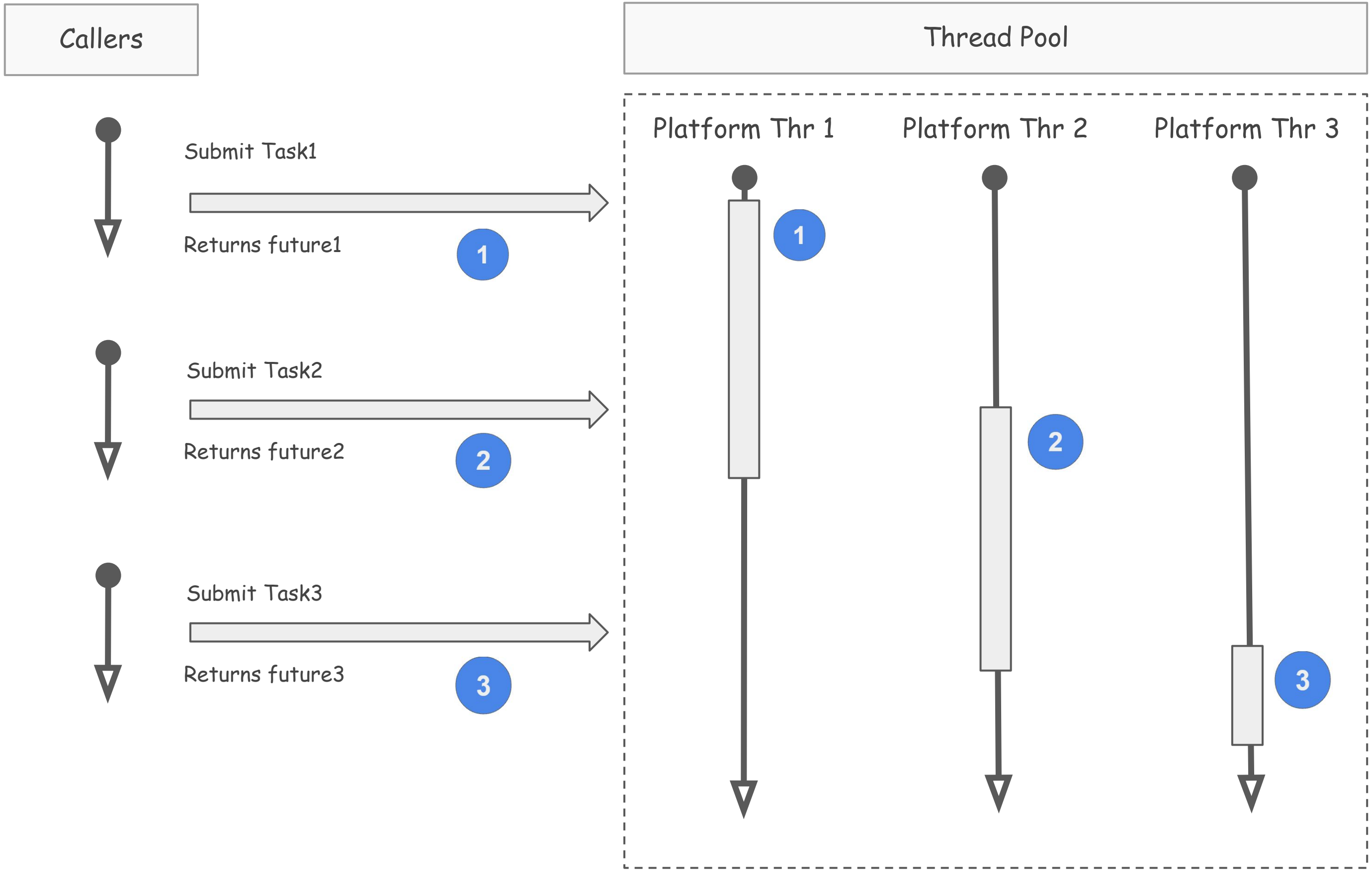
```
thread.setDaemon(true);
```

Java Futures

- Thread Pools
  - Platform Thread is an expensive Resource

- Key Abstraction is a Task
  - Runnable
  - Callable

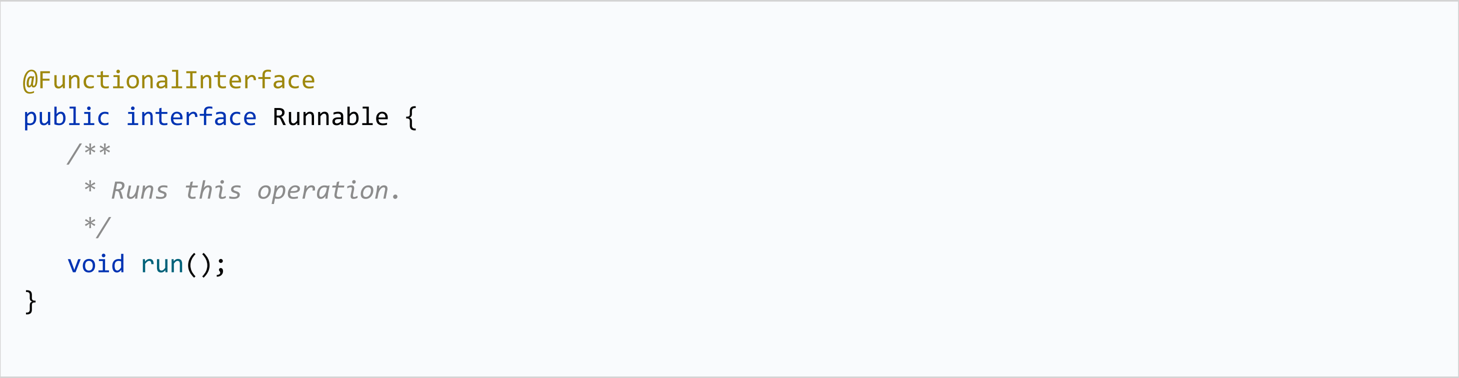
- Executor Service
  - Mostly backed by a Thread Pool
  - Separates Task from its execution Policy


Callers
Thread Pool
Submit Task1
Returns future1
1
Submit Task2
Returns future2
2
Submit Task3
Returns future3
3
Platform Thr 1
Platform Thr 2
Platform Thr 3
1
2
3

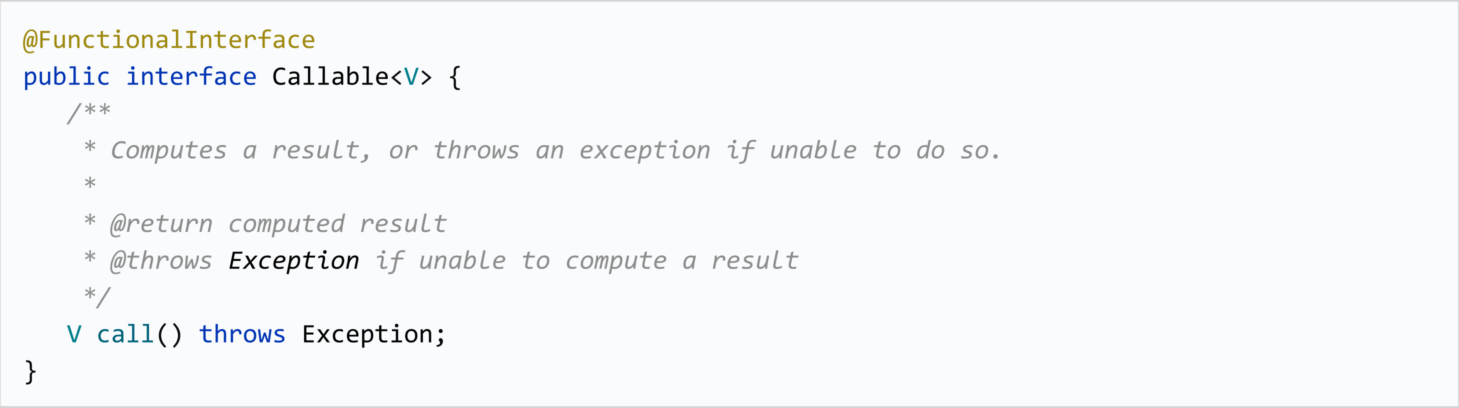
Java Futures

- Thread Pools
  - Platform Thread is an expensive Resource
- Key Abstraction is a Task
  - Runnable
  - Callable
- Executor Service
  - Mostly backed by a Thread Pool
  - Separates Task from its execution Policy

Tasks - Runnable and Callable



```java
@FunctionalInterface
public interface Runnable {
   /**
     * Runs this operation.
     */
   void run();
}
```



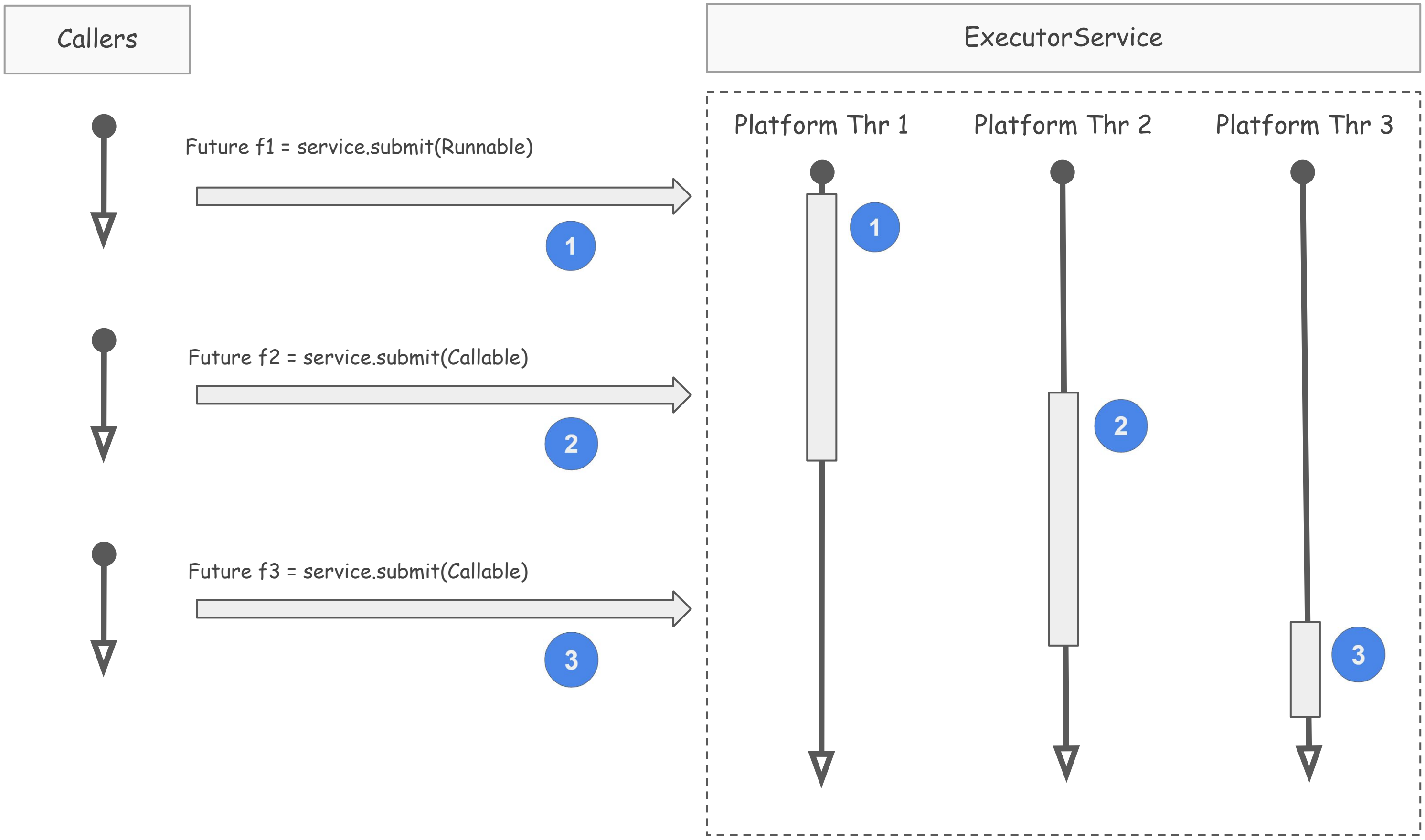
```java
@FunctionalInterface
public interface Callable<V> {
   /**
     * Computes a result, or throws an exception if unable to do so.
     *
     * @return computed result
     * @throws Exception if unable to compute a result
     */
   V call() throws Exception;
}
```

