20. Managing concurrent code execution

Exam Objectives

1. Create worker threads using Runnable and Callable, manage the thread lifecycle, including automations provided by different Executor services and concurrent API
2. Develop thread-safe code, using different locking mechanisms and concurrent API
3. Process Java collections concurrently including the use of parallel streams
4. Perform decomposition, concatenation and reduction, and grouping and partitioning on parallel streams

20.1 Concurrency Basics

I am sure you have encountered terms such as multi-processing, multi-threading, hyper-threading, concurrency, asynchronous execution, and parallel processing before. You may also have been confused by them due the hair-splitting that goes on about their meaning on discussion boards and interviews. Let me tell you that although these terms do have different meanings, they all try to achieve the same thing one way or the other - doing multiple things at the same time. Therefore, the broader term that encompasses all of the rest is **multi-tasking**. A person speaking on the phone while driving, a chef preparing different dishes for different customers, a computer playing a movie while downloading the rest of it in the background, are all examples of multi-tasking. Their differences arise due to the difference in perspective and/or their approach towards the solution.

There are only two ways multi-tasking can be achieved - either put multiple resources to work on different tasks or have the same resource work on multiple tasks on a time-sharing basis. For example, a restaurant may have multiple chefs, all working on different dishes. From the perspective of the restaurant, it is **multi-processing** because there are multiple "processors" each working on its own task. It can also be called **parallel processing** because more than one order is actually being prepared in parallel. On the other hand, a food cart operator might be flipping burger one second and chopping salad another. The cart operator is also multi-tasking but since he is just one "processor", he is not multi-processing. He too is preparing multiple orders at the same time but by working in way that creates an illusion of having multiple processors. Since multiple orders are being prepared in both the cases, they are both examples of "concurrent" processing.

Just a decade or so, ago, a CPU chip used to be capable of executing just one instruction at a time. But operating systems such as Windows and Linux were sill able to execute multiple programs at the same time. They were able to do that by time sharing the CPU among all programs. These days even entry level PCs have multi-core processors, in which each core acts as an independent CPU. Furthermore, the chips may also employ techniques such as **hyperthreading** that create an illusion of having multiple logical CPUs within each core. So, from the perspective of an operating system, even a single core chip may come across as a multiprocessing chip. Thus, on such systems, an operating system performs parallel processing when it executes two programs at once on different CPUs.

Many computer programs perform tasks that take a while to complete. Such tasks require a long time either because they involve a lot of computation or because they get stuck on external factors such as user input or network I/O. In both the cases, it is possible to write the program in such a way that independent tasks can be executed concurrently. For example, when you click on a download link in a browser, the mouse still works while the browser downloads the file. Moving the mouse and downloading the file are two independent tasks that can be executed **concurrently**. Whether these two tasks actually execute in parallel depends on the resources available under the hood.

Writing programs that can perform tasks concurrently is not trivial. You need to first identity and separate out the tasks that can be executed concurrently and then use the techniques

provided by the programming language to execute them concurrently. You may have used applications that "hang" for a while when you click on a button. That's because the application is probably busy performing the task linked to the button click and is not able to respond to mouse moves. It is not able to handle two tasks at once because either it is badly written or the language or the platform on which the application is built does not support concurrent programming. Fortunately, most modern programming languages support concurrent programming.

Multi-threading and **Asynchronous functions** are two different techniques provided by programming languages for executing tasks concurrently. **Java** provides multi-threading, while **JavaScript** and **Dart** provide asynchronous functions. **C#** provides both. It is the job of the language and the platform to map the execution of concurrent operations to the resources provided by the underlying subsystem. For example, the JVM maps multiple threads created by a Java program to multiple threads of the underlying OS. The OS, in turn, executes its threads on one or more logical CPUs provided by the underlying hardware on a time sharing basis. The logical CPUs may internally use hyperthreading to execute the instructions on one or more physical CPUs inside the chip. There is no fixed universal rule on how this mapping is done. It really depends on the experience and objectives of the language and platform architects.

20.1.1 Overview of multi-tasking in the Java world

The following diagram provides an overview of the various components involved in achieving multitasking in the Java world.

