PROGRAMMING IN JAVA

Lecture 14

Multithreading and Concurrent Programming

Threads, `Runnable`, Synchronization, Atomic Variables,
Locks, Executors, Futures, Deadlocks, and Virtual Threads

Lecturer: Arshyn Altybay

Department of Computer Engineering,
Almaty Technological University, Almaty, Kazakhstan

1. Learning Objectives

After completing this lecture, students should be able to:

1. explain the difference between a process and a thread;
2. describe why multithreading is useful;
3. create threads by extending `Thread`;
4. create tasks by implementing `Runnable`;
5. explain the difference between `start()` and `run()`;
6. explain important thread lifecycle states;
7. use `sleep()` and `join()`;
8. identify race conditions;
9. use the `synchronized` keyword;
10. explain mutual exclusion and critical sections;
11. use atomic variables such as `AtomicInteger`;
12. use explicit locks such as `ReentrantLock`;
13. explain deadlock and basic prevention strategies;
14. use `ExecutorService` and thread pools;
15. use `Callable` and `Future`;
16. use `CompletableFuture` for asynchronous composition;
17. explain the purpose of virtual threads in modern Java.

2. Introduction

Many programs perform one operation at a time.

This is called sequential execution.

For example:

Illustration

```
Task A
  |
Task B
  |
Task C
```

In concurrent programming, several tasks may make progress during overlapping periods of time.

Examples include:

- a GUI responding to the user while loading data;
- a web server processing several requests;
- downloading several files;
- performing background calculations;
- monitoring several devices;
- processing independent tasks in parallel.

Java provides extensive support for concurrency.

3. Process vs Thread

A **process** is an executing program with its own resources.

A **thread** is a path of execution inside a process.

A Java application may contain several threads that share the same process memory.

Conceptually:

Illustration

```
Java Process
  |
  +-- Thread 1
```

```
|
+-- Thread 2
|
+-- Thread 3
```

Threads are lighter-weight than independent operating-system processes.

4. The Main Thread

Every ordinary Java application begins with a thread that executes:

```
public static void main(String[] args)
```

The current thread can be obtained using:

```
Thread.currentThread()
```

Example 1. Inspecting the Current Thread

```
1  public class Main {
2
3      public static void main(String[] args) {
4
5          Thread current =
6                  Thread.currentThread();
7
8          System.out.println(
9                  "Name: " +
10                 current.getName()
11         );
12
13         System.out.println(
14                 "Alive: " +
15                 current.isAlive()
16         );
17     }
18 }
```

A typical output includes the thread name:

Illustration

```
Name: main
Alive: true
```

5. Creating a Thread by Extending `Thread`

One way to create a thread is to extend `Thread` and override `run()`.

Example 2. Extending `Thread`

```
1  public class Main {
2
3      static class Worker extends Thread {
4
5          @Override
6          public void run() {
7
8              for (int i = 1;
9                   i <= 3;
10                  i++) {
11
12                 System.out.println(
13                     "Worker: " + i
14                 );
15             }
16         }
17     }
18
19     public static void main(String[] args) {
20
21         Worker worker =
22                 new Worker();
23
24         worker.start();
25
26         System.out.println(
27                 "Main continues"
28         );
29     }
30 }
```

The exact order of output is not guaranteed.

Possible output:

Illustration

```
Main continues
Worker: 1
Worker: 2
Worker: 3
```

6. `start()` vs `run()`

This distinction is fundamental.

Calling:

```
thread.start();
```

asks the JVM to start a new thread of execution.

Calling:

```
thread.run();
```

is just a normal method call in the current thread.

Therefore:

Use `start()` when you want concurrent thread execution.

7. Creating Tasks with `Runnable`

A more flexible approach is to separate the task from the thread.

The `Runnable` interface contains one method:

```
void run();
```

A class can implement `Runnable`.

Example 3. Implementing `Runnable`

```
1  public class Main {
2
3      static class Task
4              implements Runnable {
5
6          @Override
7          public void run() {
8
9              System.out.println(
10                 "Task executed by " +
11                 Thread.currentThread()
12                       .getName()
13             );
14         }
15     }
16
17     public static void main(String[] args) {
18
19         Thread thread =
20                 new Thread(
21                     new Task()
22                 );
23
24         thread.start();
25     }
26 }
```

8. Runnable with Lambda Expressions

Because `Runnable` is a functional interface, it can be implemented with a lambda.