Executor Service

```
public interface ExecutorService extends Executor, AutoCloseable {

      // Only the most important methods are shown here

      // Submit a Runnable or a Callable task
      <T> Future<T> submit(Callable<T> task);
      Future<?> submit(Runnable task);

      // Orderly Shutdown. All submitted Tasks will be executed
      void shutdown();

      // Attempts to stop all executing Tasks, halts processing of waiting Tasks
      List<Runnable> shutdownNow();

      // Initiates Orderly Shutdown and waits for all tasks to finish
      default void close();

}
```

Future interface

```
public interface Future<V> {

   // Get the Result of the Task Execution. Wait till result is available
   V get() throws InterruptedException, ExecutionException;
   V get(long timeout, TimeUnit unit)
           throws InterruptedException, ExecutionException, TimeoutException;

   // Get the Result immediately. Assumes that Task is Completed
   default V resultNow();
   default Throwable exceptionNow();

   // Attempts to cancel the execution of the Task
   boolean cancel(boolean mayInterruptIfRunning);

   // Computation State : RUNNING, SUCCESS, FAILED, CANCELLED
   default java.util.concurrent.Future.State state();
   boolean isCancelled();

}
```


Callers
ExecutorService
Platform Thr 1
Platform Thr 2
Platform Thr 3
Future f1 = service.submit(Runnable)
1
Future f2 = service.submit(Callable)
2
Future f3 = service.submit(Callable)
3
1
2
3

Submitting a Runnable Task

```java
// Submit a Task to a Single Thread Executor
try(ExecutorService service
                = Executors.newSingleThreadExecutor()) {

   Future<?> future = service.submit(FuturesPlay::doSimpleTask);

   // do other tasks here

   // wait for future to complete
   future.get();

   // do something else

}
```

```java
public static void doSimpleTask() {

   System.out.printf("%s : Starting Simple Task\n",
           Thread.currentThread().getName());
   try {

       TimeUnit.SECONDS.sleep(5);

   } catch (InterruptedException e) {
       System.out.println("Task Interrupted");
   }

   System.out.printf("%s : Ending Simple Task\n",
           Thread.currentThread().getName());
}
```

Submitting a Callable Task

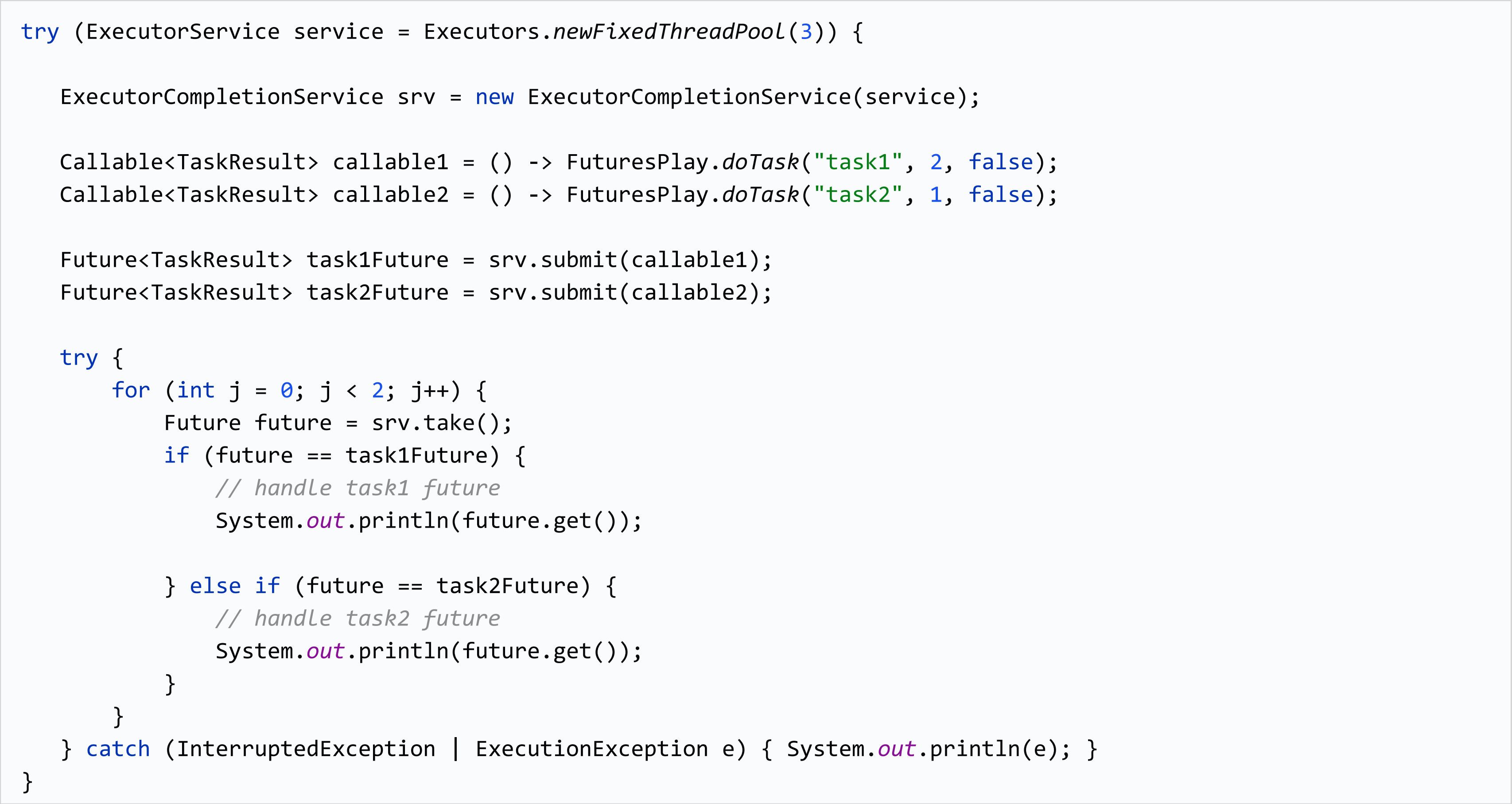
```java
try (ExecutorService service
            = Executors.newFixedThreadPool(3)) {

   Future<TaskResult> future
       = service.submit(
           () -> FuturesPlay.doTask("SimpleTask", 1, false));

   // supposed to do some other work

   try {
       TaskResult taskData = future.get();
       System.out.println(taskData);
   } catch (InterruptedException | ExecutionException e) {
       System.out.println(e);
   }

}
```

```java
public static TaskResult doTask(String name,
                                int secs, boolean fail) {

   System.out.printf("%s : Starting Task %s\n",
           Thread.currentThread().getName(), name);

   try {
       TimeUnit.SECONDS.sleep(secs);
   } catch (InterruptedException e) {
       throw new RuntimeException(e);
   }

   if (fail) {
       throw new RuntimeException("Task Failed");
   }

   System.out.printf("%s : Ending Task %s\n",
           Thread.currentThread().getName(), name);
   return new TaskResult(name, secs);
}
```

Submitting Multiple Callable Tasks

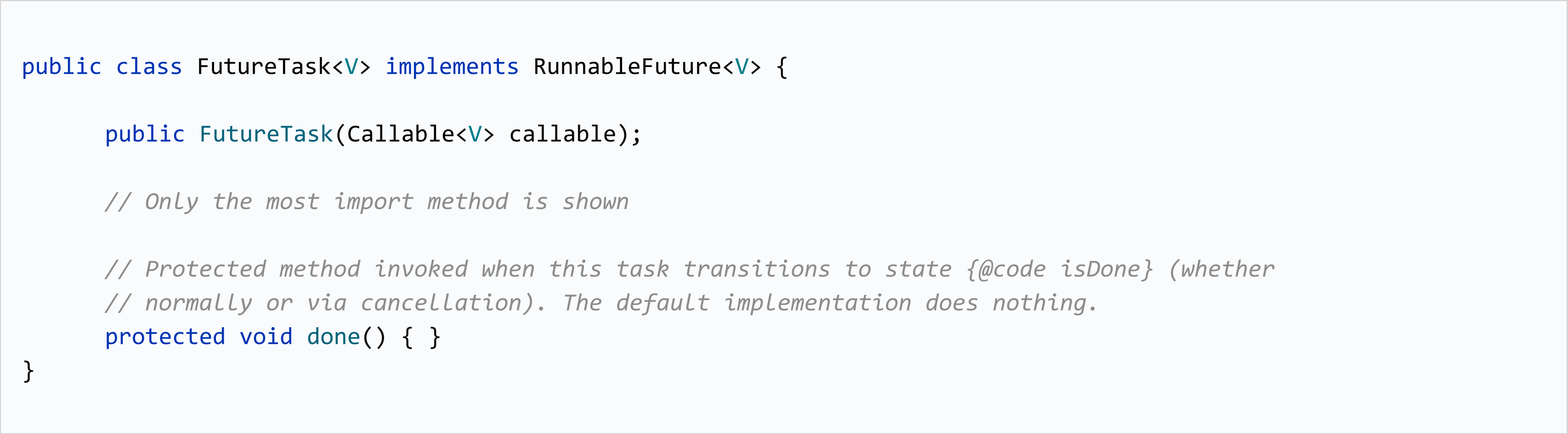
```
try (ExecutorService service = Executors.newFixedThreadPool(3)) {

   Future<TaskResult> task1Future = service.submit(() -> FuturesPlay.doTask("task1", 3, false));
   Future<TaskResult> task2Future = service.submit(() -> FuturesPlay.doTask("task2", 2, false));
   Future<TaskResult> task3Future = service.submit(() -> FuturesPlay.doTask("task3", 1, false));

   try {
       // Handle taskData1. get() will block till task1 completes
       TaskResult taskResult1 = task1Future.get();
       System.out.println(taskResult1);

       // Handle taskData2. get() will block till task2 completes
       TaskResult taskResult2 = task1Future.get();
       System.out.println(taskResult2);

       // Handle taskData3. get() will block till task3 completes
       TaskResult taskResult3 = task1Future.get();
       System.out.println(taskResult3);

   } catch (InterruptedException | ExecutionException e) {
       System.out.println(e);
   }
}
```

ExecutorCompletionService

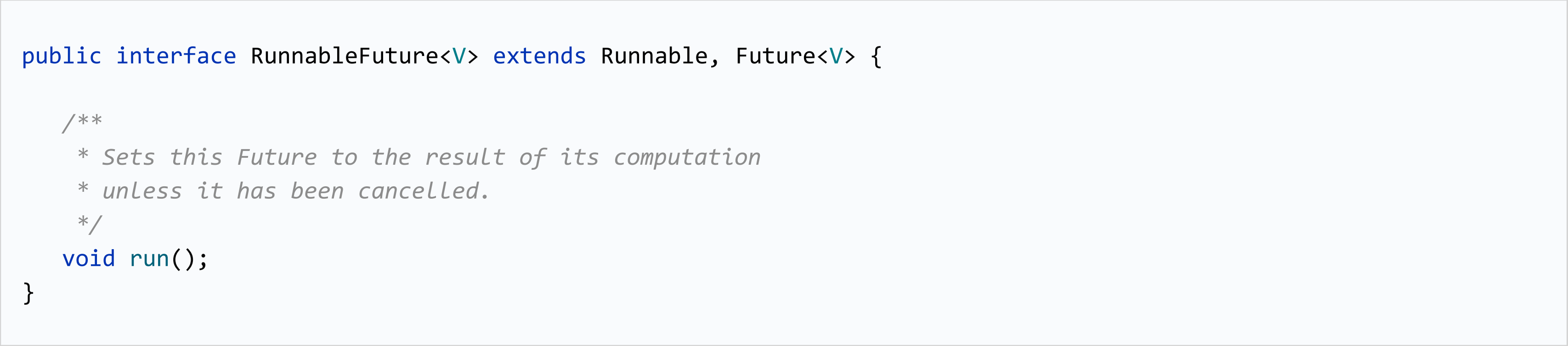


```
try (ExecutorService service = Executors.newFixedThreadPool(3)) {

   ExecutorCompletionService srv = new ExecutorCompletionService(service);

   Callable<TaskResult> callable1 = () -> FuturesPlay.doTask("task1", 2, false);
   Callable<TaskResult> callable2 = () -> FuturesPlay.doTask("task2", 1, false);

   Future<TaskResult> task1Future = srv.submit(callable1);
   Future<TaskResult> task2Future = srv.submit(callable2);

   try {
       for (int j = 0; j < 2; j++) {
           Future future = srv.take();
           if (future == task1Future) {
               // handle task1 future
               System.out.println(future.get());

           } else if (future == task2Future) {
               // handle task2 future
               System.out.println(future.get());
           }
       }
   } catch (InterruptedException | ExecutionException e) { System.out.println(e); }
}
```

