Complete Guide to Java Threading and Spring Boot Concurrency

Table of Contents

Single-Threading

What it is

Single-threading means executing one task at a time in sequence. Each task must complete before the next one begins. In Java, this is the default behavior of the main thread.

Under the hood: The JVM allocates one thread of execution with its own stack memory. Tasks are executed sequentially on the CPU core assigned to this thread.

Why and when to use it

- **Simple applications** with minimal I/O operations
- **CPU-intensive tasks** that don't benefit from parallelization
- **Sequential dependencies** where order matters
- **Resource-constrained environments**

How to implement it

```java
// Simple single-threaded example
public class SingleThreadedExample {
    public static void main(String[] args) {
        System.out.println("Task 1 starting...");
        performTask("Task 1", 2000);

        System.out.println("Task 2 starting...");
        performTask("Task 2", 1500);
```

```java
        System.out.println("Task 3 starting...");
        performTask("Task 3", 1000);

        System.out.println("All tasks completed!");
    }

    private static void performTask(String taskName, int durationMs) {
        try {
            Thread.sleep(durationMs); // Simulating work
            System.out.println(taskName + " completed in " + durationMs + "ms");
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}
```

Best Practices

- Keep tasks short and non-blocking
- Handle exceptions properly
- Avoid blocking operations in critical paths
- Use for simple, sequential workflows

Multi-Threading

What it is

Multi-threading allows multiple threads to execute concurrently, potentially on different CPU cores. Each thread has its own stack but shares heap memory.

Under the hood: The JVM creates multiple threads, each with 1MB stack space (default). The OS scheduler distributes these threads across available CPU cores.

Why and when to use it

- **I/O-intensive operations** (file reading, network calls)
- **CPU-intensive tasks** that can be parallelized
- **Background processing** while keeping UI responsive
- **Handling multiple client requests** simultaneously

How to implement it

Basic Threading

```java
public class BasicMultiThreading {
    public static void main(String[] args) {
        // Method 1: Extending Thread
        Thread thread1 = new CustomThread("Thread-1");

        // Method 2: Implementing Runnable
        Thread thread2 = new Thread(new CustomRunnable("Thread-2"));

        // Method 3: Lambda expression
        Thread thread3 = new Thread(() -> {
            performTask("Lambda-Thread", 1000);
        });

        thread1.start();
        thread2.start();
        thread3.start();

        // Wait for all threads to complete
        try {
            thread1.join();
            thread2.join();
            thread3.join();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }

        System.out.println("All threads completed!");
    }

    static class CustomThread extends Thread {
        private String name;

        public CustomThread(String name) {
            this.name = name;
        }

        @Override
        public void run() {
            performTask(name, 2000);
        }
    }

    static class CustomRunnable implements Runnable {
        private String name;

        public CustomRunnable(String name) {
```

```java
            this.name = name;
        }

        @Override
        public void run() {
            performTask(name, 1500);
        }
    }

    private static void performTask(String taskName, int durationMs) {
        try {
            System.out.println(taskName + " started on " + Thread.currentThread().getName()
            Thread.sleep(durationMs);
            System.out.println(taskName + " completed!");
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}
```

Advanced: Thread Synchronization

```java
public class ThreadSynchronization {
    private static int counter = 0;
    private static final Object lock = new Object();

    public static void main(String[] args) throws InterruptedException {
        List<Thread> threads = new ArrayList<>();

        // Create 10 threads that increment counter
        for (int i = 0; i < 10; i++) {
            Thread thread = new Thread(() -> {
                for (int j = 0; j < 1000; j++) {
                    incrementCounter();
                }
            });
            threads.add(thread);
            thread.start();
        }

        // Wait for all threads
        for (Thread thread : threads) {
            thread.join();
        }

        System.out.println("Final counter value: " + counter);
```

```java
        // Should be 10,000 with proper synchronization
    }

    private static void incrementCounter() {
        synchronized (lock) {
            counter++;
        }
    }
}
```

Best Practices

- Always call `start()`, never `run()` directly
- Use `join()` to wait for thread completion
- Handle `InterruptedException` properly
- Avoid shared mutable state or use synchronization
- Use thread-safe collections when needed

Future

What it is

Future represents the result of an asynchronous computation. It provides methods to check if the computation is complete, wait for completion, and retrieve the result.

Under the hood: Future is an interface that acts as a placeholder for a value that will be available later. It's typically returned by ExecutorService when submitting tasks.

Why and when to use it

- **Asynchronous task execution** with result retrieval
- **Non-blocking operations** where you need the result later
- **Timeout handling** for long-running operations
- **Task cancellation** capabilities

How to implement it

Basic Future Usage

```java
import java.util.concurrent.*;

public class FutureExample {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(3);
```

```java
        try {
            // Submit callable tasks that return values
            Future<String> future1 = executor.submit(new ApiCallTask("https://api1.com"));
            Future<String> future2 = executor.submit(new ApiCallTask("https://api2.com"));
            Future<Integer> future3 = executor.submit(new CalculationTask(100));

            // Do other work while tasks execute
            System.out.println("Tasks submitted, doing other work...");
            Thread.sleep(500);

            // Get results (blocking calls)
            try {
                String result1 = future1.get(5, TimeUnit.SECONDS);
                String result2 = future2.get(5, TimeUnit.SECONDS);
                Integer result3 = future3.get(5, TimeUnit.SECONDS);

                System.out.println("Results: " + result1 + ", " + result2 + ", " + result3);
            } catch (TimeoutException e) {
                System.out.println("Task timed out!");
                future1.cancel(true);
                future2.cancel(true);
                future3.cancel(true);
            }

        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        } finally {
            executor.shutdown();
        }
    }

    static class ApiCallTask implements Callable<String> {
        private String url;

        public ApiCallTask(String url) {
            this.url = url;
        }

        @Override
        public String call() throws Exception {
            // Simulate API call
            Thread.sleep(2000);
            return "Response from " + url;
        }
    }
```

```
    static class CalculationTask implements Callable<Integer> {
        private int input;

        public CalculationTask(int input) {
            this.input = input;
        }

        @Override
        public Integer call() throws Exception {
            // Simulate heavy calculation
            Thread.sleep(1000);
            return input * input;
        }
    }
}
```

