Concurrency in Java

IN5170 - Models of Concurrency

University of Oslo, Norway

1 Introduction

Concurrency in Java allows multiple threads to execute simultaneously, enabling efficient use of resources and improving the performance of applications. Java provides built-in support for concurrency through the ‘java.lang.Thread‘ class and the ‘java.util.concurrent‘ package.

2 Threads

A thread is a lightweight process that can run concurrently with other threads. In Java, you can create a thread by extending the ‘Thread‘ class or implementing the ‘Runnable‘ interface. They are units of execution that can run concurrently, allowing for tasks to be performed simultaneously, as shown in Listing 1.1.

An application begins with a single thread, known as the main thread. You can create additional threads to perform tasks concurrently, which can improve the responsiveness and performance of your application.

Processes vs. threads (in Java)

- A process is an independent instance running on its own memory space.
- A thread runs inside a process and shares its resources with other threads

We here focus on threads, multi-process applications in Java are possible but not common. Both are handled by the operating system, but threads are more lightweight and efficient for concurrent tasks within a single application. Processes also have costly context switching, and communication between threads is easier than between processes

- Only processes require full cache flushes during context switches, which can be costly in terms of performance.
- Threads share the same memory space, allowing for easier communication and data sharing between them, while processes require inter-process communication (IPC) mechanisms, which can be more complex and less efficient.

Listing 1.1. Sample Java Code implementing Runnable

```java
class Printer implements Runnable {
    private String name;

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

    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println(name + " is printing: " + i);
            try {
                Thread.sleep(100); // Simulate time-consuming task
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }

    public static void main(String[] args) {
        Thread thread1 = new Thread(new Printer("Thread 1"));
        Thread thread2 = new Thread(new Printer("Thread 2"));

        thread1.start();
        thread2.start();
    }
}
```

The `Thread` class encapsulates a system thread and provides methods for managing its execution. The `Runnable` interface defines a single method, `run()`, which contains the code that will be executed by the thread. By implementing `Runnable`, you can create a thread that executes the code defined in the `run()` method.

Start vs run

- `Thread.start()` starts a new *concurrent* thread of execution.
- `Runnable.run()` executes the `run()` method in the current thread, not creating a new thread.
- `Thread.start()` calls the `run()` method internally, but it does so in a new thread, allowing for concurrent execution.
- Calling `Runnable.run()` directly will not create a new thread and will execute the code sequentially in the current thread, which is not the intended use of the `Runnable` interface for concurrent programming.

Common patterns for anonymous runnables include using lambda expressions or anonymous inner classes to create and start threads without needing to define a separate class that implements `Runnable`, as shown in Listing 1.2.

Listing 1.2. Sample Java Code implementing Runnable

```java
// Using a lambda expression
Thread thread1 = new Thread(() -> {
    for (int i = 0; i < 5; i++) {
        System.out.println("Thread 1 is printing: " + i);
        try {
            Thread.sleep(100); // Simulate time-consuming task
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
});

// Using an anonymous inner class
Thread thread2 = new Thread(new Runnable() {
    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("Thread 2 is printing: " + i);
            try {
                Thread.sleep(100); // Simulate time-consuming task
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }
});
```

Rather than declaring a class that implements `Runnable`, you can use a lambda expression or an anonymous inner class to create and start a thread in a more concise way. This approach is often preferred for simple tasks that do not require a separate class definition, making the code cleaner and easier to read.

Shared state

Shared states between threads is introduced through multiple means

- Static state (e.g., static variables), see Listing 1.3
- Shared references (e.g., shared objects), see Listing 1.4
- Resources (e.g., files, etc.), see Listing 1.5

The three examples below illustrate different ways in which shared state can be introduced in Java, leading to potential concurrency issues such as race conditions and unpredictable behavior when multiple threads access and modify shared resources concurrently.