FutureTask



```java
public class FutureTask<V> implements RunnableFuture<V> {

      public FutureTask(Callable<V> callable);

      // Only the most import method is shown

      // Protected method invoked when this task transitions to state {@code isDone} (whether
      // normally or via cancellation). The default implementation does nothing.
      protected void done() { }
}
```



```java
public interface RunnableFuture<V> extends Runnable, Future<V> {

   /**
    * Sets this Future to the result of its computation
    * unless it has been cancelled.
    */
   void run();
}
```

FutureTask

```java
OurFutureTask<TaskResult> task1 = new OurFutureTask<>(
             () -> FuturesPlay.doTask("task1", 1, true));
OurFutureTask<TaskResult> task2 = new OurFutureTask<>(
             () -> FuturesPlay.doTask("task2", 4, false));

try (var service = Executors.newCachedThreadPool()) {

   Future<?> future1 = service.submit(task1);
   Future<?> future2 = service.submit(task2);

   // wait for both to complete
   future1.get();
   future2.get();

   // do other tasks

} catch (Exception e) {
   // handle exceptions
}

System.out.println("Completed all");
```

```java
class OurFutureTask<V> extends FutureTask<V> {

   public OurFutureTask(Callable<V> callable) {
       super(callable);
   }

   @Override
   protected void done() {
       try {
           System.out.println("Done Task1..." + get());
       } catch (Exception e) {
           System.out.println("Exception Task1..."
                                       + exceptionNow());
       }
  }

}
```

Java Futures Limitations

- Cannot create an Asynchronous Pipeline
- Cannot Complete a Future
- Limited Features

Imperative Style - Pseudo Code (Blocking)

```
// Pseudo code for handling User Request

// Fetch some data from DB
data1 = FetchDataFromDB(dbUrl)

// Fetch some data from a Microservice 1
data2 = FetchDataFromService1(url1)

// Process all data
combinedData = ProcessAndCombine(data1, data2)

// send data to user
SendData(combinedData)
```

Reactive Style - Pseudo Code

```
// Reactive Pseudo code for handling User Request
// The user thread does not block


Pipeline
      .Run(FetchDataFromDB(dbUrl))
      .Run(FetchDataFromService1(url1))
      .Combine(dataResult, serviceResult)
      .SendData(combinedData)


// Method exits before Database and Service operations
// are completed
```

Submitting a Callable Task

```
try (ExecutorService service
            = Executors.newFixedThreadPool(3)) {

   Future<TaskResult> future
       = service.submit(
           () -> FuturesPlay.doTask("SimpleTask", 1, false));

   // supposed to do some other work

   try {
       TaskResult taskData = future.get();
       System.out.println(taskData);
   } catch (InterruptedException | ExecutionException e) {
       System.out.println(e);
   }

}
```

```
public static TaskResult doTask(String name,
                          int secs, boolean fail) {

   System.out.printf("%s : Starting Task %s\n",
           Thread.currentThread().getName(), name);

   try {
       TimeUnit.SECONDS.sleep(secs);
   } catch (InterruptedException e) {
       throw new RuntimeException(e);
   }

   if (fail) {
       throw new RuntimeException("Task Failed");
   }

   System.out.printf("%s : Ending Task %s\n",
           Thread.currentThread().getName(), name);
   return new TaskResult(name, secs);
}
```

Java Futures Limitations

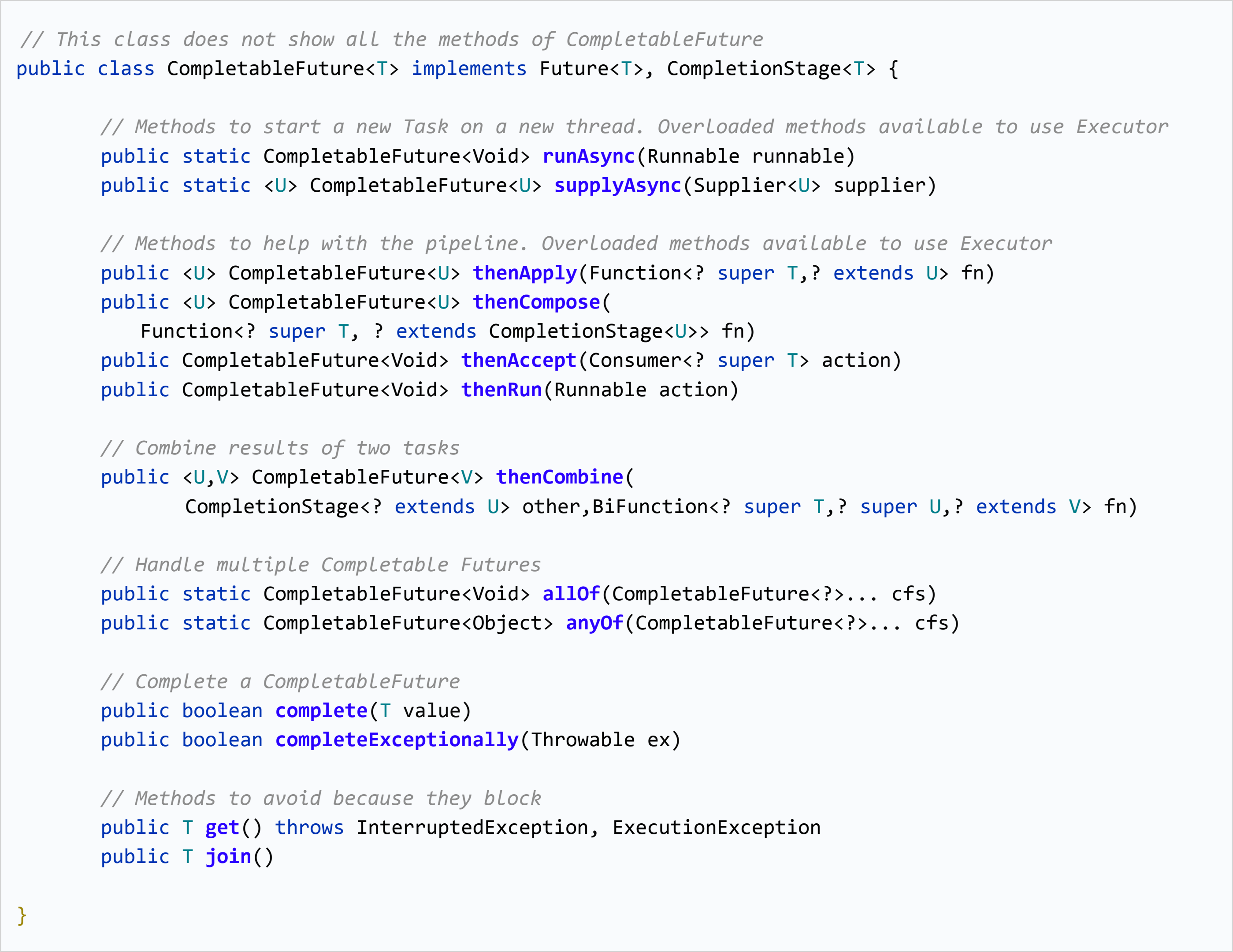
- Cannot create an Asynchronous Pipeline
- Cannot Complete a Future
- Limited Features

Java CompletableFuture

Completable Future Pipeline

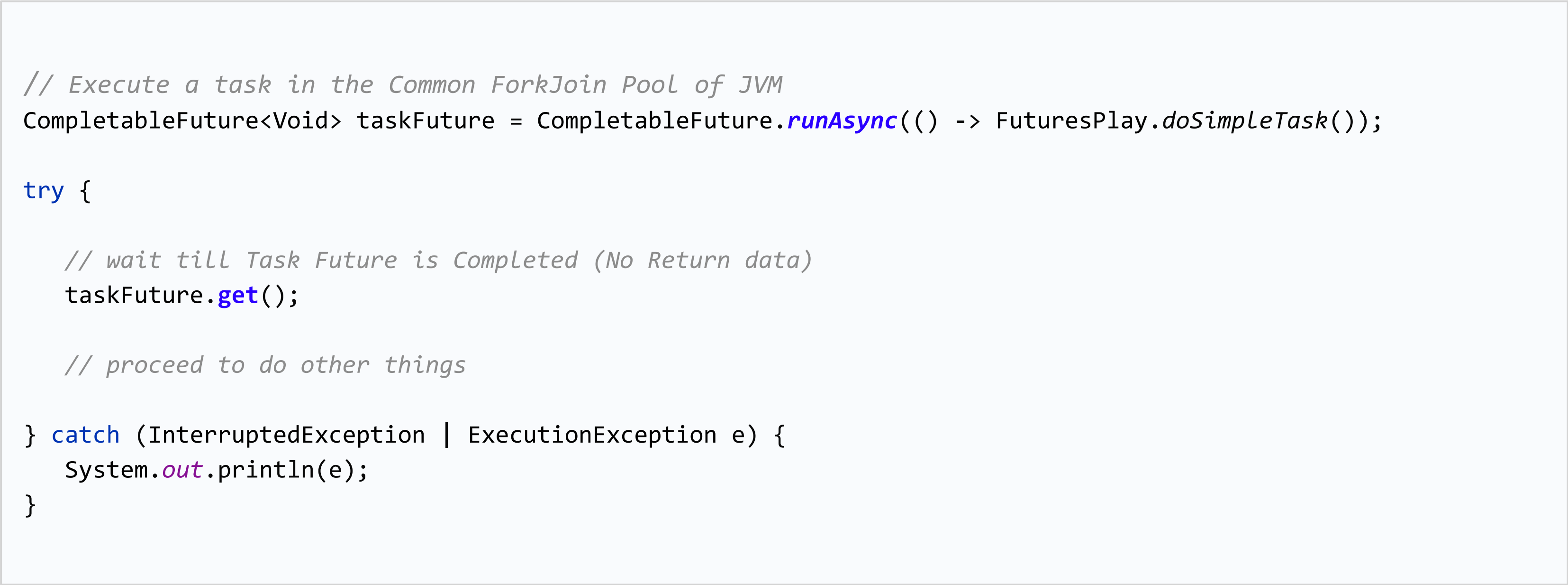
```
// Tasks to execute in parallel
Supplier<TaskResult> task1 = () -> FuturesPlay.doTask("task1", 3, false);
Supplier<TaskResult> task2 = () -> FuturesPlay.doTask("task2", 5, false);

// supplyAsync will start the task in a separate thread
// thenCombine will combine the results of Task1 and Task2
// thenApply will operate on this combined Result and generate new Result
// thenAccept will handle the final Result
CompletableFuture.supplyAsync(task1)
       .thenCombine(
           CompletableFuture.supplyAsync(task2),
           (result1, result2) -> String.format("Combined (%s : %s)", result1.taskName(), result2.taskName()))
       .thenApply(data -> data + " :: Handled Apply")
       .thenAccept(data -> {
           System.out.println(data + " :: Handled Accept");
       })

doSomethingElse();
```