Advanced: Future with Error Handling

```
public class AdvancedFutureExample {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(2);

        List<Future<String>> futures = new ArrayList<>();

        // Submit multiple tasks
        for (int i = 1; i <= 5; i++) {
            final int taskId = i;
            Future<String> future = executor.submit(() -> {
                if (taskId == 3) {
                    throw new RuntimeException("Task " + taskId + " failed!");
                }
                Thread.sleep(1000 * taskId);
                return "Task " + taskId + " completed";
            });
            futures.add(future);
        }

        // Process results as they complete
        for (int i = 0; i < futures.size(); i++) {
            try {
                String result = futures.get(i).get();
                System.out.println("Success: " + result);
            } catch (ExecutionException e) {
                System.out.println("Task " + (i + 1) + " failed: " + e.getCause().getMessage
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
```

```java
                break;
            }
        }

        executor.shutdown();
    }
}
```

Best Practices

- Always use timeouts with `get(timeout, unit)`
- Handle `ExecutionException` and `InterruptedException`
- Cancel futures when no longer needed
- Use `isDone()` and `isCancelled()` for status checking

CompletableFuture

What it is

CompletableFuture is an enhanced Future that supports functional-style programming with method chaining, composition, and better exception handling. It implements both Future and CompletionStage interfaces.

Under the hood: Built on top of ForkJoinPool, it uses a callback-based approach with continuation passing style for non-blocking operations.

Why and when to use it

- **Complex async workflows** with dependencies between tasks
- **Non-blocking programming** with callbacks
- **Functional composition** of async operations
- **Better error handling** compared to raw Future

How to implement it

Basic CompletableFuture

```java
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.TimeUnit;

public class CompletableFutureBasics {
    public static void main(String[] args) {
        // Creating CompletableFuture in different ways

        // 1. Completed future
        CompletableFuture<String> completed = CompletableFuture.completedFuture("Hello");
```

```java
        // 2. Async supplier
        CompletableFuture<String> async = CompletableFuture.supplyAsync(() -> {
            try {
                Thread.sleep(1000);
                return "Async result";
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        });

        // 3. Async runnable (no return value)
        CompletableFuture<Void> runAsync = CompletableFuture.runAsync(() -> {
            System.out.println("Background task executed");
        });

        // Chain operations
        CompletableFuture<String> chained = async
            .thenApply(result -> result.toUpperCase())
            .thenApply(result -> "Processed: " + result);

        // Get results
        try {
            System.out.println(completed.get());
            System.out.println(chained.get(3, TimeUnit.SECONDS));
            runAsync.get();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Advanced: Composition and Error Handling

```java
public class AdvancedCompletableFuture {
    public static void main(String[] args) {
        // Complex async workflow
        CompletableFuture<String> userFuture = fetchUser(123);
        CompletableFuture<String> profileFuture = fetchUserProfile(123);

        // Combine two independent futures
        CompletableFuture<String> combinedFuture = userFuture
            .thenCombine(profileFuture, (user, profile) ->
                "User: " + user + ", Profile: " + profile);

        // Chain dependent operations
        CompletableFuture<String> processedFuture = userFuture
```

```java
            .thenCompose(user -> fetchUserPreferences(user))
            .thenApply(preferences -> "Processed preferences: " + preferences)
            .exceptionally(throwable -> {
                System.err.println("Error occurred: " + throwable.getMessage());
                return "Default preferences";
            });

        // Handle both success and failure
        CompletableFuture<String> handledFuture = fetchUser(456)
            .handle((result, throwable) -> {
                if (throwable != null) {
                    return "Error: " + throwable.getMessage();
                }
                return "Success: " + result;
            });

        // Wait for all to complete
        CompletableFuture<Void> allOf = CompletableFuture.allOf(
            combinedFuture, processedFuture, handledFuture
        );

        allOf.thenRun(() -> {
            try {
                System.out.println("Combined: " + combinedFuture.get());
                System.out.println("Processed: " + processedFuture.get());
                System.out.println("Handled: " + handledFuture.get());
            } catch (Exception e) {
                e.printStackTrace();
            }
        }).join();
    }

    private static CompletableFuture<String> fetchUser(int userId) {
        return CompletableFuture.supplyAsync(() -> {
            // Simulate API call
            try {
                Thread.sleep(1000);
                if (userId == 456) {
                    throw new RuntimeException("User not found");
                }
                return "User-" + userId;
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        });
    }
```

```java
    private static CompletableFuture<String> fetchUserProfile(int userId) {
        return CompletableFuture.supplyAsync(() -> {
            try {
                Thread.sleep(800);
                return "Profile-" + userId;
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        });
    }

    private static CompletableFuture<String> fetchUserPreferences(String user) {
        return CompletableFuture.supplyAsync(() -> {
            try {
                Thread.sleep(500);
                return "Preferences for " + user;
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        });
    }
}
```

Best Practices

- Use method chaining for readable async workflows
- Handle exceptions with `exceptionally()` or `handle()`
- Use `thenCompose()` for dependent operations, `thenCombine()` for independent ones
- Prefer `supplyAsync()` over manual thread creation
- Use custom executors for better control

ExecutorService

What it is

ExecutorService is a high-level interface for managing and controlling thread execution. It provides thread pools, task scheduling, and lifecycle management.

Under the hood: Manages a pool of worker threads, uses blocking queues for task submission, and handles thread lifecycle automatically.

Why and when to use it

- **Thread pool management** to avoid thread creation overhead

- **Resource control** to limit concurrent threads
- **Task queuing** when threads are busy
- **Graceful shutdown** of threading resources

How to implement it

Basic ExecutorService

```
import java.util.concurrent.*;
import java.util.List;
import java.util.ArrayList;

public class ExecutorServiceExample {
    public static void main(String[] args) {
        // Different types of thread pools

        // 1. Fixed thread pool
        ExecutorService fixedPool = Executors.newFixedThreadPool(3);

        // 2. Cached thread pool (creates threads as needed)
        ExecutorService cachedPool = Executors.newCachedThreadPool();

        // 3. Single thread executor
        ExecutorService singleExecutor = Executors.newSingleThreadExecutor();