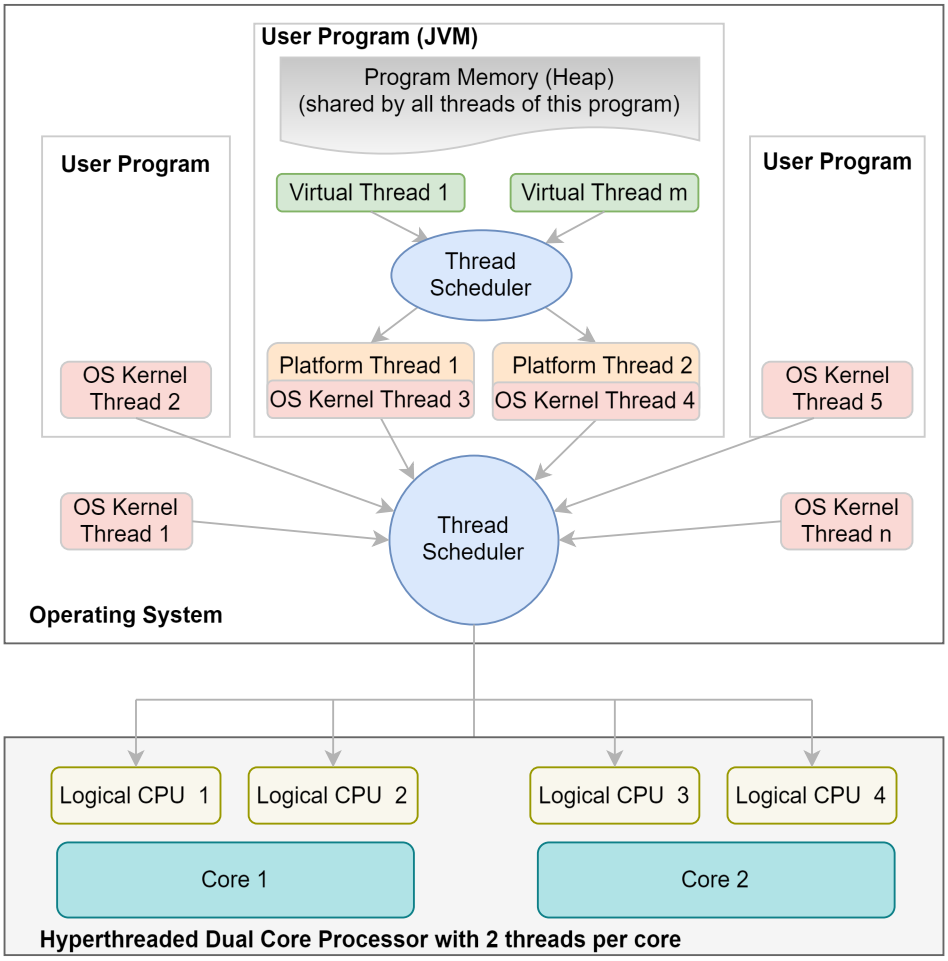

Figure 20.1: Overview of Multi-threading

Here are a few points worth noting about the above diagram:

1. At the top most layer, there are processes. A process is a program, i.e., a list of instructions, in execution. For example, when you click on an executable file, the OS locates the entry point of the program and schedules the execution of instructions from that point. The OS means the world to a process. The OS provides the process with whatever it needs using **system calls**. If you are not from a computer science background, you can imagine a system call as a function provided by the OS that a process can call to utilize the resources of the underlying hardware. For example, if a process wants to read a file, or open a network socket, or wants more memory, it makes a system call. Similarly, the process registers itself with the OS to receive inputs such as mouse moves, key strokes, and network connections, and the OS makes calls to the process to deliver those inputs as and when they occur. It is important to understand that system calls are specific to an OS, which is why a program written for one OS cannot be executed on another OS. Hence the term "platform dependent".

   The OS also provides a process with exclusive access to a certain amount of RAM where the process can keep its execution state. The process is solely responsible for managing

the memory that has been assigned to it. This memory is called program memory and is accessible from all parts of the process. The OS reclaims this memory when the process terminates. A process is not allowed to access memory of another process. If it tries to do that, the OS usually kills the process.