Result ⇒ `Combined (task1 : task2) :: Handled Apply :: Handled Accept`



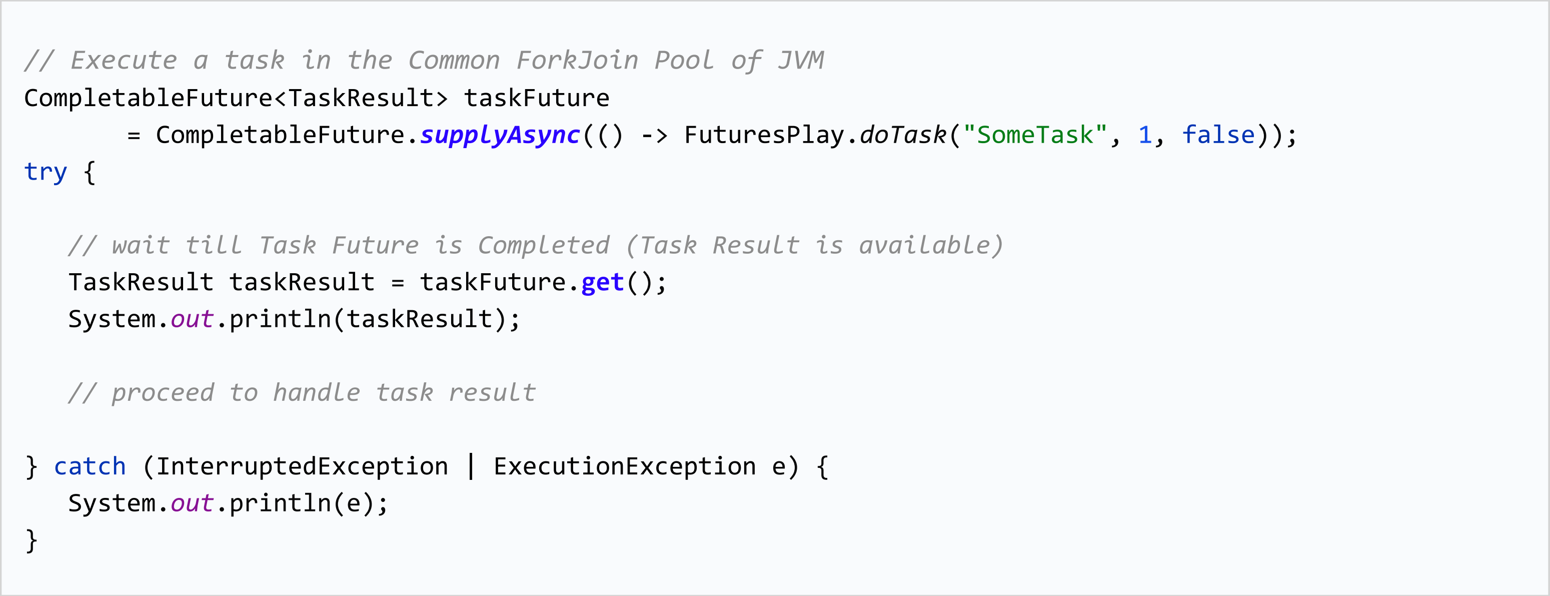
```
// This class does not show all the methods of CompletableFuture
public class CompletableFuture<T> implements Future<T>, CompletionStage<T> {

        // Methods to start a new Task on a new thread. Overloaded methods available to use Executor
        public static CompletableFuture<Void> runAsync(Runnable runnable)
        public static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier)

        // Methods to help with the pipeline. Overloaded methods available to use Executor
        public <U> CompletableFuture<U> thenApply(Function<? super T,? extends U> fn)
        public <U> CompletableFuture<U> thenCompose(
           Function<? super T, ? extends CompletionStage<U>> fn)
        public CompletableFuture<Void> thenAccept(Consumer<? super T> action)
        public CompletableFuture<Void> thenRun(Runnable action)

        // Combine results of two tasks
        public <U,V> CompletableFuture<V> thenCombine(
               CompletionStage<? extends U> other,BiFunction<? super T,? super U,? extends V> fn)

        // Handle multiple Completable Futures
        public static CompletableFuture<Void> allOf(CompletableFuture<?>... cfs)
        public static CompletableFuture<Object> anyOf(CompletableFuture<?>... cfs)

        // Complete a CompletableFuture
        public boolean complete(T value)
        public boolean completeExceptionally(Throwable ex)

        // Methods to avoid because they block
        public T get() throws InterruptedException, ExecutionException
        public T join()

}
```

runAsync(..)



```java
// Execute a task in the Common ForkJoin Pool of JVM
CompletableFuture<Void> taskFuture = CompletableFuture.runAsync(() -> FuturesPlay.doSimpleTask());

try {

   // wait till Task Future is Completed (No Return data)
   taskFuture.get();

   // proceed to do other things

} catch (InterruptedException | ExecutionException e) {
   System.out.println(e);
}
```

supplyAsync(..)



```java
// Execute a task in the Common ForkJoin Pool of JVM
CompletableFuture<TaskResult> taskFuture
       = CompletableFuture.supplyAsync(() -> FuturesPlay.doTask("SomeTask", 1, false));
try {

   // wait till Task Future is Completed (Task Result is available)
   TaskResult taskResult = taskFuture.get();
   System.out.println(taskResult);

   // proceed to handle task result

} catch (InterruptedException | ExecutionException e) {
   System.out.println(e);
}
```

Create a Pipeline

```
// Execute a task in common pool
// then Apply a function
// then Accept the result which will be consumed by Consumer
CompletableFuture pipeline =
   CompletableFuture.supplyAsync(() -> FuturesPlay.doTask("SomeTask", 3, false))
           .thenApply(taskResult -> taskResult.secs())
           .thenAccept(time -> {
               System.out.println(time);
           });
```

```
Result ⇒ 3
```

Pipeline with multiple thenApply()

```
// Execute a task in common pool
// then Apply a function
// then Accept the result which will be consumed by Consumer
CompletableFuture pipeline =
   CompletableFuture.supplyAsync(() -> FuturesPlay.doTask("SomeTask", 3, false))
           .thenApply(taskResult -> taskResult.secs())
           .thenApply(time -> time * 1000)
           .thenAccept(time -> {
               System.out.println(time);
           });
```

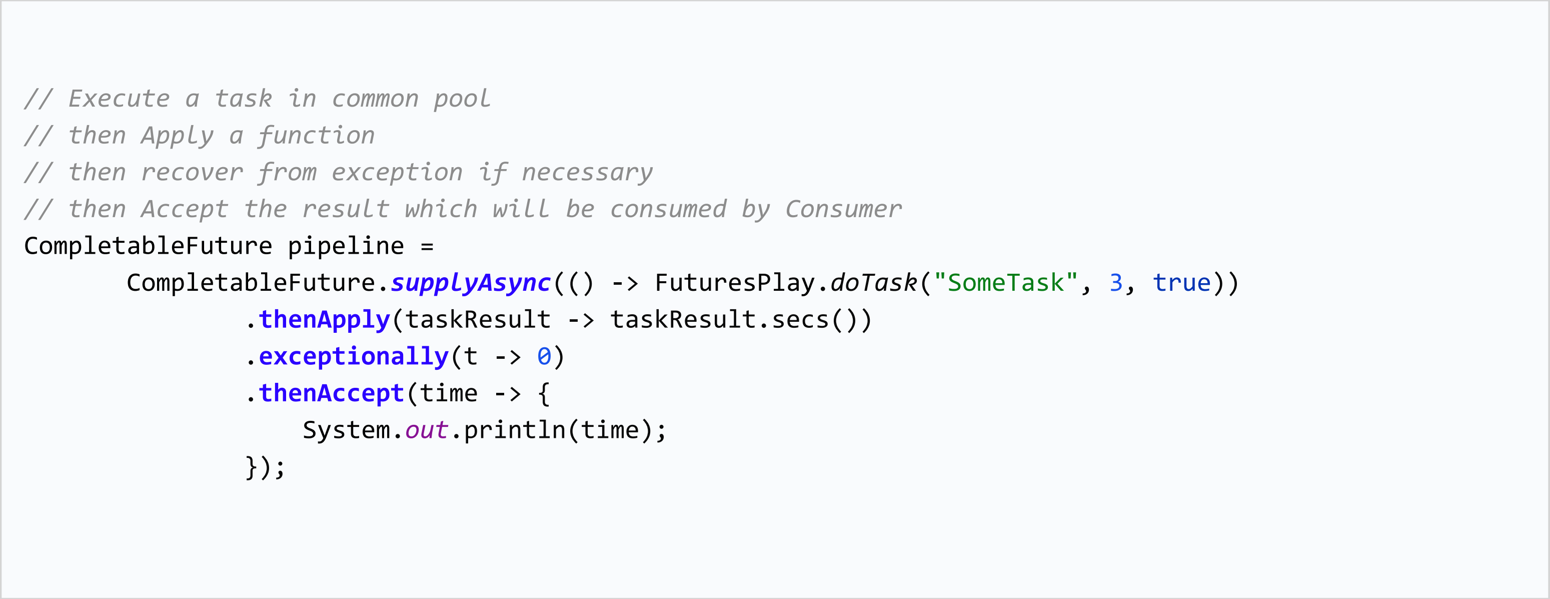
```
Result ⇒ 3000
```

Exception Recovery - exceptionally()

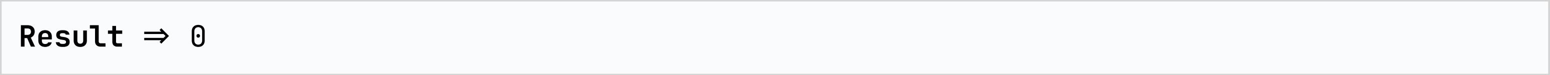
```java
// Execute a task in common pool
// then recover from exception if necessary
// then Apply a function
// then Accept the result which will be consumed by Consumer
CompletableFuture pipeline =
   CompletableFuture.supplyAsync(() -> FuturesPlay.doTask("SomeTask", 3, true))
           .exceptionally(t -> new TaskResult("SomeTask", 0))
           .thenApply(taskResult -> taskResult.secs())
           .thenAccept(time -> {
               System.out.println(time);
           });
```

```
Result ⇒ 0
```

Exception Recovery - exceptionally()



```java
// Execute a task in common pool
// then Apply a function
// then recover from exception if necessary
// then Accept the result which will be consumed by Consumer
CompletableFuture pipeline =
        CompletableFuture.supplyAsync(() -> FuturesPlay.doTask("SomeTask", 3, true))
                 .thenApply(taskResult -> taskResult.secs())
                 .exceptionally(t -> 0)
                 .thenAccept(time -> {
                     System.out.println(time);
                 });
```



```
Result ⇒ 0
```

thenCompose(..)

```java
// Execute a task in common pool
// thenCompose handles function which returns a CompletableStage<Output>
CompletableFuture pipeline =
   CompletableFuture.supplyAsync(() -> FuturesPlay.doTask("task", 3, false))
           .thenCompose(taskResult -> CompletableFuturesPlay.handleTaskResult(taskResult))
           .thenApply(data -> data + " :: Handled Apply")
           .thenAccept(data -> {
               System.out.println(data + ":: Handled Accept");
           });


private static CompletableFuture<String> handleTaskResult(TaskResult taskResult) {
   return CompletableFuture.supplyAsync(() -> {
       return taskResult + " :: Handled Compose";
   });
}
```

```
Result ⇒ TaskResult[taskName=task, secs=3] :: Handled Compose :: Handled Apply:: Handled Accept
```

thenCombine(..)

```java
// Tasks to execute asynchronously and in parallel
Supplier<TaskResult> task1 = () -> FuturesPlay.doTask("task1", 3, false);
Supplier<TaskResult> task2 = () -> FuturesPlay.doTask("task2", 5, false);

// thenCombine will combine the results of Task1 and Task2
// thenApply will operate on this combined Result
CompletableFuture pipeline =
    CompletableFuture.supplyAsync(task1)
            .thenCombine(
                    CompletableFuture.supplyAsync(task2),
                    (result1, result2) -> fuze(result1.taskName(), result2.taskName()))
            .thenApply(data -> data + " :: Handled Apply")
            .thenAccept(data -> {
                System.out.println(data + " :: Handled Accept");
            });


private static String fuze(String s1, String s2) {
   return String.format("Combined (%s : %s)", s1, s2);
}
```

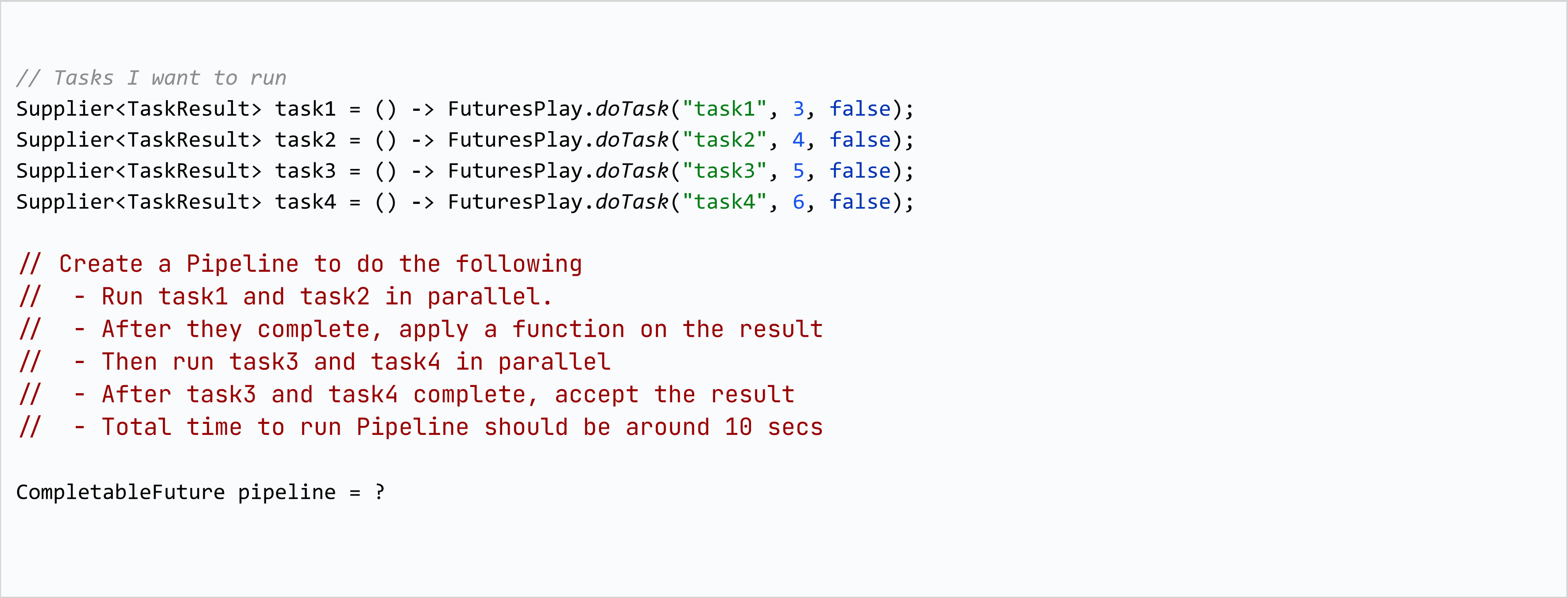
```
Result ⇒ Combined (task1 : task2) :: Handled Apply :: Handled Accept
```