        // 4. Scheduled executor
        ScheduledExecutorService scheduledExecutor = Executors.newScheduledThreadPool(2);

        try {
            // Submit tasks to fixed pool
            List<Future<String>> futures = new ArrayList<>();

            for (int i = 1; i <= 5; i++) {
                final int taskId = i;
                Future<String> future = fixedPool.submit(() -> {
                    Thread.sleep(1000 * taskId);
                    return "Task " + taskId + " completed by " + Thread.currentThread().getN
                });
                futures.add(future);
            }

            // Collect results
            for (Future<String> future : futures) {
                System.out.println(future.get());
            }
```

```
            // Schedule recurring task
            scheduledExecutor.scheduleAtFixedRate(() -> {
                System.out.println("Scheduled task executed at " + System.currentTimeMillis(
            }, 0, 2, TimeUnit.SECONDS);

            Thread.sleep(6000); // Let scheduled task run a few times

        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            // Proper shutdown
            shutdownExecutor(fixedPool);
            shutdownExecutor(cachedPool);
            shutdownExecutor(singleExecutor);
            shutdownExecutor(scheduledExecutor);
        }
    }

    private static void shutdownExecutor(ExecutorService executor) {
        executor.shutdown();
        try {
            if (!executor.awaitTermination(5, TimeUnit.SECONDS)) {
                executor.shutdownNow();
            }
        } catch (InterruptedException e) {
            executor.shutdownNow();
            Thread.currentThread().interrupt();
        }
    }
}
```

Custom ThreadPoolExecutor

```
public class CustomThreadPoolExample {
    public static void main(String[] args) {
        // Custom thread pool with specific parameters
        ThreadPoolExecutor customExecutor = new ThreadPoolExecutor(
            2,                      // core pool size
            4,                      // maximum pool size
            60L,                    // keep alive time
            TimeUnit.SECONDS,       // time unit
            new ArrayBlockingQueue<>(10), // work queue
            new CustomThreadFactory(),    // thread factory
            new ThreadPoolExecutor.CallerRunsPolicy() // rejection policy
        );
```

```java
        // Submit tasks
        for (int i = 1; i <= 15; i++) {
            final int taskId = i;
            customExecutor.submit(() -> {
                try {
                    System.out.println("Task " + taskId + " started by " +
                        Thread.currentThread().getName());
                    Thread.sleep(2000);
                    System.out.println("Task " + taskId + " completed");
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            });
        }

        // Monitor pool status
        System.out.println("Active threads: " + customExecutor.getActiveCount());
        System.out.println("Pool size: " + customExecutor.getPoolSize());
        System.out.println("Queue size: " + customExecutor.getQueue().size());

        customExecutor.shutdown();
    }

    static class CustomThreadFactory implements ThreadFactory {
        private int counter = 1;

        @Override
        public Thread newThread(Runnable r) {
            Thread thread = new Thread(r, "CustomWorker-" + counter++);
            thread.setDaemon(false);
            return thread;
        }
    }
}
```

Best Practices

- Choose appropriate thread pool type for your use case
- Always shutdown executors properly
- Handle rejection policies appropriately
- Monitor thread pool metrics in production
- Use custom ThreadFactory for better thread naming

Virtual Threads (Java 19+)

What it is

Virtual threads are lightweight threads managed by the JVM rather than the OS. They're designed for high-throughput concurrent applications with millions of threads.

Under the hood: Virtual threads are mapped to a small number of OS threads (carrier threads) by the JVM. They're suspended when blocked and resumed when unblocked, allowing massive concurrency with minimal memory overhead.

Why and when to use it

- **High-concurrency applications** (millions of concurrent operations)
- **I/O-intensive workloads** where threads spend time waiting
- **Microservices** with many external API calls
- **Web servers** handling many concurrent requests

How to implement it

Basic Virtual Threads (Java 21+)

```java
// Note: This requires Java 21+ for stable virtual threads
public class VirtualThreadsExample {
    public static void main(String[] args) throws InterruptedException {
        // Create virtual thread executor
        try (ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor()) {

            // Submit many lightweight tasks
            List<Future<String>> futures = new ArrayList<>();

            for (int i = 1; i <= 10000; i++) {
                final int taskId = i;
                Future<String> future = executor.submit(() -> {
                    try {
                        // Simulate I/O operation
                        Thread.sleep(1000);
                        return "Virtual task " + taskId + " on " + Thread.currentThread();
                    } catch (InterruptedException e) {
                        throw new RuntimeException(e);
                    }
                });
                futures.add(future);
            }

            // Process first 10 results
            for (int i = 0; i < 10; i++) {
```

```java
                try {
                    System.out.println(futures.get(i).get());
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }

            System.out.println("Submitted 10,000 virtual threads successfully!");
        }
    }
}
```

Virtual Threads with Structured Concurrency

```java
import java.util.concurrent.StructuredTaskScope;

// Java 21+ structured concurrency (preview feature)
public class StructuredConcurrencyExample {

    public String fetchUserData(int userId) throws Exception {
        try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {

            // Start multiple related tasks
            var userTask = scope.fork(() -> fetchUser(userId));
            var profileTask = scope.fork(() -> fetchProfile(userId));
            var preferencesTask = scope.fork(() -> fetchPreferences(userId));

            // Wait for all to complete or any to fail
            scope.join();           // Wait for all tasks
            scope.throwIfFailed();  // Throw if any failed

            // All succeeded, combine results
            return combineUserData(
                userTask.resultNow(),
                profileTask.resultNow(),
                preferencesTask.resultNow()
            );
        }
    }

    private String fetchUser(int userId) throws InterruptedException {
        Thread.sleep(1000);
        return "User-" + userId;
    }

    private String fetchProfile(int userId) throws InterruptedException {
```

```java
        Thread.sleep(800);
        return "Profile-" + userId;
    }

    private String fetchPreferences(int userId) throws InterruptedException {
        Thread.sleep(600);
        return "Preferences-" + userId;
    }

    private String combineUserData(String user, String profile, String preferences) {
        return String.format("Combined: %s, %s, %s", user, profile, preferences);
    }
}
```

Best Practices

- Use for I/O-intensive applications
- Avoid CPU-intensive tasks in virtual threads
- Don't use ThreadLocal extensively (memory overhead)
- Use structured concurrency for related task groups
- Monitor carrier thread utilization

Spring-Managed Threads

What it is

Spring provides several mechanisms for async processing including @Async annotation, TaskExecutor, and TaskScheduler. Spring manages the thread lifecycle and configuration.

Under the hood: Spring uses AOP proxies to intercept @Async method calls and execute them on configured thread pools.

Why and when to use it

- **Spring Boot applications** needing async processing
- **Declarative async programming** with annotations
- **Integration with Spring ecosystem** (transactions, security)
- **Configuration-driven** thread management

How to implement it

Basic @Async Setup