Example 4. Thread with Lambda

```
public class Main {

    public static void main(String[] args) {

        Runnable task = () -> {

            for (int i = 1;
                 i <= 3;
                 i++) {

                System.out.println(
                    Thread.currentThread()
                          .getName()
                    + ": " + i
                );
            }
        };

        Thread t1 =
                new Thread(
                    task,
                    "Worker-1"
                );

        Thread t2 =
                new Thread(
                    task,
                    "Worker-2"
                );

        t1.start();
        t2.start();
    }
}
```

The two threads may interleave their outputs differently on different runs.

9. Thread Lifecycle

Java represents thread states with `Thread.State`.

Important states include:

- `NEW`
- `RUNNABLE`
- `BLOCKED`
- `WAITING`
- `TIMED_WAITING`
- `TERMINATED`

A simplified lifecycle is:

Illustration

```
NEW
 |
start()
 |
RUNNABLE
 |
running / waiting / blocked
 |
TERMINATED
```

The exact scheduling is controlled by the runtime and operating system.

10. Thread.sleep()

The method:

```
Thread.sleep(milliseconds);
```

temporarily pauses the current thread.

It throws `InterruptedException`.

Example 5. Sleeping Thread

```
public class Main {

    public static void main(String[] args) {

        for (int i = 3;
             i >= 1;
             i--) {

            System.out.println(i);

            try {

                Thread.sleep(1000);

            } catch (
                    InterruptedException e) {

                Thread.currentThread()
                      .interrupt();

                System.out.println(
                    "Interrupted"
                );

                break;
            }
        }

        System.out.println("Start!");
    }
}
```

11. Interruption

Interruption is a cooperative mechanism for requesting that a thread stop what it is doing or respond to cancellation.

A thread can be interrupted using:

```
thread.interrupt();
```

Blocking methods such as `sleep()` may throw `InterruptedException`.

A common good practice after catching an interruption that cannot be fully handled is to restore the interruption status:

```
Thread.currentThread().interrupt();
```

12. join()

The method `join()` causes one thread to wait for another thread to finish.

Example 6. Waiting with `join()`

```
public class Main {

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

        Thread worker =
                new Thread(
                    () -> {

                        for (int i = 1;
                             i <= 3;
                             i++) {

                            System.out.println(
                                "Working " + i
                            );
                        }
                    }
                );

        worker.start();

        worker.join();

        System.out.println(
                "Worker finished."
        );
    }
}
```

The final message appears only after the worker thread terminates.

13. Shared Mutable State

Concurrency becomes more difficult when several threads access the same mutable data.

Example:

```
int counter = 0;
```

If several threads execute:

```
counter++;
```

the result may be incorrect.

Why?

Because increment is conceptually composed of several steps:

Illustration

```
read current value
add 1
write new value
```

Two threads may interfere with each other.

14. Race Conditions

A **race condition** occurs when the correctness of a program depends on uncontrolled timing or interleaving of concurrent operations.

Example 7. Race Condition

```
1  public class Main {
2
3      static class Counter {
4
5          int value = 0;
6
7          void increment() {
8              value++;
9          }
10     }
11
```

Example 7 (continued)

```
12      public static void main(String[] args)
13              throws InterruptedException {
14
15          Counter counter =
16                  new Counter();
17
18          Runnable task = () -> {
19
20              for (int i = 0;
21                   i < 100000;
22                   i++) {
23
24                  counter.increment();
25              }
26          };
27
28          Thread t1 =
29                  new Thread(task);
30
31          Thread t2 =
32                  new Thread(task);
33
34          t1.start();
35          t2.start();
36
37          t1.join();
38          t2.join();
39
40          System.out.println(
41                  "Expected: 200000"
42          );
43
44          System.out.println(
45                  "Actual: " +
46                  counter.value
47          );
48      }
49  }
```

The actual result may be smaller than 200000.

15. Critical Sections

A **critical section** is a part of code that accesses shared mutable state and must not be executed by several threads simultaneously when doing so would violate correctness.