Combining many Asynchronous Tasks

```java
// Tasks we want to run in parallel
Supplier<TaskResult> task1 = () -> FuturesPlay.doTask("task1", 3, false);
Supplier<TaskResult> task2 = () -> FuturesPlay.doTask("task2", 4, false);
Supplier<TaskResult> task3 = () -> FuturesPlay.doTask("task3", 5, false);
Supplier<TaskResult> task4 = () -> FuturesPlay.doTask("task4", 6, false);

// Let's run all of them in parallel.
var future1 = CompletableFuture.supplyAsync(task1);
var future2 = CompletableFuture.supplyAsync(task2);
var future3 = CompletableFuture.supplyAsync(task3);
var future4 = CompletableFuture.supplyAsync(task4);

// Now chain the task executions
CompletableFuture pipeline =
   future1.thenCombine(future2, (result1, result2) -> fuze(result1.taskName(),result2.taskName()))
          .thenCombine(future3, (s, taskResult) -> fuze(s,taskResult.taskName()))
          .thenCombine(future4, (s, taskResult) -> fuze(s,taskResult.taskName()))
          .thenApply(data -> data + " :: Handled Apply")
          .thenAccept(data -> {
              System.out.println(data + " :: Handled Accept");
          });
```

```
Result ⇒ Combined (Combined (Combined (task1 : task2) : task3) : task4) :: Handled Apply :: Handled Accept
```

Problem : Create a Pipeline



```
// Tasks I want to run
Supplier<TaskResult> task1 = () -> FuturesPlay.doTask("task1", 3, false);
Supplier<TaskResult> task2 = () -> FuturesPlay.doTask("task2", 4, false);
Supplier<TaskResult> task3 = () -> FuturesPlay.doTask("task3", 5, false);
Supplier<TaskResult> task4 = () -> FuturesPlay.doTask("task4", 6, false);

// Create a Pipeline to do the following
//   - Run task1 and task2 in parallel.
//   - After they complete, apply a function on the result
//   - Then run task3 and task4 in parallel
//   - After task3 and task4 complete, accept the result
//   - Total time to run Pipeline should be around 10 secs

CompletableFuture pipeline = ?
```

Cop

```
// Tasks I want to run
Supplier<TaskResult> task1 = () -> FuturesPlay.doTask("task1", 3, false);
Supplier<TaskResult> task2 = () -> FuturesPlay.doTask("task2", 4, false);
Supplier<TaskResult> task3 = () -> FuturesPlay.doTask("task3", 5, false);
Supplier<TaskResult> task4 = () -> FuturesPlay.doTask("task4", 6, false);

// Lets run task1 and task 2 in parallel
var future1 = CompletableFuture.supplyAsync(task1);
var future2 = CompletableFuture.supplyAsync(task2);
CompletableFuture pipeline =
   future1.thenCombine(future2, (result1, result2) -> fuze(result1.taskName(), result2.taskName()))
          .thenApply(s -> s + " :: Glue")
          .thenCompose(s -> {

              // Let's run task 3 and task 4 in parallel.
              // Note we do not start the tasks until tasks 1 and 2 are completed
              var future3 = CompletableFuture.supplyAsync(task3);
              var future4 = CompletableFuture.supplyAsync(task4);
              return future3.thenCombine(
                      future4, (result1, result2) -> s + " :: " + fuze(result1.taskName(), result2.taskName()));
          })
          .thenAccept(data -> {
              System.out.println(data + " :: Handled Accept");
          });
```

```
Result ⇒ Combined (task1 : task2) :: Glue :: Combined (task3 : task4) :: Handled Accept
```

CompletableFuture and Threads

thread-x

```
// Execute a task in common pool
// then Apply a function
// then Accept the result which will be consumed by Consumer
CompletableFuture pipeline =
   CompletableFuture.supplyAsync(() -> FuturesPlay.doTask("SomeTask", 3, false))
           .thenApply(taskResult -> taskResult.secs())
           .thenAccept(time -> {
               System.out.println(time);
           });
```

common.thread-0

common.thread-0

common.thread-0

```
Result ⇒ 3
```

CompletableFuture and Threads

thread-x

```
// Execute a task in common pool
// then Apply a function
// then Accept the result which will be consumed by Consumer
CompletableFuture<TaskResult> pipeline =
       CompletableFuture.supplyAsync(() -> FuturesPlay.doTask("SomeTask", 3, false));  common.thread-0

try {
   TimeUnit.SECONDS.sleep(5);
} catch (InterruptedException e) {
   throw new RuntimeException(e);
}

pipeline.thenApply(taskResult -> taskResult.secs())  thread-x
        .thenAccept(time -> {  thread-x
            System.out.println(time);
        });
```

```
Result ⇒ 3
```

thenApplyAsync(), thenAcceptAsync()

thread-x

```
// Execute a task in common pool
// then Apply a function on another executor service thread
// then Accept the result on another executor service thread
CompletableFuture pipeline =
        CompletableFuture.supplyAsync(() -> FuturesPlay.doTask("SomeTask", 3, false))
                .thenApplyAsync(taskResult -> taskResult.secs())
                .thenAcceptAsync(time -> {
                    System.out.println(time);
                });
```

common.thread-0

common.thread-1

common.thread-2

```
Result ⇒ 3
```

thenApplyAsync(), thenAcceptAsync()

thread-x

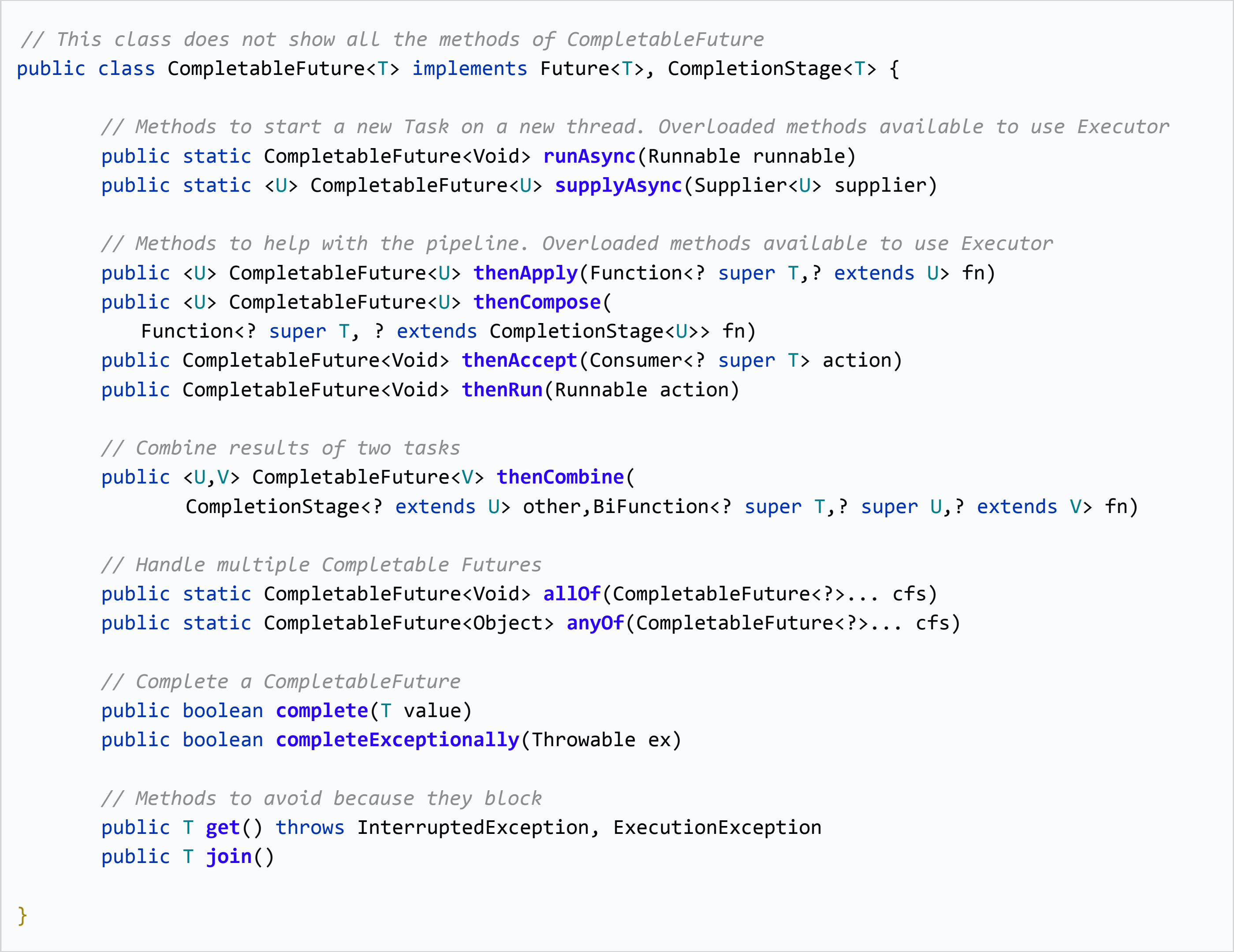
```
// An executor service declared
private static ExecutorService mypool = Executors.newCachedThreadPool();

// Execute a task in common pool
// then Apply a function on another executor service thread
// then Accept the result on another executor service thread
CompletableFuture pipeline =
        CompletableFuture.supplyAsync(() -> FuturesPlay.doTask("SomeTask", 3, false))
                 .thenApplyAsync(taskResult -> taskResult.secs(), mypool)
                 .thenAcceptAsync(time -> {
                     System.out.println(time);
                 }, mypool);
```