```java
// Configuration
@Configuration
@EnableAsync
```

```java
public class AsyncConfig {

    @Bean(name = "taskExecutor")
    public TaskExecutor taskExecutor() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        executor.setCorePoolSize(2);
        executor.setMaxPoolSize(5);
        executor.setQueueCapacity(100);
        executor.setThreadNamePrefix("Async-");
        executor.setRejectedExecutionHandler(new ThreadPoolExecutor.CallerRunsPolicy());
        executor.initialize();
        return executor;
    }

    @Bean(name = "longRunningTaskExecutor")
    public TaskExecutor longRunningTaskExecutor() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        executor.setCorePoolSize(1);
        executor.setMaxPoolSize(3);
        executor.setQueueCapacity(50);
        executor.setThreadNamePrefix("LongTask-");
        executor.initialize();
        return executor;
    }
}

// Service class
@Service
public class AsyncService {

    private static final Logger logger = LoggerFactory.getLogger(AsyncService.class);

    @Async("taskExecutor")
    public CompletableFuture<String> processDataAsync(String data) {
        logger.info("Processing {} on thread {}", data, Thread.currentThread().getName());

        try {
            // Simulate processing
            Thread.sleep(2000);
            String result = "Processed: " + data.toUpperCase();
            return CompletableFuture.completedFuture(result);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            return CompletableFuture.failedFuture(e);
        }
    }
```

```java
    @Async("longRunningTaskExecutor")
    public void performBackgroundTask(String taskName) {
        logger.info("Background task {} started on thread {}",
            taskName, Thread.currentThread().getName());

        try {
            Thread.sleep(5000);
            logger.info("Background task {} completed", taskName);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            logger.error("Background task {} interrupted", taskName);
        }
    }

    @Async
    public CompletableFuture<List<String>> processMultipleItems(List<String> items) {
        return CompletableFuture.supplyAsync(() -> {
            return items.stream()
                .map(item -> "Processed: " + item)
                .collect(Collectors.toList());
        });
    }
}
```

Spring Boot Application

```java
@SpringBootApplication
@EnableAsync
public class AsyncDemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(AsyncDemoApplication.class, args);
    }
}

@RestController
public class AsyncController {

    @Autowired
    private AsyncService asyncService;

    @GetMapping("/process/{data}")
    public CompletableFuture<ResponseEntity<String>> processData(@PathVariable String data)
        return asyncService.processDataAsync(data)
            .thenApply(result -> ResponseEntity.ok(result))
```

```java
            .exceptionally(throwable ->
                ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
                    .body("Error: " + throwable.getMessage()));
    }

    @PostMapping("/background-task")
    public ResponseEntity<String> startBackgroundTask(@RequestBody String taskName) {
        asyncService.performBackgroundTask(taskName);
        return ResponseEntity.accepted().body("Task started: " + taskName);
    }

    @PostMapping("/process-batch")
    public CompletableFuture<ResponseEntity<List<String>>> processBatch(
            @RequestBody List<String> items) {
        return asyncService.processMultipleItems(items)
            .thenApply(results -> ResponseEntity.ok(results));
    }
}
```

Advanced: Custom Async Exception Handler

```java
@Component
public class CustomAsyncExceptionHandler implements AsyncUncaughtExceptionHandler {

    private static final Logger logger = LoggerFactory.getLogger(CustomAsyncExceptionHandler

    @Override
    public void handleUncaughtException(Throwable throwable, Method method, Object... params
        logger.error("Async method {} failed with parameters {}",
            method.getName(), Arrays.toString(params), throwable);

        // You could also send notifications, metrics, etc.
        sendErrorNotification(method.getName(), throwable);
    }

    private void sendErrorNotification(String methodName, Throwable error) {
        // Implementation for error notification
        logger.warn("Sending error notification for method: {}", methodName);
    }
}

@Configuration
@EnableAsync
public class AsyncConfigWithExceptionHandler implements AsyncConfigurer {

    @Override
```

```java
    public Executor getAsyncExecutor() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        executor.setCorePoolSize(3);
        executor.setMaxPoolSize(6);
        executor.setQueueCapacity(100);
        executor.setThreadNamePrefix("SpringAsync-");
        executor.initialize();
        return executor;
    }

    @Override
    public AsyncUncaughtExceptionHandler getAsyncUncaughtExceptionHandler() {
        return new CustomAsyncExceptionHandler();
    }
}
```

Best Practices

- Configure appropriate thread pool sizes
- Use different executors for different task types
- Implement proper exception handling
- Avoid calling @Async methods from the same class
- Monitor thread pool metrics

Java Threading Guide - Part 2: Spring & Advanced Concepts

Spring-Managed Threads (Continued)

Advanced Spring Async Patterns

Event-Driven Async Processing

```java
@Component
public class OrderEventHandler {

    private static final Logger logger = LoggerFactory.getLogger(OrderEventHandler.class);

    @EventListener
    @Async("orderProcessingExecutor")
    public void handleOrderCreated(OrderCreatedEvent event) {
        logger.info("Processing order {} asynchronously", event.getOrderId());

        try {
            // Simulate order processing
```

```java
            Thread.sleep(3000);

            // Send confirmation email
            sendConfirmationEmail(event.getOrderId());

            // Update inventory
            updateInventory(event.getItems());

            logger.info("Order {} processing completed", event.getOrderId());
        } catch (Exception e) {
            logger.error("Failed to process order {}", event.getOrderId(), e);
        }
    }