Listing 1.3. Sample Java Code implementing Runnable

```
class C {
    private static int counter = 0;
}

new Thread(() -> {
    counter++; // Accessing and modifying the static variable 'counter'
}).start();
new Thread(() -> {
    counter++; // Accessing and modifying the static variable 'counter'
}).start();
```

As depicted in Listing 1.3, a variable declared as static means there is only one copy of it for the entire class becoming a shared variable. When multiple threads access and modify this static variable concurrently, it can lead to race conditions, where the outcome depends on the timing of the threads' execution. This can result in unpredictable behavior and bugs in the program, such as incorrect values being stored in the static variable or even crashes if the variable is accessed while it is being modified by another thread. In this regard, the counter increments are not atomic, and without proper synchronization, the final value of the counter may not reflect the total number of increments made by all threads.

Listing 1.4. Sample Java Code implementing Runnable

```
class C {
    private int counter = 0;
}
final C c = new C();

new Thread(() -> {
    c.counter++; // Accessing and modifying the instance variable 'counter'
}).start();
new Thread(() -> {
    c.counter++; // Accessing and modifying the instance variable 'counter'
}).start();
```

In Listing 1.4, what is final (immutable) is the reference to the object, not the object itself. In this example, the variable 'c' is declared as final, which means that it cannot be reassigned to point to a different instance of class C. However, the state of the object that 'c' references (i.e., the 'counter' variable) can still be modified by multiple threads concurrently.

Listing 1.5. Sample Java Code implementing Runnable

```
new Thread(() -> {
    new File("/path").delete();
}).start();
new Thread(() -> {
    new File("/path").delete();
```

```
}).start();
```

As shown in Listing 1.5, one could have two different links pointing to the same file, becoming a shared resource. If both threads attempt to delete the same file concurrently, it can lead to race conditions and unpredictable behavior. For example, one thread might successfully delete the file while the other thread is still trying to access it, resulting in a 'FileNotFoundException' or other errors.

Java offers static methods of `Thread` that refer to the current thread

- `Thread.currentThread()` is a static method that returns a reference to the currently executing thread, allowing you to access thread-specific information and perform operations on the current thread.
- `join()` is an instance method that allows one thread to wait for the completion of another thread. When a thread calls the `join()` method on another thread, it will block until the other thread has finished executing, ensuring that the calling thread does not proceed until the specified thread has completed its task.
- `sleep(long millis)` is a static method that causes the currently executing thread to pause its execution for a specified number of milliseconds. This can be used to simulate time-consuming tasks or to create delays in the execution of a thread.
- `yield()` is a static method that causes the currently executing thread to temporarily pause and allow other threads of the same priority to execute. It is a hint to the thread scheduler that the current thread is willing to yield its remaining time slice to other threads, but it does not guarantee that other threads will be scheduled immediately.

3 Synchronization

Java does not have atomic block constructs, but it provides synchronization mechanisms such as the 'synchronized' keyword and the 'java.util.concurrent' package to manage concurrent access to shared resources and ensure thread safety, as shown in Listing 1.6.

Listing 1.6. Sample Java Code with `synchronized` Methods

```
public class SynchronizedCounter {
    private int counter = 0;
    public synchronized void increment() { counter++; }
    public synchronized void decrement() { counter--; }
    public synchronized int getCounter() { return counter; }
}

SynchronizedCounter counter1 = new SynchronizedCounter();
SynchronizedCounter counter2 = new SynchronizedCounter();

Runnable runnable1 = () -> { counter1.increment(); };
```

```
Runnable runnable2 = () -> { counter1.increment(); };
Runnable runnable3 = () -> { counter1.decrement(); };
Runnable runnable4 = () -> { counter2.increment(); };
```

In Listing 1.7, the two calls will not execute at the same time in `counter1`

Listing 1.7. Sample Java Code with `synchronized` Methods

```
new Thread(runnable1).start();
new Thread(runnable2).start();
```

In Listing 1.8 the two calls will also not execute at the same time in `counter1`

Listing 1.8. Sample Java Code with `synchronized` Methods

```
new Thread(runnable1).start();
new Thread(runnable3).start();
```

In Listing 1.9 the call is to two *different* objects, so they will execute concurrently. —

Listing 1.9. Sample Java Code with `synchronized` Methods

```
new Thread(runnable1).start();
new Thread(runnable4).start();
```

Interleaving

Interleaving is the arrangement of operations from multiple concurrently executing threads into a single execution order, while preserving the order of operations within each individual thread. The `synchronized` keyword restricts possible interleavings by ensuring that only one thread at a time can execute a synchronized block or method protected by the same lock. Since each synchronized method acquires the lock associated with its own instance, threads operating on different instances do not block each other, as shown in Listing 1.9. The `synchronized` keyword does not remove concurrency or interleaving; it controls interleaving by making critical sections execute atomically with respect to other threads using the same lock.