Java provides several synchronization mechanisms.

The most basic is the `synchronized` keyword.

16. Synchronized Methods

Example:

```
synchronized void increment() {
    value++;
}
```

For an instance method, synchronization uses the current object as the monitor lock.

Only one thread at a time can execute synchronized code guarded by that same monitor.

Example 8. Synchronized Counter

```
1  public class Main {
2
3      static class Counter {
4
5          private int value = 0;
6
7          public synchronized
8          void increment() {
9
10             value++;
11         }
12
13         public synchronized
14         int getValue() {
15
16             return value;
17         }
18     }
19
20     public static void main(String[] args)
21             throws InterruptedException {
```

Example 8 (continued)

```
22
23          Counter counter =
24                  new Counter();
25
26          Runnable task = () -> {
27
28              for (int i = 0;
29                   i < 100000;
30                   i++) {
31
32                  counter.increment();
33              }
34          };
35
36          Thread t1 =
37                  new Thread(task);
38
39          Thread t2 =
40                  new Thread(task);
41
42          t1.start();
43          t2.start();
44
45          t1.join();
46          t2.join();
47
48          System.out.println(
49                  counter.getValue()
50          );
51      }
52  }
```

Expected output

200000

17. Synchronized Blocks

Instead of synchronizing an entire method, a smaller block may be protected.

```
synchronized (lockObject) {
    // critical section
}
```

This can reduce unnecessary synchronization when only part of a method accesses shared state.

18. Mutual Exclusion

Mutual exclusion means that only one thread at a time is allowed to execute a protected critical section associated with a particular lock.

Synchronization helps protect invariants such as:

$$\text{balance} \geq 0$$

or:

$$\text{counter value is updated without lost increments}$$

19. Visibility

Concurrency is not only about simultaneous writes.

One thread may also need to see updates performed by another thread.

Synchronization mechanisms establish important memory-visibility guarantees.

Without suitable synchronization, assuming that one thread immediately sees another thread's writes can be incorrect.

20. The `volatile` Keyword

A `volatile` field provides visibility guarantees for reads and writes of that field.

Example:

```
private volatile boolean running = true;
```

However:

> **`volatile` does not make compound operations such as `counter++` atomic.**

21. Atomic Variables

The package:

Illustration

```
java.util.concurrent.atomic
```

contains classes for atomic operations.

Important examples:

- `AtomicInteger`
- `AtomicLong`
- `AtomicBoolean`
- `AtomicReference`

Example 9. `AtomicInteger`

```
1  import java.util.concurrent.atomic.AtomicInteger;
2
3  public class Main {
4
5      public static void main(String[] args)
6              throws InterruptedException {
7
8          AtomicInteger counter =
9                  new AtomicInteger(0);
10
11         Runnable task = () -> {
12
```

Example 9 (continued)

```
13              for (int i = 0;
14                   i < 100000;
15                   i++) {
16
17                  counter.incrementAndGet();
18              }
19          };
20
21          Thread t1 =
22                  new Thread(task);
23
24          Thread t2 =
25                  new Thread(task);
26
27          t1.start();
28          t2.start();
29
30          t1.join();
31          t2.join();
32
33          System.out.println(
34                  counter.get()
35          );
36      }
37  }
```

Expected output

```
200000
```

22. Explicit Locks

Java also provides explicit lock objects.

A commonly used class is:

```
ReentrantLock
```

Typical structure:

```
lock.lock();

try {
    // critical section
} finally {
    lock.unlock();
}
```

The `finally` block is essential so that the lock is released even when an exception occurs.