    @EventListener
    @Async("notificationExecutor")
    public void handlePaymentProcessed(PaymentProcessedEvent event) {
        // Send notification asynchronously
        sendPaymentNotification(event.getOrderId(), event.getAmount());
    }

    private void sendConfirmationEmail(String orderId) throws InterruptedException {
        Thread.sleep(1000);
        logger.info("Confirmation email sent for order {}", orderId);
    }

    private void updateInventory(List<String> items) throws InterruptedException {
        Thread.sleep(500);
        logger.info("Inventory updated for items: {}", items);
    }

    private void sendPaymentNotification(String orderId, double amount) {
        logger.info("Payment notification sent for order {} amount ${}", orderId, amount);
    }
}

// Event classes
public class OrderCreatedEvent {
    private String orderId;
    private List<String> items;

    // constructors, getters, setters
    public OrderCreatedEvent(String orderId, List<String> items) {
        this.orderId = orderId;
        this.items = items;
    }
```

```java
    public String getOrderId() { return orderId; }
    public List<String> getItems() { return items; }
}

public class PaymentProcessedEvent {
    private String orderId;
    private double amount;

    public PaymentProcessedEvent(String orderId, double amount) {
        this.orderId = orderId;
        this.amount = amount;
    }

    public String getOrderId() { return orderId; }
    public double getAmount() { return amount; }
}
```

Conditional Async Execution

```java
@Service
public class ConditionalAsyncService {

    @Value("${app.async.enabled:true}")
    private boolean asyncEnabled;

    public CompletableFuture<String> processData(String data) {
        if (asyncEnabled) {
            return processDataAsync(data);
        } else {
            return CompletableFuture.completedFuture(processDataSync(data));
        }
    }

    @Async
    public CompletableFuture<String> processDataAsync(String data) {
        try {
            Thread.sleep(2000);
            return CompletableFuture.completedFuture("Async: " + data.toUpperCase());
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            return CompletableFuture.failedFuture(e);
        }
    }

    public String processDataSync(String data) {
```

```java
        try {
            Thread.sleep(2000);
            return "Sync: " + data.toUpperCase();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            throw new RuntimeException(e);
        }
    }
}
```

Evolution of Java Concurrency

Timeline and Key Improvements

1. Thread Class (Java 1.0 - 1996)

```java
// Basic thread creation
Thread thread = new Thread(() -> {
    System.out.println("Hello from thread!");
});
thread.start();
```

Limitations: Manual lifecycle, resource leaks, poor scalability

2. ExecutorService (Java 5 - 2004)

```java
// Thread pool management
ExecutorService executor = Executors.newFixedThreadPool(5);
Future<String> future = executor.submit(() -> "Hello");
String result = future.get();
executor.shutdown();
```

Improvements: Thread pooling, lifecycle management, better resource control

3. CompletableFuture (Java 8 - 2014)

```java
// Functional async programming
CompletableFuture<String> future = CompletableFuture
    .supplyAsync(() -> "Hello")
    .thenApply(String::toUpperCase)
    .thenCompose(this::processString);
```

Improvements: Method chaining, better composition, exception handling

4. Virtual Threads (Java 19-21)

```java
// Lightweight concurrency
try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
```

```java
    // Can handle millions of concurrent tasks
    for (int i = 0; i < 1_000_000; i++) {
        executor.submit(() -> {
            // I/O intensive work
        });
    }
}
```

Improvements: Massive concurrency, lower memory footprint, simpler programming model

---

Comparison Table

Feature	Single-Threading	Multi-Threading	Future	CompletableFuture	ExecutorService	Virtual Threads	Spring @Async
Complexity	Very Low	Medium	Medium	High	Medium	Low	Low
Performance	Poor for I/O	Good	Good	Excellent	Good	Excellent	Good
Memory Usage	Low	High	High	Medium	Medium	Very Low	Medium
Scalability	Poor	Limited	Limited	Good	Good	Excellent	Good
Error Handling	Simple	Complex	Basic	Advanced	Basic	Simple	Spring-managed
Composability	N/A	Poor	Limited	Excellent	Limited	Good	Limited
Learning Curve	Easy	Hard	Medium	Hard	Medium	Easy	Easy
Best Use Case	Simple tasks	CPU-intensive	Basic async	Complex workflows	General purpose	High I/O	Spring apps

---

Real-World Mini-Projects

Project 1: API Response Aggregator

```java
@Service
public class ApiAggregatorService {

    private final RestTemplate restTemplate = new RestTemplate();
```

```java
// Single-threaded approach (slow)
public AggregatedResponse fetchDataSingleThreaded(List<String> apiUrls) {
    List<String> responses = new ArrayList<>();
    long startTime = System.currentTimeMillis();

    for (String url : apiUrls) {
        try {
            String response = restTemplate.getForObject(url, String.class);
            responses.add(response);
        } catch (Exception e) {
            responses.add("Error: " + e.getMessage());
        }
    }

    long duration = System.currentTimeMillis() - startTime;
    return new AggregatedResponse(responses, duration);
}

// CompletableFuture approach (fast)
public CompletableFuture<AggregatedResponse> fetchDataAsync(List<String> apiUrls) {
    long startTime = System.currentTimeMillis();

    List<CompletableFuture<String>> futures = apiUrls.stream()
        .map(this::fetchSingleApiAsync)
        .collect(Collectors.toList());

    return CompletableFuture.allOf(futures.toArray(new CompletableFuture[0]))
        .thenApply(v -> {
            List<String> responses = futures.stream()
                .map(CompletableFuture::join)
                .collect(Collectors.toList());

            long duration = System.currentTimeMillis() - startTime;
            return new AggregatedResponse(responses, duration);
        });
}

private CompletableFuture<String> fetchSingleApiAsync(String url) {
    return CompletableFuture.supplyAsync(() -> {
        try {
            return restTemplate.getForObject(url, String.class);
        } catch (Exception e) {
            return "Error: " + e.getMessage();
        }
    });
}
```