common.thread-0

mypool.thread-0

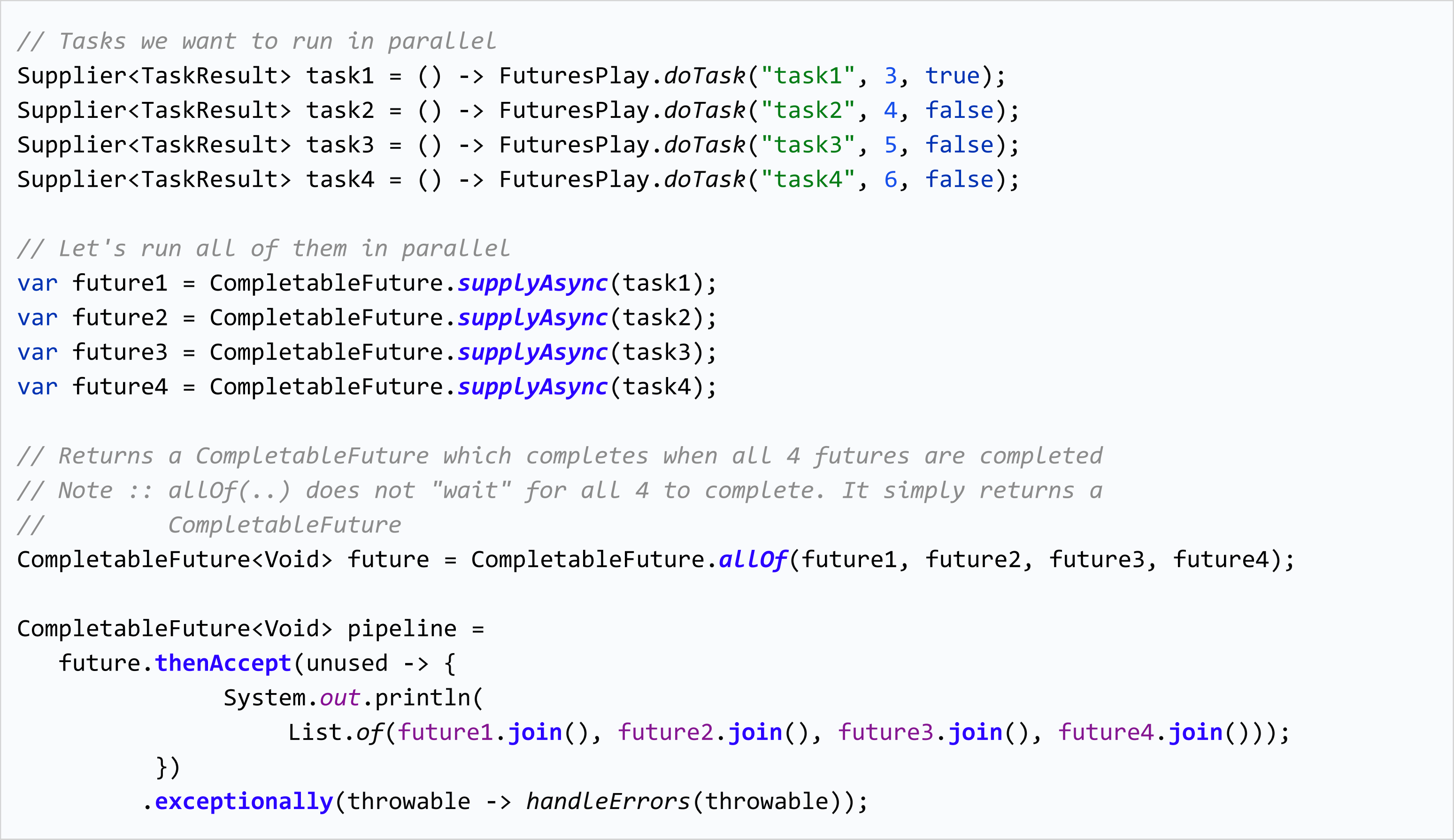
mypool.thread-1

```
Result ⇒ 3
```



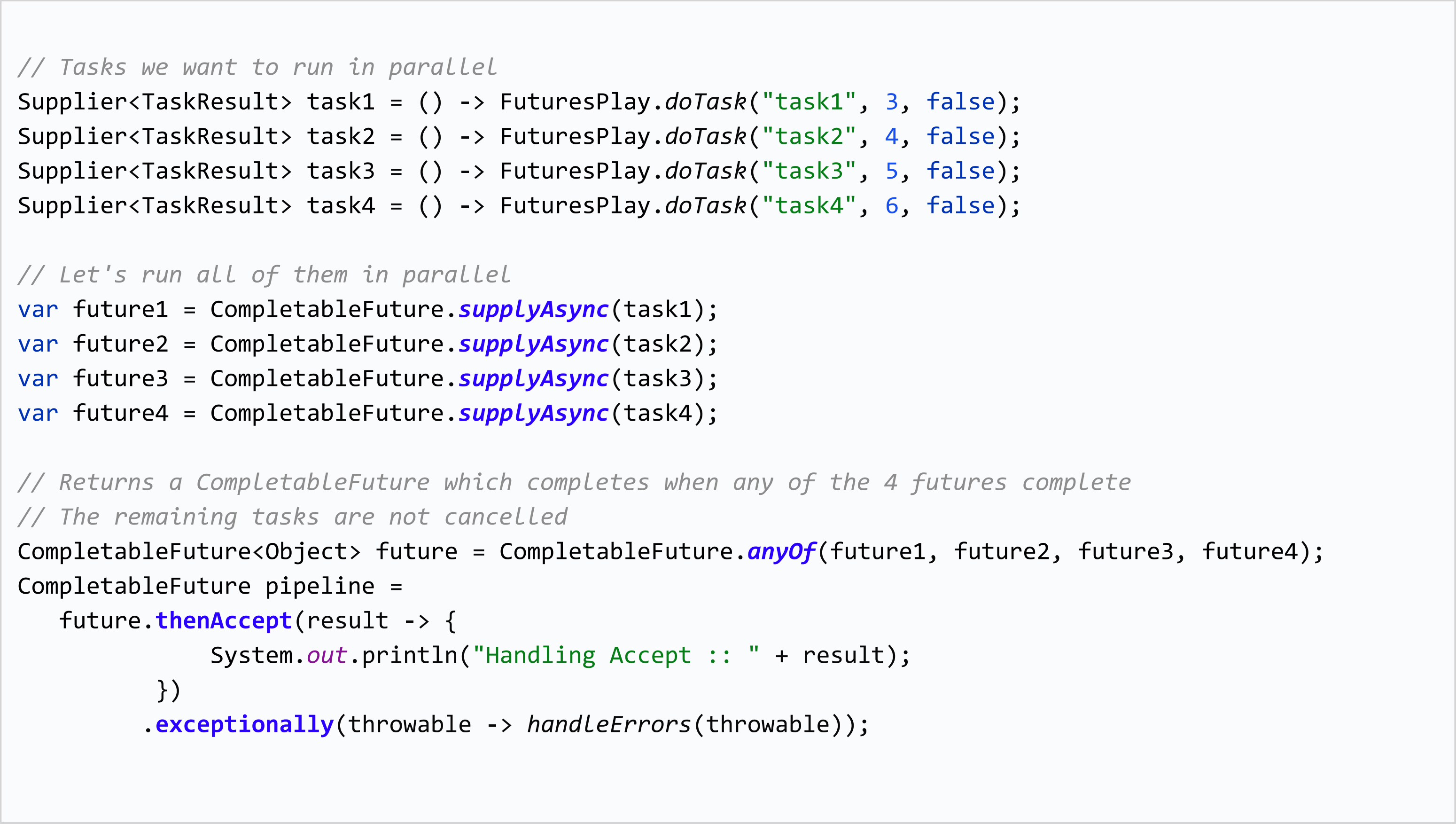
```
// This class does not show all the methods of CompletableFuture
public class CompletableFuture<T> implements Future<T>, CompletionStage<T> {

       // Methods to start a new Task on a new thread. Overloaded methods available to use Executor
       public static CompletableFuture<Void> runAsync(Runnable runnable)
       public static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier)

       // Methods to help with the pipeline. Overloaded methods available to use Executor
       public <U> CompletableFuture<U> thenApply(Function<? super T,? extends U> fn)
       public <U> CompletableFuture<U> thenCompose(
          Function<? super T, ? extends CompletionStage<U>> fn)
       public CompletableFuture<Void> thenAccept(Consumer<? super T> action)
       public CompletableFuture<Void> thenRun(Runnable action)

       // Combine results of two tasks
       public <U,V> CompletableFuture<V> thenCombine(
              CompletionStage<? extends U> other,BiFunction<? super T,? super U,? extends V> fn)

       // Handle multiple Completable Futures
       public static CompletableFuture<Void> allOf(CompletableFuture<?>... cfs)
       public static CompletableFuture<Object> anyOf(CompletableFuture<?>... cfs)

       // Complete a CompletableFuture
       public boolean complete(T value)
       public boolean completeExceptionally(Throwable ex)

       // Methods to avoid because they block
       public T get() throws InterruptedException, ExecutionException
       public T join()

}
```

allOf(..)



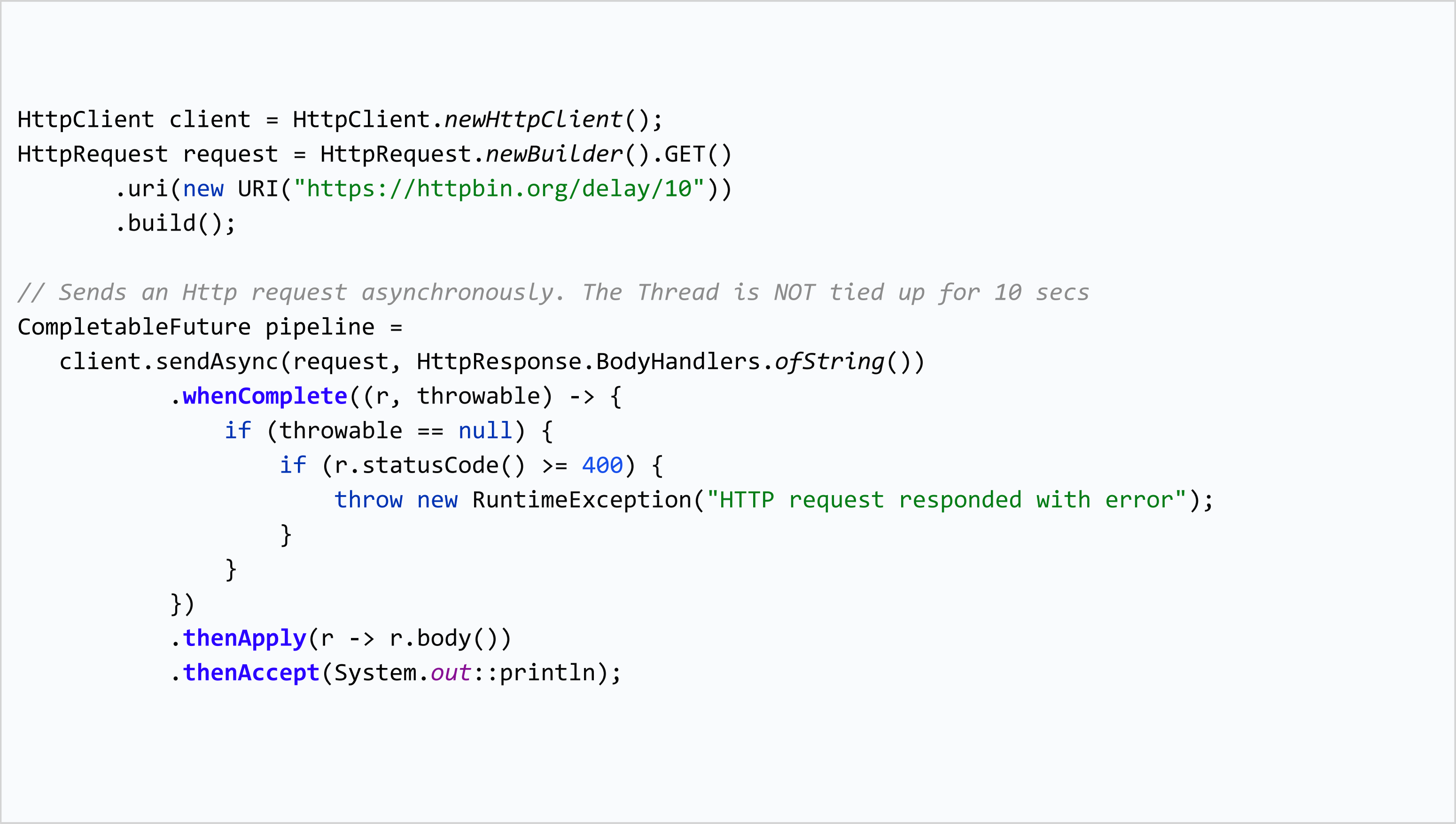
```
// Tasks we want to run in parallel
Supplier<TaskResult> task1 = () -> FuturesPlay.doTask("task1", 3, true);
Supplier<TaskResult> task2 = () -> FuturesPlay.doTask("task2", 4, false);
Supplier<TaskResult> task3 = () -> FuturesPlay.doTask("task3", 5, false);
Supplier<TaskResult> task4 = () -> FuturesPlay.doTask("task4", 6, false);

// Let's run all of them in parallel
var future1 = CompletableFuture.supplyAsync(task1);
var future2 = CompletableFuture.supplyAsync(task2);
var future3 = CompletableFuture.supplyAsync(task3);
var future4 = CompletableFuture.supplyAsync(task4);

// Returns a CompletableFuture which completes when all 4 futures are completed
// Note :: allOf(..) does not "wait" for all 4 to complete. It simply returns a
//         CompletableFuture
CompletableFuture<Void> future = CompletableFuture.allOf(future1, future2, future3, future4);

CompletableFuture<Void> pipeline =
   future.thenAccept(unused -> {
               System.out.println(
                    List.of(future1.join(), future2.join(), future3.join(), future4.join()));
          })
         .exceptionally(throwable -> handleErrors(throwable));
```

anyOf(..)