3.1 Synchronized Static Methods

Synchronized static methods are **per class**. A synchronized static method locks the `Class` object, meaning that it affects all instances of the class rather than one particular instance. This means that if one thread is executing a synchronized static method, it will block all other threads from executing any synchronized static methods of the same class, regardless of which instance they are trying to access, as shown in Listing 1.10.

Listing 1.10. Sample Java Code with `synchronized` Static Methods

```
public class StaticSyncCounter {
    private static int counter = 0;
    public static synchronized void increment() { counter++; }
    public static synchronized void decrement() { counter--; }
    public static synchronized int getCounter() { return counter; }
}

Runnable s1 = () -> { StaticSyncCounter.increment(); };
Runnable s2 = () -> { StaticSyncCounter.decrement(); };
```

3.2 Synchronized Block

Synchronized blocks allows to synchronize only a part of a code rather than an entire method. One need to give the object from which the lock should be obtained — any object can be used as a lock, but it is common to use the object itself (i.e., `this`) or a dedicated lock object, as shown in Listing 1.11.

Listing 1.11. Sample Java Code with `synchronized` Blocks

```
public class C {
    int l = 0;
    int r = 0;
    void method (Object lock, boolean left) {
        synchronized (lock) {
            if (left) l++;
            else r++;
        }
    }
}

// The following will execute concurrently
C c = new C();
Object o1 = new Object();
Object o2 = new Object();

// thread 1
new Thread(() -> {
    c.method(o1, true);
}).start();
// thread 2
new Thread(() -> {
    c.method(o2, false);
}).start();
```

3.3 Atomic Expressions

JVM

Java is compiled down to JVM bytecode, which does not correspond to a 1-to-1 machine instructions. The JVM has no registers, but loads from a variable/heap onto a stack. All computations target the top values on the stack.

Given the execution `x++;` where `x` is a local variable declared as long. How many machine instructions does it compile to? The answer is 4-6. The reason is that the JVM bytecode for this operation consists of multiple instructions that load the value of `x` onto the stack, increment it, and then store the result back into `x`. The exact number of instructions may vary depending on the specific implementation of the JVM and the optimizations applied during compilation. The following is a possible sequence of JVM bytecode instructions for the operation `x++;`:

```
LLOAD_1  // push the value of x onto the stack
LCONST_1 // push the constant 1 onto the stack
LADD     // add the top two values on the stack (x + 1)
LSTORE_1 // store the result back into x
```

- Reference reads and writes are atomic
- Basic type reads and writes except `long` and `double` are atomic
- On 64-bit machines, `long` and `double` reads and writes might be atomic, might be two instructions
- Accessing variables modified by `volatile` is atomic

4 Weak Memory Model

Java Memory Model

The JVM defines a *weak* memory mode that allows certain reordering in read and write operations.

The memory model mainly target performance like cache optimisation. It can be done statically by the compiler, or dynamically by the CPU. The main consequence is that the order of operations in a thread might not be the same as the order of execution, and that threads might see different values for shared variables.

The memory model defines which reordering are allowed and which are not. In a **strong** memory model, no reordering is allowed, and all threads see the same values for shared variables. In a **weak** memory model, certain reordering are allowed, and threads might see different values for shared variables.

Independence

Reordering must take into account whether the operations are independent or not.

- `x := 1; r := x;` cannot be reordered because the second operation has a read-after-write dependency on the first one.
- `r := x; x := 1;` cannot be reordered because changing the order changes the value observed by the read operation.
- `x := 1; y := 1;` can be reordered because the two operations are independent.