Example 10. `ReentrantLock`

```
1  import java.util.concurrent.locks.Lock;
2  import java.util.concurrent.locks.ReentrantLock;
3
4  public class Main {
5
6      static class Counter {
7
8          private int value = 0;
9
10         private final Lock lock =
11                 new ReentrantLock();
12
13         public void increment() {
14
15             lock.lock();
16
17             try {
18
19                 value++;
20
21             } finally {
22
23                 lock.unlock();
24             }
25         }
26
27         public int getValue() {
28             return value;
29         }
30     }
31
32     public static void main(String[] args)
33             throws InterruptedException {
34
```

Example 10 (continued)

```
35          Counter counter =
36                  new Counter();
37
38          Runnable task = () -> {
39
40              for (int i = 0;
41                   i < 100000;
42                   i++) {
43
44                  counter.increment();
45              }
46          };
47
48          Thread t1 =
49                  new Thread(task);
50
51          Thread t2 =
52                  new Thread(task);
53
54          t1.start();
55          t2.start();
56
57          t1.join();
58          t2.join();
59
60          System.out.println(
61                  counter.getValue()
62          );
63      }
64  }
```

23. Deadlock

A **deadlock** occurs when threads wait permanently for one another.

Example situation:

Illustration

```
Thread 1 owns Lock A
Thread 1 waits for Lock B

Thread 2 owns Lock B
```

```
Thread 2 waits for Lock A
```

Neither thread can continue.

The following code demonstrates a dangerous locking pattern.

Example 11. Deadlock Pattern

```
public class Main {

    private static final Object lockA =
            new Object();

    private static final Object lockB =
            new Object();

    public static void main(String[] args) {

        Thread t1 =
            new Thread(() -> {

                synchronized (lockA) {

                    synchronized (lockB) {

                        System.out.println(
                            "Thread 1 finished"
                        );
                    }
                }
            });

        Thread t2 =
            new Thread(() -> {

                synchronized (lockB) {

                    synchronized (lockA) {

                        System.out.println(
                            "Thread 2 finished"
                        );
                    }
                }
            });

        t1.start();
```

Example 11 (continued)

```
40          t2.start();
41      }
42  }
```

Depending on timing, the program can deadlock.

24. Deadlock Prevention

Common strategies include:

1. acquire multiple locks in a consistent global order;
2. keep lock scope small;
3. avoid unnecessary nested locking;
4. avoid calling unknown external code while holding a lock;
5. use higher-level concurrent utilities;
6. use timed lock acquisition when appropriate.

25. Thread Pools

Creating a new platform thread for every small task can be inefficient.

A thread pool contains reusable worker threads.

Tasks are submitted to the pool and executed by available workers.

Java provides the `Executor` framework in:

Illustration

```
java.util.concurrent
```

26. ExecutorService

`ExecutorService` manages task execution and termination.

A fixed thread pool can be created with:

```
Executors.newFixedThreadPool(4)
```

Tasks can be submitted using:

```
executor.submit(task);
```

Modern `ExecutorService` also implements `AutoCloseable`, so it can be used in try-with-resources on current Java versions.

Example 12. Fixed Thread Pool

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class Main {

    public static void main(String[] args) {

        try (ExecutorService executor =
                Executors
                    .newFixedThreadPool(3)) {

            for (int i = 1;
                 i <= 6;
                 i++) {

                int taskId = i;

                executor.submit(() -> {

                    System.out.println(
                        "Task " +
                        taskId +
                        " -> " +
                        Thread.currentThread()
                              .getName()
                    );
                });
            }
        }
    }
}
```

The exact task-to-thread assignment is not guaranteed.

27. Callable

`Runnable` does not return a result.

`Callable<V>` can return a result and may throw checked exceptions.

Its method is:

```
V call() throws Exception;
```

28. Future

When a `Callable` is submitted to an executor, the result can be represented by a `Future<V>`.

Important methods include:

- `get()`
- `isDone()`
- `cancel()`
- `isCancelled()`

Example 13. `Callable` and `Future`

```
1  import java.util.concurrent.Callable;
2  import java.util.concurrent.ExecutorService;
3  import java.util.concurrent.Executors;
4  import java.util.concurrent.Future;
5
6  public class Main {
7
8      public static void main(String[] args)
9              throws Exception {
10
11         try (ExecutorService executor =
12                 Executors
13                     .newFixedThreadPool(2)) {
14
15             Callable<Integer> task =
16                     () -> {
17
18                 int sum = 0;
19
20                 for (int i = 1;
21                      i <= 100;
22                      i++) {
23
24                     sum += i;
25                 }
26
27                 return sum;
```

Example 13 (continued)

```
            };

            Future<Integer> future =
                    executor.submit(task);

            System.out.println(
                "Result = " +
                future.get()
            );
        }
    }
}
```

Expected output