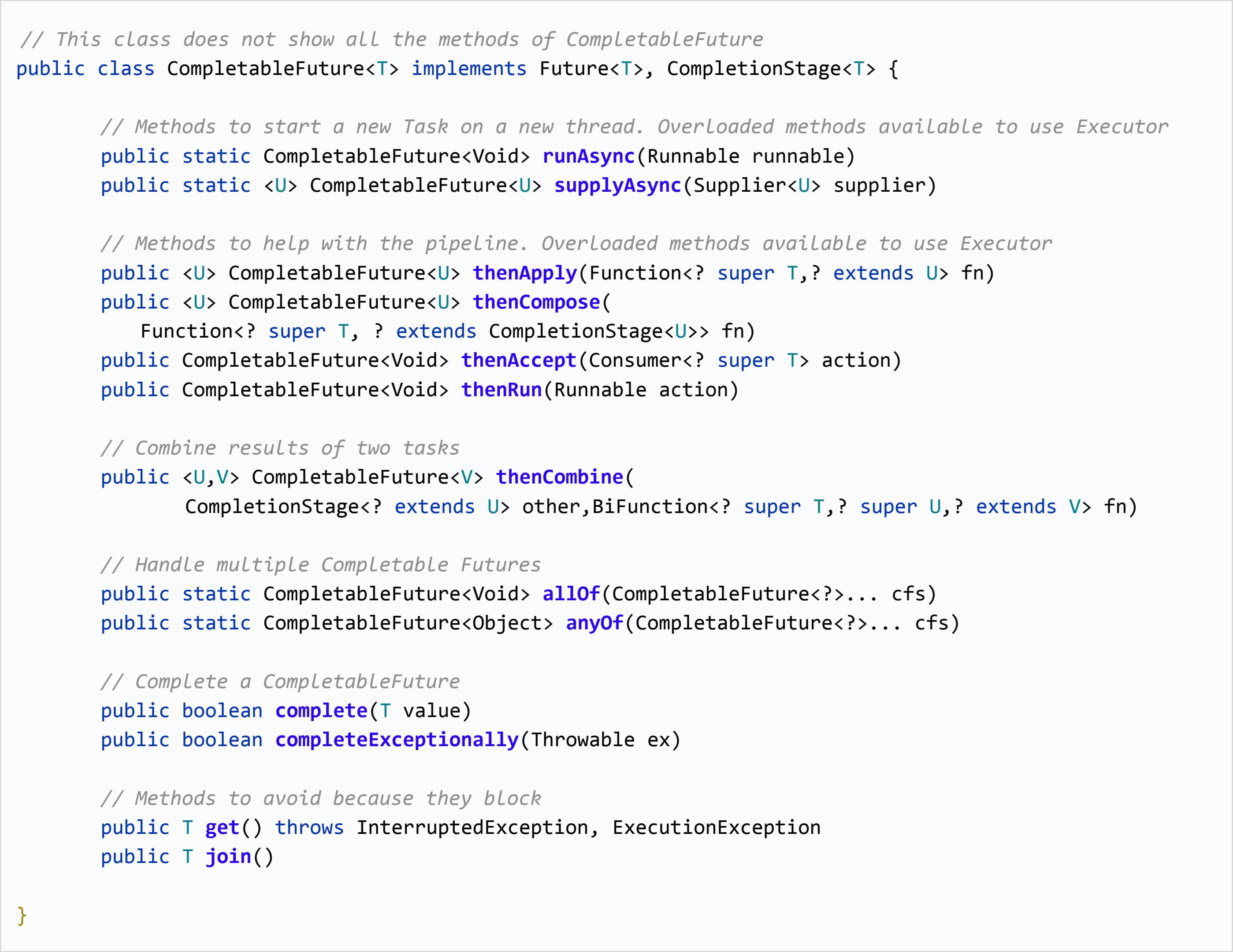


```java
// Tasks we want to run in parallel
Supplier<TaskResult> task1 = () -> FuturesPlay.doTask("task1", 3, false);
Supplier<TaskResult> task2 = () -> FuturesPlay.doTask("task2", 4, false);
Supplier<TaskResult> task3 = () -> FuturesPlay.doTask("task3", 5, false);
Supplier<TaskResult> task4 = () -> FuturesPlay.doTask("task4", 6, false);

// Let's run all of them in parallel
var future1 = CompletableFuture.supplyAsync(task1);
var future2 = CompletableFuture.supplyAsync(task2);
var future3 = CompletableFuture.supplyAsync(task3);
var future4 = CompletableFuture.supplyAsync(task4);

// Returns a CompletableFuture which completes when any of the 4 futures complete
// The remaining tasks are not cancelled
CompletableFuture<Object> future = CompletableFuture.anyOf(future1, future2, future3, future4);
CompletableFuture pipeline =
   future.thenAccept(result -> {
              System.out.println("Handling Accept :: " + result);
          })
         .exceptionally(throwable -> handleErrors(throwable));
```

HttpClient - Asynchronous HTTP example



```
HttpClient client = HttpClient.newHttpClient();
HttpRequest request = HttpRequest.newBuilder().GET()
       .uri(new URI("https://httpbin.org/delay/10"))
       .build();

// Sends an Http request asynchronously. The Thread is NOT tied up for 10 secs
CompletableFuture pipeline =
   client.sendAsync(request, HttpResponse.BodyHandlers.ofString())
           .whenComplete((r, throwable) -> {
               if (throwable == null) {
                   if (r.statusCode() >= 400) {
                       throw new RuntimeException("HTTP request responded with error");
                   }
               }
           })
           .thenApply(r -> r.body())
           .thenAccept(System.out::println);
```



```java
// This class does not show all the methods of CompletableFuture
public class CompletableFuture<T> implements Future<T>, CompletionStage<T> {

        // Methods to start a new Task on a new thread. Overloaded methods available to use Executor
        public static CompletableFuture<Void> runAsync(Runnable runnable)
        public static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier)

        // Methods to help with the pipeline. Overloaded methods available to use Executor
        public <U> CompletableFuture<U> thenApply(Function<? super T,? extends U> fn)
        public <U> CompletableFuture<U> thenCompose(
            Function<? super T, ? extends CompletionStage<U>> fn)
        public CompletableFuture<Void> thenAccept(Consumer<? super T> action)
        public CompletableFuture<Void> thenRun(Runnable action)

        // Combine results of two tasks
        public <U,V> CompletableFuture<V> thenCombine(
                CompletionStage<? extends U> other,BiFunction<? super T,? super U,? extends V> fn)

        // Handle multiple Completable Futures
        public static CompletableFuture<Void> allOf(CompletableFuture<?>... cfs)
        public static CompletableFuture<Object> anyOf(CompletableFuture<?>... cfs)

        // Complete a CompletableFuture
        public boolean complete(T value)
        public boolean completeExceptionally(Throwable ex)

        // Methods to avoid because they block
        public T get() throws InterruptedException, ExecutionException
        public T join()

}
```

Completion Methods

- Create a CompletableFuture

```
var future
    = new CompletableFuture<T>();
```

- Completion Methods

```
future.complete(result)
future.completeExceptionally(e)
```

```
private static CompletableFuture<String> readFileAsync(String filename) throws IOException {

   // Create a Completable Future
   CompletableFuture<String> future = new CompletableFuture<>();

   // Create a Path to a file in the current working directory.
   Path path = Paths.get(".").resolve(fileName);
   AsynchronousFileChannel fileChannel
           = AsynchronousFileChannel.open(path, StandardOpenOption.READ);
   ByteBuffer buffer = ByteBuffer.allocate((int)path.toFile().length());
   fileChannel.read(buffer, 0, buffer, new CompletionHandler<Integer, ByteBuffer>() {
               @Override
               public void completed(Integer result, ByteBuffer attachment) {

                   // extract the data from the attachment
                   attachment.flip();
                   byte[] data = new byte[attachment.limit()];
                   attachment.get(data);
                   attachment.clear();

                   // complete successfully
                   future.complete(new String(data));

               }
               @Override
               public void failed(Throwable exc, ByteBuffer attachment) {
                   // complete exceptionally
                   future.completeExceptionally(exc);
               }
           });

   // Return the CompletableFuture
   return future;
}
```

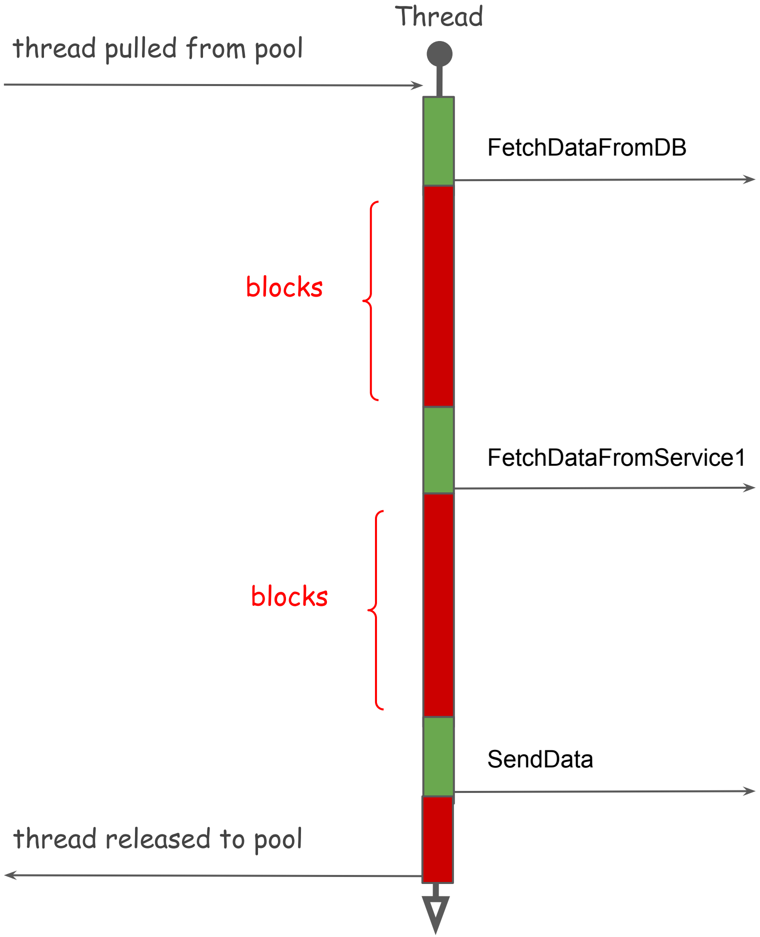
whenComplete

```java
// Let's run all of them in parallel
var future1 = CompletableFuture.supplyAsync(() -> FuturesPlay.doTask("task1", 3, false));
var future2 = CompletableFuture.supplyAsync(() -> FuturesPlay.doTask("task2", 4, false));
var future3 = CompletableFuture.supplyAsync(() -> FuturesPlay.doTask("task3", 5, false));
var future4 = CompletableFuture.supplyAsync(() -> FuturesPlay.doTask("task4", 6, false));

// Returns a CompletableFuture which completes when all 4 futures are completed
// whenComplete stage is called when previous stage completes - successfully or not
CompletableFuture pipeline =
    CompletableFuture
            .allOf(future1, future2, future3, future4)
            .whenComplete((unused, throwable) -> {

                if (throwable == null) {
                    System.out.println(
                            List.of(future1.join(), future2.join(), future3.join(), future4.join()));
                }
                else {
                    handleErrors(throwable);
                }

            });
```

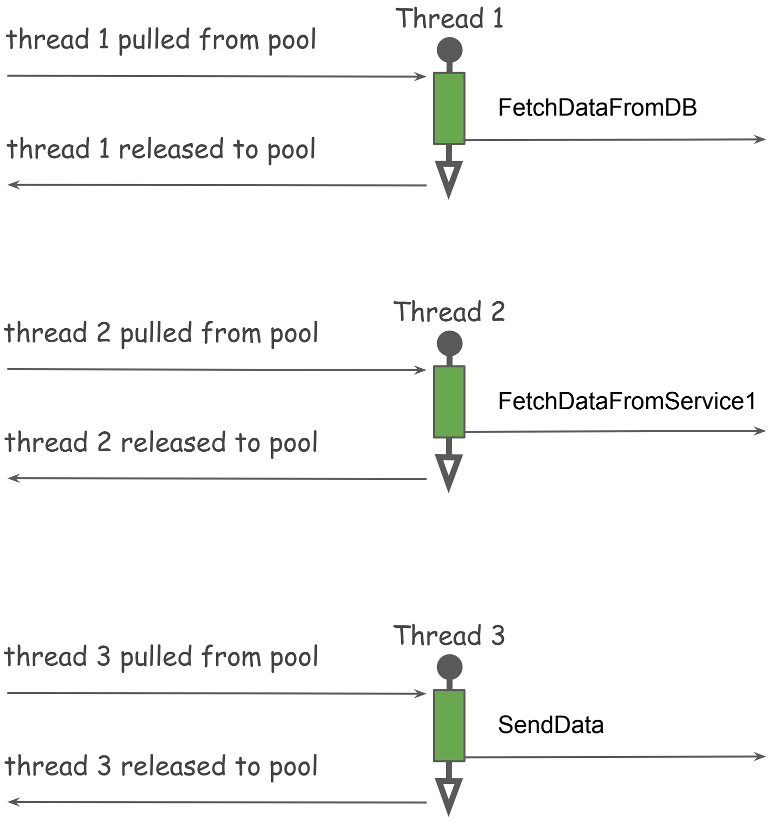
orTimeout

```
Supplier<TaskResult> task1 = () -> FuturesPlay.doTask("task1", 3, false);
CompletableFuture pipeline =
   CompletableFuture.supplyAsync(task1)
           .orTimeout(1, TimeUnit.SECONDS)
           .thenAccept(System.out::println);
```