Therefore, whether two operations can be reordered depends on whether the reordering preserves the program semantics. In particular:

- Independent operations may be reordered.
- Operations with data or control dependencies cannot be reordered if doing so changes the observable behavior.
- Accesses to `volatile` variables and synchronization operations impose additional ordering constraints.

For ordinary (non-synchronized) memory accesses, the Java Memory Model may permit reorderings, provided they do not affect the single-threaded semantics of the program. Furthermore, some architectures may also reorder other atomic operations. Reordering is possible when the operations refer to different memory locations and changing their order does not affect the observable behavior of the program.

Weak memory models can lead to very unintuitive results in concurrent settings.

Example:

Assume `P1` and `P2` execute concurrently

```
int x,y; //default 0
    P1:                              P2:
        x := 1; //shared variable        y := 1; //shared variable
        r1 := y; //register             r2 := x;
        print r1;                        print r2;
```

What are the possible outputs? Is 0, 0 possible?
If the read of `x` in `P2` is reordered, then 0,0 is possible, e.g., the compiler may decide to do such reordering in `P2` since locally the result will not change. This happens only under weak memory models, and is not possible under strong memory models. The output 0,0 is possible because the read of `x` in `P2` can be reordered to occur before the write to `y` in `P2`, allowing both threads to read the initial values of the shared variables before any writes

have occurred. A strong memory model, such as Sequential Consistency (SC), guarantees that:

- Operations within each thread appear in program order.
- There exists a single global order of all operations that respects each thread's order.

This output cannot be explained by reasoning about interleavings. If the language does not require variables to be initialised, we get out-of-thin-air values, which are values that are not produced by any thread but can be read by other threads. This can lead to very unintuitive results and is a consequence of the weak memory model.

Sequential Consistency

Most weak memory models guarantee **sequential consistency**: the result of execution is the same as if all the operations were executed in some sequential order, and the operations of each individual thread appear in this sequence in the order specified by the program. However, this does not mean that all threads see the same values for shared variables, as the order of operations can be different for different threads due to reordering.

In this matter, *volatile* can help to mitigate some of the challenges posed by weak memory models. A variable declared as `volatile` in Java ensures that a volatile read sees the most recent write to that volatile variable according to the Java Memory Model, and it also prevents certain types of reordering, as shown in Listing 1.12. When a thread writes to a volatile variable, it establishes a happens-before relationship with subsequent reads of that variable by other threads, ensuring visibility and ordering guarantees that can help to avoid some of the issues associated with weak memory models. Volatile guarantees visibility and ordering, but it does not provide atomicity for compound actions, such as incrementing a variable (`count++`). Therefore, while volatile can help ensure visibility and prevent certain instruction reorderings, it is not a substitute for proper synchronization when multiple operations on shared mutable state must be performed atomically.

Listing 1.12. Sample Java Code with `volatile`

```
1 public class C {
2     private volatile long l = 5;
3
4     long increment() {
5         return l++;
6     } // called from two different threads
7 }
```

- All read and write accesses to `l` are atomic
- All write accesses to `l` are **immediately** visible to all threads

- In terms of memory model, no reads and write to `l` can be reordered with respect to each other, but they can be reordered with respect to accesses to other variables
- In terms of memory, `l` is read and written from global memory, and not from thread-local caches, which means that all threads see the most up-to-date value of `l`
- It does **not** introduce synchronisation, but removes opportunities for optimisation and makes access more expensive, so it should be used only when necessary.