```
Result = 5050
```

29. Concurrent Collections

Java provides thread-safe concurrent collections such as:

- `ConcurrentHashMap`
- `CopyOnWriteArrayList`
- `BlockingQueue`

These are often safer and more scalable than manually synchronizing ordinary collections.

Example 14. `ConcurrentHashMap`

```
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ConcurrentMap;

public class Main {

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

        ConcurrentMap<String, Integer>
            scores =
            new ConcurrentHashMap<>();

        Runnable task1 = () ->
```

Example 14 (continued)

```
14              scores.put("Java", 90);
15
16          Runnable task2 = () ->
17              scores.put("Databases", 85);
18
19          Thread t1 =
20                  new Thread(task1);
21
22          Thread t2 =
23                  new Thread(task2);
24
25          t1.start();
26          t2.start();
27
28          t1.join();
29          t2.join();
30
31          System.out.println(scores);
32      }
33  }
```

The order of map entries is not guaranteed.

30. CompletableFuture

`CompletableFuture<T>` represents a future result that can also participate in a chain of dependent computations.

It implements both:

- `Future<T>`
- `CompletionStage<T>`

Useful methods include:

- `supplyAsync()`
- `runAsync()`
- `thenApply()`
- `thenAccept()`
- `thenCompose()`
- `thenCombine()`
- `exceptionally()`

- `join()`

Example 15. `CompletableFuture`

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

public class Main {

    public static void main(String[] args) {

        CompletableFuture<Integer>
            future =
            CompletableFuture
                .supplyAsync(() -> 20)
                .thenApply(
                    value -> value * 2
                )
                .thenApply(
                    value -> value + 5
                );

        System.out.println(
            future.join()
        );
    }
}
```

Expected output

```
45
```

31. Handling Asynchronous Errors

A `CompletableFuture` pipeline can recover from an exceptional completion.

Example:

```
CompletableFuture<Integer>
    future =
    CompletableFuture
        .supplyAsync(() -> 10 / 0)
        .exceptionally(
            error -> 0
        );
```

This can separate error-recovery logic from the successful pipeline.

32. Parallelism vs Concurrency

These terms are related but not identical.

Concurrency means multiple tasks can make progress during overlapping time intervals.

Parallelism means multiple computations are literally executing at the same time, typically on multiple processor cores.

A concurrent program may run on one core through interleaving.

A parallel program uses simultaneous execution to speed up computation.

33. CPU-Bound vs I/O-Bound Tasks

A **CPU-bound** task spends most of its time calculating.

Examples:

- numerical simulation;
- image processing;
- large matrix computation.

An **I/O-bound** task spends much of its time waiting.

Examples:

- network requests;
- database calls;
- file access;
- remote service calls.

This distinction affects concurrency design.

34. Modern Java: Virtual Threads

Modern Java supports two broad kinds of `Thread` objects:

- platform threads;
- virtual threads.

Virtual threads became a final Java feature in JDK 21.

They are lightweight threads managed primarily by the Java runtime and are especially suitable for applications with large numbers of tasks that spend much of their time blocked on I/O.

They improve scalability and throughput for many thread-per-task workloads.

They are not intended to make CPU-intensive computation itself faster.

Example 16. Creating a Virtual Thread

```
public class Main {

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

        Thread thread =
            Thread.ofVirtual()
                  .name("virtual-worker")
                  .start(() -> {

                      System.out.println(
                          "Running in: " +
                          Thread.currentThread()
                                .getName()
                      );
                  });

        thread.join();
    }
}
```

This example requires a modern Java release that supports virtual threads.

35. Virtual-Thread Executor

Modern Java also provides:

```
Executors.newVirtualThreadPerTaskExecutor()
```

It creates a new virtual thread for each submitted task.