Blocking IO

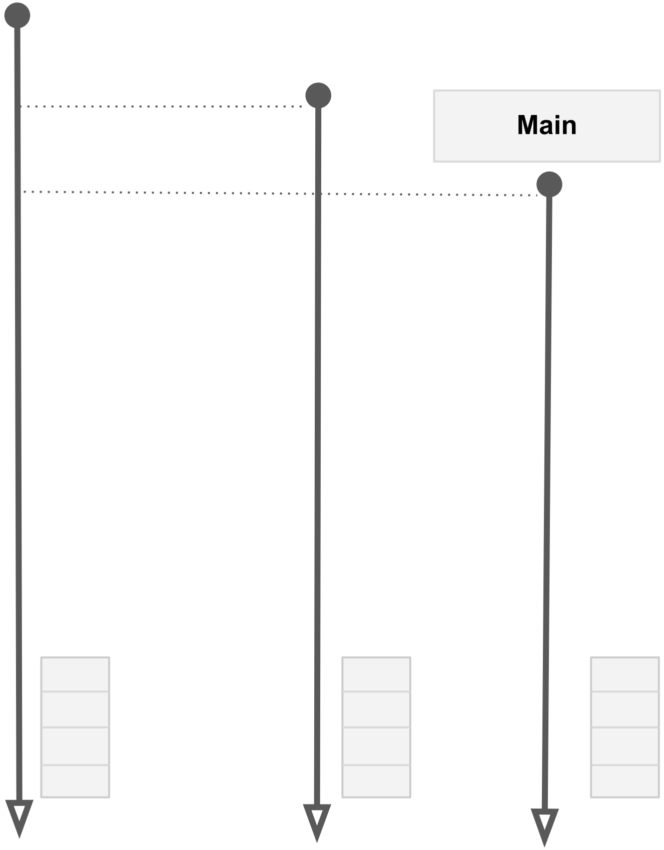


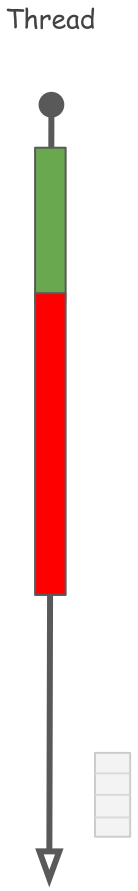
Non Blocking IO

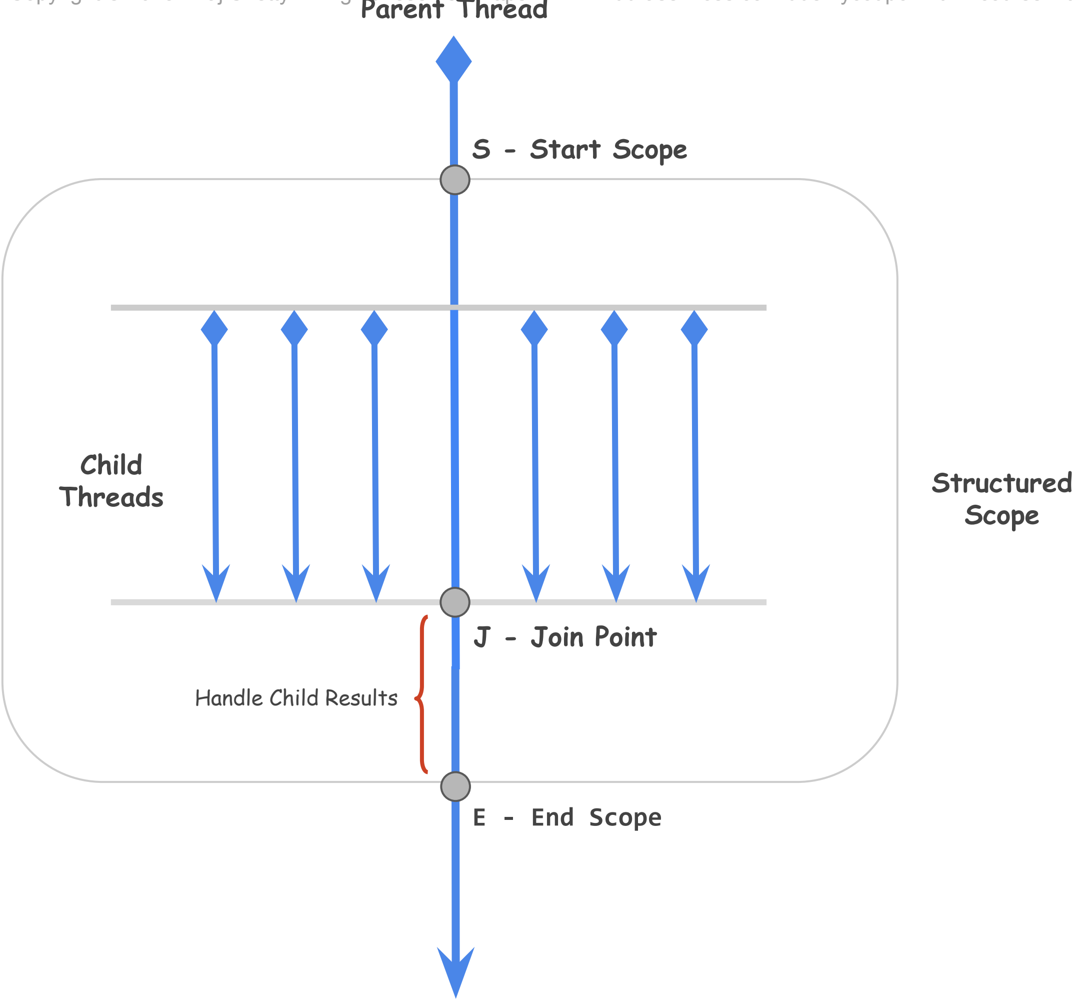

Thread 1
thread 1 pulled from pool
FetchDataFromDB
thread 1 released to pool
Thread 2
thread 2 pulled from pool
FetchDataFromService1
thread 2 released to pool
Thread 3
thread 3 pulled from pool
SendData
thread 3 released to pool

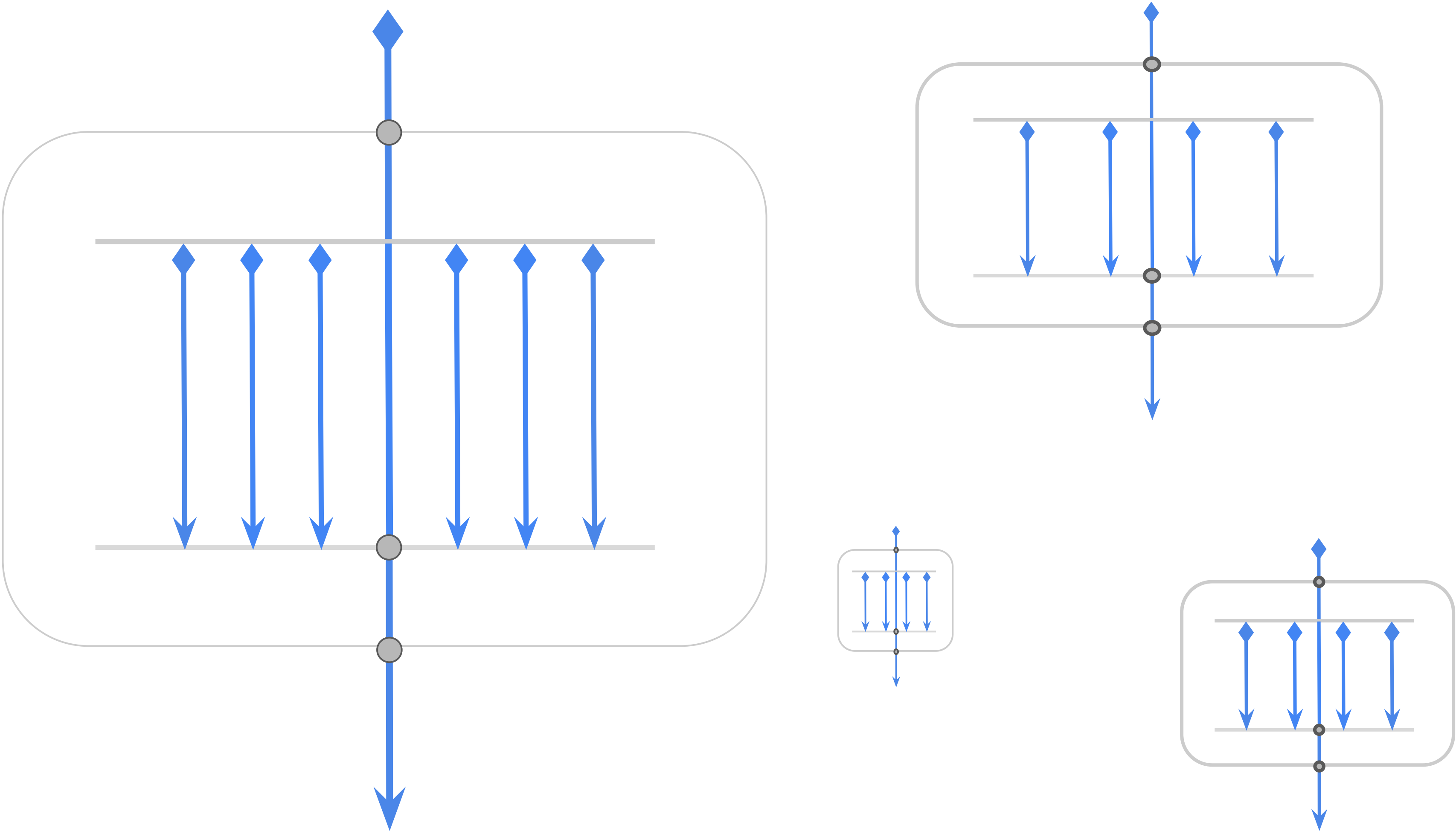
End Of Course

Thread images






Parent Thread
S - Start Scope
Child Threads
Structured Scope
J - Join Point
Handle Child Results
E - End Scope



Note that the signature verification only requires the certificate of the IDP. The Entity IDs and the Endpoints are asked by SAML Developer Tools for other SAML validations

Ignore the SingleLogoutService tags for now. You may not see it in your metadata

SAML tab must be open when the SAML Request is sent - so we can see the requests dynamically. That’s why I am resending the Request

This is the **Single Sign On Request** and not the Single Logout Request

In practice, SHA1 must not be used. I am just testing to see that the IDP in fact would use SHA1 as Digest Algorithm

I am defining 4 fields - **first, last, title and zip**. The Assertion will contain these attributes for the user.

You have the option to change the Thread Factory in the constructor

The **record** featur added in
Java 14 to
for Immut
name of t
required it
are autom
9

The Suppressed exception originates from the **close** method of the scope and not from the **join**

ShutdownOnFailure is a static inner class of **StructuredTaskScope**. That's the reason for syntax

StructuredTaskScope.ShutdownOnFailure

Note that the Thread.interrupted() method clears the interrupt status flag after returning but this is okay because we are throwing InterruptedException to tell calling method that method was interrupted.

Note that in this design, both **dbCall** and **restCall** are catching all exceptions and returning **null**. will not throw e interrupted

In

ming that you have a wledge of **Java**

The latest **JDK Loom** Early Access build should be used to run all examples in this course. Even though I mention **JDK 19 Early Access Build** in the course, the examples are valid for future releases as well.

Here a metho **self::dbCall** is being passed as a **lambda** to the **submit** method. dbCall behaves like the **call** method of the **Callable**

assuming that both **dbCall** and **restCall** ssary even if either of them return **null**.

All exceptions are essentially swallowed

In later v lasses e

I am continuously sending the request

An HTTP **GET** request http://localhost:8080/demo will be handled by method **getThreadInfo()**

Eclipse automatically detected that there are application properties specific to **MASCOT** and **WONDER**. That's why it showed **MASCOT** and **WONDER** as two profiles