Weak memory is a complex topic, mostly relevant to low level architectures and compilers:

- Further operations: read-own-write-early, read-others-write-early
- Memory leaks might not be visible in sequential programs, but might become relevant to programmers in concurrent settings when dealing with weak memory models
- Hard to debug, most languages have no clearly defined memory model
- Often hardware-dependent solutions

Rough guideline on when to use `volatile`:

- If a field is **not supposed** to have data races, do **not** use `volatile`
- If a field is supposed to be accessed by multiple threads and you need to ensure visibility and ordering, consider using `volatile`

5 Java Libraries

- Java's standard library offers further data structures for common patterns or to encapsulate complex and efficient solutions
- It has thread-safe collections that are less efficient, but internally race-free version of collections
- Atomic classes encapsulate data with efficient, atomic access

5.1 Atomic Classes

Atomic classes are available for data and references. Operations that are atomic without synchronisation blocks are more efficient but can affect control flow. They also offer additional atomic operations that are not available with synchronisation, such as compare-and-set (CAS) operations, which allow for lock-free algorithms and can improve performance in certain concurrent scenarios. It is to consider, in this regard, the difference between `AtomicInteger` and `Integer`, where the former provides atomic operations for integer values as shown in Listing 1.13, while the latter does not guarantee atomicity as shown in Listing 1.14 and is not thread-safe when accessed by multiple threads concurrently.

Listing 1.13. Sample Java Code with `AtomicInteger`

```java
public class AtomicCounter {
    private java.util.concurrent.atomic.AtomicInteger counter = new java.
    util.concurrent.atomic.AtomicInteger(0);
    public void increment() { counter.incrementAndGet(); }
    public void decrement() { counter.decrementAndGet(); }
    public int getCounter() { return counter.get(); }
}
```

Listing 1.14. Sample Java Code with `int`

```java
public class SynchronizedCounter {
    private int counter = 0;
    public synchronized void increment() { counter++; }
    public synchronized void decrement() { counter--; }
    public synchronized int getCounter() { return counter; }
}
```

`AtomicReference` does **not** make the content of the reference atomic, but only the reference itself. This means that while you can atomically update the reference to point to a new object, the operations on the object that the reference points to are not atomic and can be accessed concurrently by multiple threads without any guarantees of thread safety.

Listing 1.15. Sample Java Code with `AtomicReference`

```java
AtomicReference<C> cache = new AtomicReference<>(new C());
cache.get();        // atomic
cache.get().m();    // the content of m is not executed atomically
```

5.2 Concurrent Collections

Concurrent Collections provide concurrent implementations that enable concurrent access

- `Map` vs `ConcurrentHashMap`
  - Most implementations of `Map` are not thread-safe, meaning that if multiple threads access and modify the map concurrently without proper synchronization, it can lead to unpredictable behavior and data corruption.
  - `ConcurrentHashMap` is designed to be thread-safe, allowing concurrent access and modifications by multiple threads without the need for external synchronization.
- `ConcurrentLinkedQueue` and variants for lists without random access
- `CopyOnWriteArrayList` makes an `ArrayList` thread-safe by creating a new copy of the underlying array for each modification, allowing for concurrent reads without locking, but it is not efficient for scenarios with frequent modifications due to the overhead of copying the entire array.

Synchronised Collections

Normal implementations, but add synchronised at the right places. They are less efficient than concurrent collections, but they can be used when you need to ensure that all operations on the collection are atomic and thread-safe, and when the performance overhead of synchronization is acceptable for your use case.

Listing 1.16. Sample Java Code with synchronised collections

```
<T> Collection<T> synchronizedCollection(Collection<T> c)

ArrayList<Object> a = new ArrayList<>();
Collection<Object> b = Collections.synchronizedCollection(a);
// access through the object a is still unsafe
```

Non-thread-safe implementations of collections, e.g. `Map` tend to be faster in single-threaded environments or when explicit synchronization is used to manage concurrent access, while thread-safe implementations like `ConcurrentHashMap` are designed to provide better performance in multi-threaded environments by allowing concurrent access and modifications without the need for external synchronization.

Thread Management

In bigger applications, it is common to manage sets of threads. We can do so with

1. Lifecycle management of a thread object
2. Interrupts
3. Thread pools

5.3 Lifecycle

The lifecycle of a thread in Java consists of several states, including:

- **New**: The thread is created but has not yet started.
- **Runnable**: The thread is ready to run and is waiting for the CPU to schedule.
- **Running**: The thread is currently executing.
- **Blocked**: The thread is waiting for a monitor lock to enter a synchronized block or method.
- **Waiting**: The thread is waiting indefinitely for another thread to perform a specific action (e.g., calling ‘wait()‘).
- **Timed Waiting**: The thread is waiting for a specified amount of time (e.g., calling ‘sleep()‘ or ‘wait(timeout)‘).
- **Terminated**: The thread has completed its execution or has been terminated and considered dead. Once a thread is in the terminated state, it cannot be restarted.