```java
    // Virtual threads approach (Java 21+)
    public AggregatedResponse fetchDataVirtualThreads(List<String> apiUrls) {
        long startTime = System.currentTimeMillis();
        List<String> responses = new ArrayList<>();

        try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
            List<Future<String>> futures = apiUrls.stream()
                .map(url -> executor.submit(() -> {
                    try {
                        return restTemplate.getForObject(url, String.class);
                    } catch (Exception e) {
                        return "Error: " + e.getMessage();
                    }
                }))
                .collect(Collectors.toList());

            for (Future<String> future : futures) {
                try {
                    responses.add(future.get(5, TimeUnit.SECONDS));
                } catch (Exception e) {
                    responses.add("Timeout or error");
                }
            }
        }

        long duration = System.currentTimeMillis() - startTime;
        return new AggregatedResponse(responses, duration);
    }
}

public class AggregatedResponse {
    private List<String> responses;
    private long durationMs;

    public AggregatedResponse(List<String> responses, long durationMs) {
        this.responses = responses;
        this.durationMs = durationMs;
    }

    // getters and toString
    public List<String> getResponses() { return responses; }
    public long getDurationMs() { return durationMs; }

    @Override
    public String toString() {
```

```java
        return String.format("AggregatedResponse{responses=%d, duration=%dms}",
            responses.size(), durationMs);
    }
}
```

Project 2: Background Job Processing System

```java
@Component
public class JobProcessingSystem {

    private final BlockingQueue<Job> jobQueue = new LinkedBlockingQueue<>();
    private final ExecutorService jobProcessor = Executors.newFixedThreadPool(3);
    private final ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(1);

    @PostConstruct
    public void startProcessing() {
        // Start job processors
        for (int i = 0; i < 3; i++) {
            jobProcessor.submit(this::processJobs);
        }

        // Schedule cleanup task
        scheduler.scheduleAtFixedRate(this::cleanupCompletedJobs, 0, 30, TimeUnit.SECONDS);
    }

    public void submitJob(Job job) {
        job.setStatus(JobStatus.QUEUED);
        job.setSubmittedAt(System.currentTimeMillis());
        jobQueue.offer(job);
        logger.info("Job {} queued", job.getId());
    }

    private void processJobs() {
        while (!Thread.currentThread().isInterrupted()) {
            try {
                Job job = jobQueue.take(); // Blocking operation
                processJob(job);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                break;
            }
        }
    }

    private void processJob(Job job) {
        try {
```

```java
            job.setStatus(JobStatus.PROCESSING);
            job.setStartedAt(System.currentTimeMillis());

            logger.info("Processing job {} on thread {}",
                job.getId(), Thread.currentThread().getName());

            // Simulate job processing
            Thread.sleep(job.getDurationMs());

            job.setStatus(JobStatus.COMPLETED);
            job.setCompletedAt(System.currentTimeMillis());

            logger.info("Job {} completed successfully", job.getId());

        } catch (InterruptedException e) {
            job.setStatus(JobStatus.FAILED);
            Thread.currentThread().interrupt();
        } catch (Exception e) {
            job.setStatus(JobStatus.FAILED);
            logger.error("Job {} failed", job.getId(), e);
        }
    }

    private void cleanupCompletedJobs() {
        // Implementation for cleanup logic
        logger.info("Cleaning up completed jobs...");
    }

    @PreDestroy
    public void shutdown() {
        jobProcessor.shutdown();
        scheduler.shutdown();
        try {
            if (!jobProcessor.awaitTermination(10, TimeUnit.SECONDS)) {
                jobProcessor.shutdownNow();
            }
            if (!scheduler.awaitTermination(5, TimeUnit.SECONDS)) {
                scheduler.shutdownNow();
            }
        } catch (InterruptedException e) {
            jobProcessor.shutdownNow();
            scheduler.shutdownNow();
            Thread.currentThread().interrupt();
        }
    }
}
```

```java
// Job class
public class Job {
    private String id;
    private JobStatus status;
    private long submittedAt;
    private long startedAt;
    private long completedAt;
    private int durationMs;

    public Job(String id, int durationMs) {
        this.id = id;
        this.durationMs = durationMs;
        this.status = JobStatus.CREATED;
    }

    // getters and setters
    public String getId() { return id; }
    public JobStatus getStatus() { return status; }
    public void setStatus(JobStatus status) { this.status = status; }
    public long getSubmittedAt() { return submittedAt; }
    public void setSubmittedAt(long submittedAt) { this.submittedAt = submittedAt; }
    public long getStartedAt() { return startedAt; }
    public void setStartedAt(long startedAt) { this.startedAt = startedAt; }
    public long getCompletedAt() { return completedAt; }
    public void setCompletedAt(long completedAt) { this.completedAt = completedAt; }
    public int getDurationMs() { return durationMs; }
}

enum JobStatus {
    CREATED, QUEUED, PROCESSING, COMPLETED, FAILED
}
```

Project 3: Parallel Data Processing Pipeline

```java
@Service
public class DataProcessingPipeline {

    private final ExecutorService ioExecutor = Executors.newFixedThreadPool(5);
    private final ExecutorService cpuExecutor = Executors.newFixedThreadPool(
        Runtime.getRuntime().availableProcessors());

    public CompletableFuture<ProcessingResult> processLargeDataset(String filePath) {
        return CompletableFuture
            // Stage 1: Read data (I/O intensive)
            .supplyAsync(() -> readDataFromFile(filePath), ioExecutor)
```

```
        // Stage 2: Parse data (CPU intensive)
        .thenComposeAsync(rawData -> parseData(rawData), cpuExecutor)
        // Stage 3: Process in parallel
        .thenComposeAsync(this::processDataInParallel, cpuExecutor)
        // Stage 4: Aggregate results
        .thenApply(this::aggregateResults)
        // Stage 5: Save results (I/O intensive)
        .thenComposeAsync(result -> saveResults(result), ioExecutor)
        .exceptionally(throwable -> {
            logger.error("Pipeline failed", throwable);
            return new ProcessingResult("Failed", 0, throwable.getMessage());
        });
}

private String readDataFromFile(String filePath) {
    try {
        logger.info("Reading data from {} on thread {}", filePath, Thread.currentThread(
        Thread.sleep(2000); // Simulate file I/O
        return "raw_data_content_from_" + filePath;
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
        throw new RuntimeException(e);
    }
}

private CompletableFuture<List<DataItem>> parseData(String rawData) {
    return CompletableFuture.supplyAsync(() -> {
        try {
            logger.info("Parsing data on thread {}", Thread.currentThread().getName());
            Thread.sleep(1000); // Simulate parsing