Example 17. Virtual-Thread-Per-Task Executor

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class Main {

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

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

            Future<String> result =
                executor.submit(() -> {

                    Thread.sleep(200);

                    return "Completed by " +
                        Thread.currentThread();
                });

            System.out.println(
                result.get()
            );
        }
    }
}
```

Virtual threads are useful particularly for high-concurrency blocking workloads, not for replacing careful synchronization of shared mutable state.

36. Thread Safety

A class is **thread-safe** when it behaves correctly when used by multiple threads according to its documented contract.

Strategies for achieving thread safety include:

- immutability;
- avoiding shared mutable state;
- confinement to one thread;
- synchronization;
- atomic variables;
- concurrent collections;
- message-passing designs.

37. Immutability and Concurrency

Immutable objects are naturally easier to share between threads because their state does not change after construction.

For example:

```
final class Point {

    private final int x;
    private final int y;

    // constructor and getters
}
```

No synchronization is required to protect state that never changes.

38. Common Mistakes

38.1 Mistake 1: Calling `run()` Instead of `start()`

Calling `run()` directly does not start a new thread.

38.2 Mistake 2: Assuming Output Order

Concurrent thread scheduling is not deterministic.

38.3 Mistake 3: Sharing Mutable State Without Coordination

This can create race conditions and visibility problems.

38.4 Mistake 4: Using `volatile` for Compound Atomic Operations

`volatile` does not make `counter++` atomic.

38.5 Mistake 5: Forgetting to Release a Lock

Always release explicit locks in a `finally` block.

38.6 Mistake 6: Creating Deadlocks with Inconsistent Lock Ordering

Acquire multiple locks in a consistent order.

38.7 Mistake 7: Forgetting Executor Lifecycle Management

Executors should be closed or shut down appropriately.

38.8 Mistake 8: Blocking the Swing Event Dispatch Thread

Long-running concurrent work should be moved away from the GUI event thread.

39. Design Guidelines

When writing concurrent Java programs:

1. avoid shared mutable state whenever practical;
2. prefer immutable data;
3. separate tasks from thread-management details;
4. prefer executors for groups of tasks;
5. use concurrent utilities instead of low-level locking when possible;
6. keep critical sections small;
7. document thread-safety expectations;
8. preserve interruption status when appropriate;
9. avoid relying on timing assumptions;
10. use consistent lock ordering;
11. distinguish CPU-bound from I/O-bound workloads;
12. use virtual threads mainly for high-concurrency blocking workloads.

40. Summary

In this lecture, we learned that:

- threads allow several tasks to make progress concurrently;
- `Thread` and `Runnable` are fundamental Java concurrency tools;
- `start()` creates concurrent execution while `run()` is a normal call;
- `sleep()` pauses the current thread;
- `join()` waits for another thread to finish;
- race conditions occur with unsafe shared mutable state;
- `synchronized` protects critical sections;
- `volatile` provides visibility but not compound-operation atomicity;
- atomic variables support lock-free atomic operations for suitable cases;
- `ReentrantLock` provides explicit locking;
- deadlocks arise from circular waiting;
- `ExecutorService` manages task execution;
- `Callable` can return values through `Future`;
- concurrent collections are designed for shared concurrent access;
- `CompletableFuture` supports asynchronous composition;
- virtual threads support scalable blocking concurrency in modern Java.

41. Review Questions

1. What is a process?
2. What is a thread?
3. What is multithreading?
4. What is the main thread?
5. How can a class extend `Thread`?
6. What is `Runnable`?
7. Why is `Runnable` often preferable to subclassing `Thread`?
8. What is the difference between `start()` and `run()`?
9. Name important thread states.
10. What does `Thread.sleep()` do?
11. What is thread interruption?
12. What does `join()` do?
13. What is shared mutable state?
14. What is a race condition?
15. What is a critical section?
16. What does `synchronized` do?
17. What is mutual exclusion?
18. What does `volatile` provide?
19. Why does `volatile` not make `counter++` atomic?
20. What is `AtomicInteger`?
21. What is `ReentrantLock`?
22. Why should `unlock()` normally appear in a `finally` block?
23. What is a deadlock?
24. How can deadlocks be reduced or prevented?
25. What is a thread pool?
26. What is `ExecutorService`?
27. What is `Callable`?
28. What is `Future`?
29. What is `ConcurrentHashMap`?
30. What is `CompletableFuture`?
31. What is the difference between concurrency and parallelism?
32. What is a CPU-bound task?
33. What is an I/O-bound task?
34. What is a virtual thread?
35. In which Java release were virtual threads finalized?

36. What workloads benefit most from virtual threads?
37. What is thread safety?
38. Why are immutable objects useful in concurrent programs?

42. Programming Exercises

Beginner Level

1. Print information about the current thread.
2. Create a thread by extending `Thread`.
3. Create a task by implementing `Runnable`.
4. Create a thread using a lambda expression.
5. Start two threads that print different messages.
6. Use `sleep()` to create a countdown.
7. Use `join()` to wait for a worker.
8. Give custom names to three threads.
9. Print thread states before and after execution.
10. Demonstrate the difference between calling `run()` and `start()`.

Intermediate Level

1. Demonstrate a race condition with a shared counter.
2. Fix the counter using a synchronized method.
3. Fix the counter using a synchronized block.
4. Fix the counter using `AtomicInteger`.
5. Protect a bank-account balance using synchronization.
6. Use `ReentrantLock` to protect shared data.
7. Create a fixed thread pool and submit ten tasks.
8. Use `Callable` to calculate the sum of a numeric range.
9. Use several `Future` objects to collect results.
10. Use `ConcurrentHashMap` from several threads.
11. Create a simple `CompletableFuture` pipeline.
12. Recover from a failed `CompletableFuture`.

Advanced Level

1. Build a multithreaded bank-transfer simulation.
2. Create a deadlock example and then redesign it with consistent lock ordering.
3. Create a parallel file-processing application using an executor.
4. Create a task scheduler using `ScheduledExecutorService`.
5. Build a producer-consumer example using a `BlockingQueue`.
6. Build a concurrent student-statistics processor.
7. Use `CompletableFuture` to combine two independent calculations.
8. Compare a fixed thread pool with a virtual-thread-per-task executor for simulated blocking tasks.
9. Build a Swing application that performs long work without blocking the EDT.
10. Design a thread-safe in-memory repository.

43. Homework

Task: Develop a Concurrent Student Result Processing System

Create the following classes:

Illustration