When we launch a Java program, we actually launch the JVM but as far as the OS is concerned, it is just another process.

2. An OS manages the hardware resources available on the physical machine and makes them available to multiple processes on time sharing basis. It achieves this through the use of threads. A thread is like an agent with a list of instructions that it needs to execute on the underlying CPU. The OS creates several threads, some for its own use and some for executing user programs. These threads are called **kernel threads**. When you launch a program, the OS assigns an agent, i.e., a kernel thread that will get the instructions contained in the program executed. Since one CPU can execute only one instruction at a time, all threads cannot get their instructions executed on the same CPU at the same instant. Therefore, the OS uses a **thread scheduler** that decides which thread will have access to the CPU at any given time. The thread scheduler distributes the CPU time among the threads as per the policy implemented by the OS. Once a thread's quota is over, the OS switches that thread out and brings in another thread. This switching between the threads is called **context switching**. It is conceptually similar to what we do while reading multiple books or watching multiple TV shows at different times. When we switch between two shows, we don't go back to Season 1 Episode 1 every time. We remember where we were the last time and we start watching from there. Similarly, a thread knows where it was when it got access to the CPU and it starts executing instructions from that point.
   The thread that the CPU is currently executing is said to be in the **running** state while the thread that is ready to run but is waiting its turn is said to be in the **ready** state. The scheduler is smart enough to know which thread is waiting for some resource to be available and puts that thread in the **waiting** state until that resource becomes available. Of course, if the underlying system has multiple CPUs as shown in the above figure, the scheduler can schedule multiple threads to run at the same time on different CPUs.

3. Zooming into the JVM process, observe the similarity between the components of the whole figure and the components inside the JVM. This is not accidental but by design. A JVM does, in fact, simulate a complete computer including the operating system. So, whatever concepts I explained in the above two points apply equally well to a JVM. Components of a JVM serve the same purpose as the corresponding components in the actual PC conceptually. For example, the program memory of a JVM is like the RAM of system. Virtual threads inside a JVM are like the threads created by the OS. **Platform threads** (aka **native threads**) are like logical CPUs of the JVM. The JVM thread scheduler schedules virtual threads to run on platform threads just like the OS thread scheduler schedules OS threads to run on logical CPUs.
   Thus, when you launch a JVM (using the `java.exe` on Windows or the `java` executable on Linux), you are basically starting up a virtual machine. Just as an OS creates a user thread to launch a process, the JVM too creates a platform thread to execute the instructions present

in your main class. This thread is called the **main thread**.

4. Prior to **Java 21**, whenever the JVM needed to create a thread, it called upon the OS to create it. The OS would then create a thread and assign it to the JVM. Thus, all threads in the JVM were actually just wrappers over OS threads and they would get a chance to execute as and when the OS scheduled the associated thread to execute. Such JVM threads are now officially called **platform threads**. This approach works fine but it has a serious limitation. Although the creation of a new thread by the OS is a lot less taxing than launching a full fledged process (which is why a thread is called a **"lightweight"** process, btw), it still does require a substantial amount of memory (for the thread's stack space) as well as CPU clock ticks. Thus, OS threads are a scarce resource. Furthermore, since the OS is common for all kinds of applications, it has to treat all applications as equal, which essentially means that the OS tries to allocate CPU time to all threads equally. This means, creation of a large number of OS threads causes the OS to spend a lot of time in context switching, which reduces the overall efficiency of the system. Therefore, if an application creates a new thread for every task (aka **thread-per-task** programming model), the application will not work when the number of concurrent tasks goes beyond a few thousand. It will simply run out of system memory after the creation of a few thousand threads. Experience has taught Java language designers that most multithreaded programs spend a lot of time waiting for I/O, whether it be for database queries, files, or network. While the thread-per-task programming model is great for coding such applications, it is impractical when scaled because it causes the machine to run out of system memory long before the CPU reaches full utilization. For this reason, many Java programs, specially server programs such as Tomcat that need to process millions of requests per second, create a pool of threads and reuse them for executing tasks instead of creating a new thread for every request. The thread pooling strategy is so commonly used in Java applications that the JDK too provides ready-made thread pools for executing user defined tasks to manage the cost of creating new threads.