5.4 Interrupts

An interrupt is a signal sent to a thread to indicate that it should stop what it is doing and do something else. In Java, you can interrupt a thread by calling the `interrupt()` method on the thread object. When a thread is interrupted, it will set an internal flag that can be checked by the thread itself or by other threads to determine if it has been interrupted.

When a thread is interrupted, it can respond in different ways depending on its current state:

- If the thread is in a blocked state (e.g., waiting for a monitor lock), it will throw an `InterruptedException` and exit the blocked state.
- If the thread is in a waiting state (e.g., waiting indefinitely for another thread to perform a specific action), it will also throw an `InterruptedException` and exit the waiting state.
- If the thread is in a timed waiting state (e.g., sleeping for a specified amount of time), it will throw an `InterruptedException` and exit the timed waiting state.
- If the thread is running, the `isInterrupted()` method can check the interrupt status and decide how to respond, such as gracefully shutting down or performing cleanup operations before exiting.

Listing 1.17. Sample Java Code with interrupt handling

```
() -> {
    while (!Thread.currentThread().isInterrupted()) {
        // Perform some work
    }
    // Clean up and exit
}
```

5.5 Thread Pools

A thread pool is a collection of worker threads that are created and managed by a thread pool executor. In Java, you can create a thread pool using the `java.util.concurrent.ExecutorService` interface and its implementations, such as `ThreadPoolExecutor` or `Executors`. An `ExecutorService` allows you to submit tasks for execution and manages the lifecycle of the threads in the pool, including thread creation, scheduling, and termination, as shown in Listing 1.19.

Listing 1.18. Sample Java Code with thread pool

```
ExecutorService executor = Executors.newFixedThreadPool(10);
executor.submit(() -> {
    // Task to be executed by a thread in the pool
});
```

To join on threads in a thread pool, you can use the `Future` objects returned by the `submit()` method. You can call the `get()` method on a `Future` to block until the task is completed and retrieve the result, as in Listing 1.19.

Futures

Futures are a way to represent the result of an asynchronous computation that may not yet be available. In Java, the `Future` interface represents the pending result of an asynchronous computation and provides methods to check whether the computation has completed, retrieve its result, and cancel the computation if needed. When a task is submitted to an `ExecutorService`, it returns a `Future` object that can be used to monitor the task's status and obtain its result once the computation has finished.

Listing 1.19. Sample Java Code with Future

```
ExecutorService executor = Executors.newFixedThreadPool(10);
Future<Integer> future = executor.submit(() -> {
    // Task to be executed by a thread in the pool
    return 42;
});
int result = future.get(); // Blocks until the task is completed
```

Thread pools have further capabilities, like `invokeAll()` to submit a collection of tasks and wait for their completion, or `shutdown()` to gracefully shut down the thread pool and stop accepting new tasks while allowing existing tasks to complete.

5.6 Completable Futures

Completable futures[1] are a feature in Java that allows for asynchronous programming and handling of future results. They provide a way to represent a value that may not yet be available, allowing you to write non-blocking code that can handle the result of an asynchronous computation once it is completed.

We can create a completable future using the `CompletableFuture` class, which provides methods for creating, combining, and handling asynchronous computations. You can use methods like `supplyAsync()` to create a new completable future that runs a task asynchronously, and `thenApply()` or `thenAccept()` to specify actions to be performed once the future is completed.

The `CompletableFuture` class implements the `Future` interface so that we can use it as a Future implementation but with additional completion logic. We can create an instance of a class to represent some future result, hand it out to the consumers, and complete it at some time in the future using the complete method.