            List<DataItem> items = new ArrayList<>();
            for (int i = 1; i <= 100; i++) {
                items.add(new DataItem("item_" + i, i * 10));
            }
            return items;
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            throw new RuntimeException(e);
        }
    }, cpuExecutor);
}

private CompletableFuture<List<ProcessedItem>> processDataInParallel(List<DataItem> item
    // Split data into chunks for parallel processing
    int chunkSize = 25;
```

```
    List<List<DataItem>> chunks = partition(items, chunkSize);

    List<CompletableFuture<List<ProcessedItem>>> chunkFutures = chunks.stream()
        .map(chunk -> CompletableFuture.supplyAsync(() -> processChunk(chunk), cpuExecut
        .collect(Collectors.toList());

    return CompletableFuture.allOf(chunkFutures.toArray(new CompletableFuture[0]))
        .thenApply(v -> chunkFutures.stream()
            .map(CompletableFuture::join)
            .flatMap(List::stream)
            .collect(Collectors.toList()));
}

private List<ProcessedItem> processChunk(List<DataItem> chunk) {
    logger.info("Processing chunk of {} items on thread {}",
        chunk.size(), Thread.currentThread().getName());

    return chunk.stream()
        .map(item -> {
            try {
                Thread.sleep(50); // Simulate processing
                return new ProcessedItem(item.getName(), item.getValue() * 2, "processed
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                throw new RuntimeException(e);
            }
        })
        .collect(Collectors.toList());
}

private ProcessingResult aggregateResults(List<ProcessedItem> items) {
    logger.info("Aggregating {} results on thread {}",
        items.size(), Thread.currentThread().getName());

    int totalValue = items.stream()
        .mapToInt(ProcessedItem::getValue)
        .sum();

    return new ProcessingResult("Success", totalValue, null);
}

private CompletableFuture<ProcessingResult> saveResults(ProcessingResult result) {
    return CompletableFuture.supplyAsync(() -> {
        try {
            logger.info("Saving results on thread {}", Thread.currentThread().getName()
            Thread.sleep(1000); // Simulate database save
```

```java
                result.setSaved(true);
                return result;
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                throw new RuntimeException(e);
            }
        }, ioExecutor);
    }

    private <T> List<List<T>> partition(List<T> list, int chunkSize) {
        List<List<T>> partitions = new ArrayList<>();
        for (int i = 0; i < list.size(); i += chunkSize) {
            partitions.add(list.subList(i, Math.min(i + chunkSize, list.size())));
        }
        return partitions;
    }

    @PreDestroy
    public void cleanup() {
        ioExecutor.shutdown();
        cpuExecutor.shutdown();
    }
}

// Data classes
class DataItem {
    private String name;
    private int value;

    public DataItem(String name, int value) {
        this.name = name;
        this.value = value;
    }

    public String getName() { return name; }
    public int getValue() { return value; }
}

class ProcessedItem {
    private String name;
    private int value;
    private String status;

    public ProcessedItem(String name, int value, String status) {
        this.name = name;
        this.value = value;
```

```
        this.status = status;
    }

    public String getName() { return name; }
    public int getValue() { return value; }
    public String getStatus() { return status; }
}

class ProcessingResult {
    private String status;
    private int totalValue;
    private String errorMessage;
    private boolean saved = false;

    public ProcessingResult(String status, int totalValue, String errorMessage) {
        this.status = status;
        this.totalValue = totalValue;
        this.errorMessage = errorMessage;
    }

    public String getStatus() { return status; }
    public int getTotalValue() { return totalValue; }
    public String getErrorMessage() { return errorMessage; }
    public boolean isSaved() { return saved; }
    public void setSaved(boolean saved) { this.saved = saved; }
}
```

Project 4: Real-Time Notification System

```
@Service
public class NotificationService {

    private final Map<String, List<NotificationListener>> subscribers = new ConcurrentHashMa
    private final ExecutorService notificationExecutor = Executors.newFixedThreadPool(10);

    @Async("notificationExecutor")
    public CompletableFuture<Void> sendNotification(Notification notification) {
        List<NotificationListener> listeners = subscribers.get(notification.getTopic());

        if (listeners == null || listeners.isEmpty()) {
            return CompletableFuture.completedFuture(null);
        }

        // Send to all listeners in parallel
        List<CompletableFuture<Void>> sendTasks = listeners.stream()
            .map(listener -> sendToListener(listener, notification))
```

```java
            .collect(Collectors.toList());

        return CompletableFuture.allOf(sendTasks.toArray(new CompletableFuture[0]));
    }

    private CompletableFuture<Void> sendToListener(NotificationListener listener, Notificati
        return CompletableFuture.runAsync(() -> {
            try {
                logger.info("Sending notification {} to listener {} on thread {}",
                    notification.getId(), listener.getId(), Thread.currentThread().getName(

                listener.onNotification(notification);

                // Simulate network delay
                Thread.sleep(200);

                logger.info("Notification {} sent successfully to {}",
                    notification.getId(), listener.getId());

            } catch (Exception e) {
                logger.error("Failed to send notification {} to listener {}",
                    notification.getId(), listener.getId(), e);
            }
        }, notificationExecutor);
    }

    public void subscribe(String topic, NotificationListener listener) {
        subscribers.computeIfAbsent(topic, k -> new CopyOnWriteArrayList<>()).add(listener)
        logger.info("Listener {} subscribed to topic {}", listener.getId(), topic);
    }

    public void unsubscribe(String topic, NotificationListener listener) {
        List<NotificationListener> listeners = subscribers.get(topic);
        if (listeners != null) {
            listeners.remove(listener);
            logger.info("Listener {} unsubscribed from topic {}", listener.getId(), topic);
        }
    }

    @PreDestroy
    public void shutdown() {
        notificationExecutor.shutdown();
        try {
            if (!notificationExecutor.awaitTermination(10, TimeUnit.SECONDS)) {
                notificationExecutor.shutdownNow();
            }
```

```java
        } catch (InterruptedException e) {
            notificationExecutor.shutdownNow();
            Thread.currentThread().interrupt();
        }
    }
}

// Supporting classes
class Notification {
    private String id;
    private String topic;
    private String message;
    private long timestamp;

    public Notification(String id, String topic, String message) {
        this.id = id;
        this.topic = topic;
        this.message = message;
        this.timestamp = System.currentTimeMillis();
    }