   With **Java 21**, a Java program can now create **virtual threads** . These threads are virtual because the OS has no knowledge of their existence. They are created and managed entirely by the JVM. Unlike platform threads, creation of virtual threads does not require allocation of OS threads and so, they do not require as much memory or CPU cycles to create. A virtual thread can also be garbage collected as soon as it goes out of scope without requiring any cleanup on the part of the OS. Thus, a virtual thread is a much lighter version of a platform thread. Just as a platform thread executes on a logical CPU as and when it is scheduled to execute by the OS's thread scheduler, a virtual thread executes on top of a platform thread as and when it is scheduled to execute by the JVM's thread scheduler. The platform thread on which a virtual thread executes is called the **carrier thread**. The benefit of this approach is that a Java program can create a huge number (as in millions) of virtual threads cheaply. The JVM's thread scheduler schedules a virtual thread to execute on top of an OS thread only when the virtual thread is ready to run. Since a virtual thread is not permanently tied to an OS thread, it does not block an OS thread when it is waiting for I/O. This means, the same OS thread can now be used to execute another virtual thread that is not waiting for anything. Thus, this approach gives the programmers all the benefits of the thread-per-task programming model while achieving optimum CPU utilization and

scalability when deployed.

20.1.2 Executing tasks concurrently

In Java, a task is represented by an instance of the `java.lang.Runnable` interface. This interface has just one method named **run**. It does not take any argument and returns `void`. Once you package your logic into a `Runnable`, it is ready to be executed by a thread, which is represented by the `java.lang.Thread` class. So, the typical way of executing one or more tasks concurrently in Java is to create a `Runnable`, pass this `Runnable` while creating a `Thread` object, and then "start" the `Thread`. Under the hood, the JVM will spawn a new **platform thread** to execute the code present in the `run` method of your `Runnable`. Here is an example:

```
import java.math.BigInteger;
class Factorial implements Runnable{
    int n;
    BigInteger result;
    Factorial(int n){
        this.n = n;
    }
    public void run(){
        BigInteger start = BigInteger.ONE;
        for(int i=1; i<n;i++) start = start.multiply(BigInteger.valueOf(i));
        result = start; //assign start to result once calculation is complete
        System.out.println("Factorial "+n+" computed.");
    }
}
public class TestClass {
    public static void main(String[] args) {
        Factorial f = new Factorial(100000); //create a task
        Thread t = new Thread( f ); //create a new thread with the task
        t.start(); //kick off the execution of the new thread concurrently with the
    current thread
        System.out.println("All Done.");
    }
}
```

The `run` method of the `Factorial` class implements the logic to compute the factorial of a number. Don't worry about the usage of the `BigInteger` class. I am using it only because the value of our factorial is going to be huge but it is not on the exam. Our objective is to execute this logic concurrently with the logic in the main method. If you run the above program, you will **most likely** get the following output:

```
All Done.
Factorial 100000 computed.
```

Until we call `t.start()`, there is only one thread in execution, i.e., the main thread. The call to `t.start()` launches a new thread concurrently with the main thread. From our perspective, both the threads execute independently from each other from this point on. Under the hood,

the OS gets both the threads executed as and when a CPU becomes available to execute their instructions. If the underlying system has multiple processors, the OS may schedule the execution of the two threads on two different CPUs at the same time and they will thus, be executed in parallel. Otherwise, the OS will share the CPU with both the threads on a time sharing basis.

At the time of writing code, we cannot be certain of when and for how long a thread will actually get to run, which is why I said the above output is most likely. Since the factorial of 100000 takes a lot longer (about 5 seconds on my machine) than the print statement in the main method (only a few milliseconds at the most), it is very likely that the main thread will finish its execution long before the thread that computes the factorial. But the important thing is that we cannot assume this sequence of execution while writing the code. It is entirely possible that, at run time, the OS allocates 10 seconds of CPU time to the factorial thread first and 10 seconds to the main thread next. In that case, the output of the two print statements will be reversed. You need to understand and remember this point very clearly because this is **one of the two** main causes of bugs in multithreaded programs. You just can't base the logic of your program on the timing of the threads. The OS can schedule any thread to run at any time.