[1] https://www.baeldung.com/java-completablefuture

`CompletableFuture` has several methods for combining multiple futures, such as `thenCombine()`, `thenCompose()`, and `allOf()`, which allow you to create complex asynchronous workflows by chaining and combining multiple futures together.

Futures vs Completable Futures

As mentioned before, futures are a way to represent the result of an asynchronous computation that may not yet be available. In contrast, completable futures provide a more powerful and flexible way to handle asynchronous computations, allowing for non-blocking code and the ability to chain and combine multiple futures together. Completable futures also provide additional methods for handling exceptions and timeouts, making them a more robust solution for managing asynchronous tasks in Java.

We now present two examples of using thread pools and completable futures to perform parallel computations on an array of integers, constructing new arrays with the squares and cubes of the integers.

Example:

Consider an array of integers. Using thread pools, construct two new arrays, one with the squares of the integers and one with the cubes of the integers.

Listing 1.20. Example Java Code using Futures

```
1  import java.util.concurrent.*;
2  import java.util.*;
3
4  class ParallelComputation {
5      public static void main(String[] args) throws InterruptedException,
         ExecutionException {
6          int[] numbers = {1, 2, 3, 4, 5, 6};
7          List<Integer> evenNumbers = new ArrayList<>();
8          List<Integer> oddNumbers = new ArrayList<>();
9
10         ExecutorService executor = Executors.newFixedThreadPool(20);
11
12         // Add the numbers to the respective lists using threads
13         for (int number : numbers) {
14             executor.submit(() -> {
15                 if (number % 2 == 0) { evenNumbers.add(number); }
16                 else { oddNumbers.add(number); }
17             });
18         }
19
20         Future<List<Integer>> squaresFuture = executor.submit(() -> {
21             List<Integer> squares = new ArrayList<>();
22             for (int n : evenNumbers) { squares.add(n * n); }
```

```
            return squares;
        });

        Future<List<Integer>> cubesFuture = executor.submit(() -> {
            List<Integer> cubes = new ArrayList<>();
            for (int n : oddNumbers) { cubes.add(n * n * n); }
            return cubes;
        });

        List<Integer> squares = squaresFuture.get();
        List<Integer> cubes = cubesFuture.get();

        System.out.println("Squares: " + squares);
        System.out.println("Cubes: " + cubes);

        executor.shutdown();
    }
}
```

Example:

Consider an array of integers. Using completable futures, construct two new arrays, one with the squares of the integers and one with the cubes of the integers.

Listing 1.21. Example Java Code using Completable Futures

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

class ParallelComputation {
    public static void main(String[] args) throws InterruptedException,
      ExecutionException {
        int[] numbers = {1, 2, 3, 4, 5, 6};
        List<Integer> evenNumbers = new ArrayList<>();
        List<Integer> oddNumbers = new ArrayList<>();

        // Add the numbers to the respective lists using CompletableFutures
        for (int number : numbers) {
            CompletableFuture.runAsync(() -> {
                if (number % 2 == 0) { evenNumbers.add(number); }
                else { oddNumbers.add(number); }
            });
        }

        CompletableFuture<List<Integer>> squaresFuture =
      CompletableFuture.supplyAsync(() -> {
            List<Integer> squares = new ArrayList<>();
```

```
20            for (int n : evenNumbers) { squares.add(n * n); }
21            return squares;
22        });
23
24        CompletableFuture<List<Integer>> cubesFuture = CompletableFuture.
      supplyAsync(() -> {
25            List<Integer> cubes = new ArrayList<>();
26            for (int n : oddNumbers) { cubes.add(n * n * n); }
27            return cubes;
28        });
29
30        List<Integer> squares = squaresFuture.get();
31        List<Integer> cubes = cubesFuture.get();
32
33        System.out.println("Squares: " + squares);
34        System.out.println("Cubes: " + cubes);
35    }
36 }
```

Useful Links

- Class Executors: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executors.html
- Different thread pools: https://www.baeldung.com/java-executors-cached-fixed-threadpool
- Concurrency: https://docs.oracle.com/javase/tutorial/essential/concurrency/
- Concurrent Package: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html
- Completable Future: https://www.baeldung.com/java-completablefuture