```
Student
StudentResult
ResultTask
ResultProcessor
Main
```

Requirements:

1. Create at least 20 students.
2. Each student must have several assessment scores.
3. Use an `ExecutorService` to process several students concurrently.
4. Each task must calculate:
   - average score;
   - final percentage;
   - letter grade.
5. Use `Callable<StudentResult>`.
6. Collect results using `Future<StudentResult>`.
7. Store completed results in a thread-safe collection or aggregate them safely.
8. Calculate overall class statistics.
9. Demonstrate at least one use of:
   - synchronization;
   - an atomic variable;
   - a concurrent collection.
10. Handle interruption properly.
11. Close the executor correctly.
12. Add a second implementation using virtual threads for the same independent tasks if the installed JDK supports them.
13. Compare the architecture of the two approaches; do not claim performance improvements without measurement.

Submission requirements:

- all Java source files;
- program output;

- explanation of which data is shared;
- explanation of how race conditions are prevented;
- explanation of how tasks are submitted and results collected;
- explanation of interruption handling;
- comparison of platform-thread executor and virtual-thread executor designs.

44. Key Terms

Process
An executing program with its own operating-system resources.

Thread
A path of execution within a process.

Concurrency
Overlapping progress of multiple tasks.

Parallelism
Simultaneous execution of computations.

Race condition
Incorrect behavior caused by uncontrolled timing of shared operations.

Critical section
Code that accesses shared mutable state and requires coordination.

Mutual exclusion
Ensuring only one thread executes protected code at a time.

`synchronized`
Java mechanism for monitor-based mutual exclusion and visibility.

`volatile`
Field modifier providing visibility guarantees for reads and writes.

Atomic variable
Variable supporting certain indivisible concurrent operations.

Lock
Synchronization object controlling access to protected code.

Deadlock
Permanent waiting cycle among threads.

Executor
Abstraction for executing tasks.

Thread pool
Reusable collection of worker threads.

`Callable`
Task capable of returning a result and throwing checked exceptions.

`Future`
Handle representing a pending or completed asynchronous result.

`CompletableFuture`
Future and completion stage supporting asynchronous composition.

Virtual thread
Lightweight Java thread designed for very high numbers of mostly blocking tasks.

Thread safety
Correct behavior under concurrent access according to a documented contract.