    // getters
    public String getId() { return id; }
    public String getTopic() { return topic; }
    public String getMessage() { return message; }
    public long getTimestamp() { return timestamp; }
}

interface NotificationListener {
    String getId();
    void onNotification(Notification notification);
}

@Component
class EmailNotificationListener implements NotificationListener {
    @Override
    public String getId() {
        return "email-listener";
    }

    @Override
    public void onNotification(Notification notification) {
        logger.info("Sending email for notification: {}", notification.getMessage());
        // Email sending logic
    }
}
```

Interview Questions & Answers

Q1: What's the difference between `submit()` and `execute()` in ExecutorService?

Answer: - `execute(Runnable)`: Fire-and-forget, no return value, exceptions are handled by UncaughtExceptionHandler - `submit(Callable/Runnable)`: Returns Future, exceptions can be retrieved via `future.get()`

```
// execute() - no return value
executor.execute(() -> System.out.println("Fire and forget"));

// submit() - returns Future
Future<String> future = executor.submit(() -> "I can return values");
```

Q2: Why might CompletableFuture.get() be dangerous in production?

Answer: `get()` blocks the calling thread indefinitely. In web applications, this can exhaust the request-handling thread pool.

```
// BAD - blocks indefinitely
String result = future.get();

// GOOD - with timeout
String result = future.get(5, TimeUnit.SECONDS);

// BETTER - non-blocking
future.thenAccept(result -> processResult(result));
```

Q3: What happens if you call an @Async method from the same class?

Answer: The method executes synchronously because Spring's AOP proxy is bypassed (self-invocation problem).

```
@Service
public class AsyncService {

    @Async
    public void asyncMethod() {
        // This will be async when called from outside
    }

    public void callerMethod() {
        asyncMethod(); // This will be SYNCHRONOUS!
```

```java
    }
}

// Solution: Inject self or use separate service
@Service
public class AsyncService {

    @Autowired
    private AsyncService self; // Self-injection

    public void callerMethod() {
        self.asyncMethod(); // Now this will be async
    }
}
```

Q4: How do Virtual Threads differ from Platform Threads?

Answer: - **Platform Threads**: 1:1 mapping to OS threads, ~1MB stack, expensive creation - **Virtual Threads**: M:N mapping, ~KB memory, very cheap creation

```java
// Platform threads - limited scalability
try (var executor = Executors.newFixedThreadPool(1000)) {
    // Can handle ~1000 concurrent operations
}

// Virtual threads - massive scalability
try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
    // Can handle millions of concurrent operations
}
```

Q5: What are the thread safety issues with CompletableFuture?

Answer: - The CompletableFuture itself is thread-safe - But the tasks you run and shared state they access may not be - Race conditions can occur with shared mutable objects

```java
// Thread-safe approach
private final AtomicInteger counter = new AtomicInteger(0);

CompletableFuture<Integer> future = CompletableFuture.supplyAsync(() -> {
    return counter.incrementAndGet(); // Atomic operation
});
```

Q6: How do you handle exceptions in different threading approaches?

Answer:

```java
// Traditional Future
try {
    String result = future.get();
} catch (ExecutionException e) {
    Throwable cause = e.getCause(); // Original exception
}

// CompletableFuture
CompletableFuture<String> future = CompletableFuture
    .supplyAsync(this::riskyOperation)
    .exceptionally(throwable -> "Default value")
    .handle((result, throwable) -> {
        if (throwable != null) {
            logger.error("Operation failed", throwable);
            return "Error occurred";
        }
        return result;
    });

// Spring @Async - implement AsyncUncaughtExceptionHandler
@Override
public void handleUncaughtException(Throwable throwable, Method method, Object... params) {
    logger.error("Async method {} failed", method.getName(), throwable);
}
```

Q7: What's the "blocking vs non-blocking" concept?

Answer: - **Blocking**: Thread waits for operation to complete (e.g., `future.get()`) - **Non-blocking**: Thread continues execution, handles result via callbacks

```java
// Blocking approach
String result = future.get(); // Thread waits here
processResult(result);

// Non-blocking approach
future.thenAccept(result -> processResult(result)); // Thread continues immediately
```

Q8: How do you choose thread pool sizes?

Answer: - **CPU-intensive**: Number of cores (Runtime.getRuntime().availableProcessors()) - **I/O-intensive**: Much higher (cores × 2 to cores × 50) - **Mixed workload**: Separate pools for different task types

```java
// CPU-intensive pool
int cpuThreads = Runtime.getRuntime().availableProcessors();
ExecutorService cpuPool = Executors.newFixedThreadPool(cpuThreads);
```

```java
// I/O-intensive pool
int ioThreads = cpuThreads * 10; // Higher multiplier for I/O
ExecutorService ioPool = Executors.newFixedThreadPool(ioThreads);
```

Best Practices Summary

General Threading Best Practices

1. **Choose the right tool**:
    - Simple sequential: Single-threading
    - CPU-intensive: Multi-threading with core-count pools
    - I/O-intensive: CompletableFuture or Virtual Threads
    - Spring apps: @Async with proper configuration
2. **Resource Management**:
    - Always shutdown executors
    - Use try-with-resources when possible
    - Monitor thread pool metrics
    - Set appropriate timeouts
3. **Exception Handling**:
    - Never ignore InterruptedException
    - Use proper exception handlers for async operations
    - Log errors with context
    - Provide fallback mechanisms
4. **Thread Safety**:
    - Avoid shared mutable state
    - Use concurrent collections
    - Understand happens-before relationships
    - Use atomic operations when appropriate
5. **Performance Optimization**:
    - Profile before optimizing
    - Separate thread pools for different workloads
    - Use appropriate queue sizes
    - Monitor for thread starvation

Spring-Specific Best Practices

1. **Configuration**:
    - Configure multiple executors for different purposes
    - Set appropriate pool sizes based on workload
    - Use meaningful thread name prefixes
    - Configure rejection policies
2. **Error Handling**:
    - Implement AsyncUncaughtExceptionHandler
    - Return CompletableFuture for better error propagation

- Use @Retryable for transient failures