Finally, when you run the above program, you will notice that the program doesn't terminate even when the end of the main method is reached. You will see the cursor blinking for a while after the print statement in the main method executes. It terminates only after the print statement from the run method is executed. That is because a Java program is not composed of just the main thread. A Java program comprises all the threads that we have started including the main thread as well as the threads that the JVM starts for its housekeeping purpose (such as for garbage collection) called **daemon threads**. The program does not terminate until all non-daemon threads finish their execution. The distinction between daemon and non-daemon threads is important because daemon threads exist only to support the non-daemon threads. If there are no non-daemon threads running in the JVM, there is no need to keep the daemon threads running. For example, if there is no business logic to execute, why would a program keep performing garbage collection?

The above example isn't too exciting because the time taken by the program to finish is the same as the time taken by a program that computes the factorial without using a thread. So, let me modify the above program to show you the advantage of concurrent execution of threads.

```
public static void main(String[] args) throws InterruptedException{
    long start = System.currentTimeMillis();
    Factorial f1 = new Factorial(150000);
    Thread t1 = new Thread( f1 );
    t1.start();

    Factorial f2 = new Factorial(150000);
    Thread t2 = new Thread( f2 );
    t2.start();

    t1.join();
    t2.join();
    long end = System.currentTimeMillis();
```

```
    System.out.println("All Done. Time taken (secs):"+ (end-start)/1000);
}
```

The output of the above program on my machine is:

```
Factorial 150000 computed.
Factorial 150000 computed.
All Done. Total time taken (secs):7
```

The above program computes two factorials concurrently. After starting the two threads, I am calling the join method on `t1` and `t2`. The `join` method makes the calling thread (i.e., the main thread in this example) to pause until the thread on which `join()` is called ends. This ensures that the print statement in main executes only after the two threads finish. The `join` method can potentially throw an `InterruptionException`, which is a checked exception. So, I have declared it in the `throws` clause.

Since I am running it on a multi-core system, the two threads get to run on two cores in parallel. That is why the program finishes the two computations in less time than the time it would have taken to finish them sequentially. It does take a little more time than it takes for one computation but that is because some time is spent by the OS in creating and in context-switching between the threads.

You can modify the above program to compute more factorials concurrently and observe that the time taken by the program doesn't increase much until you create more threads than the number of cores on your system because the OS is able to execute each thread in parallel on different cores.

Using lambdas to create tasks

Since `Runnable` has exactly one abstract method, it is a functional interface and so, it is common to use lambda expressions to create `Runnable`s for simple tasks that do not require to hold any state. For example:

```
public static void main(String[] args){
    new Thread( ()-> System.out.println(Thread.currentThread().getName())
    ).start();
    System.out.println(Thread.currentThread().getName());
}
```

As I mentioned before, which thread gets to execute its instructions first cannot be determined in advance, so, the following is a likely output:

```
main
Thread-0
```

Extending `Thread` to create a task

The `Thread` class implements `Runnable` and so, it is possible to extend the `Thread` class and override its `run` method to implement your logic in it. For example:

```
public static void main(String[] args) {
    Thread t = new Thread(){ //creating an anonymous class that extends Thread
        public void run(){
            System.out.println(Thread.currentThread().getName());
        }
    };
    t.start();
}
```

Extending `Thread` eliminates the need to create a separate `Runnable` object and may seem convenient. However, from a design perspective, this is a bad approach because a `Thread` does not represent a task. A `Thread` is only a means to execute a task. For this reason, this approach is not recommended.

Exam Tip

Thread's start() vs run()

An unfortunate consequence of the `Thread` class implementing `Runnable` is that it is possible to call `run` on a `Thread` object. However, calling `run` on a `Thread` object acts just like any other normal method call, which means, the code in the `run` method executes on the same thread from which it is called (instead of starting a new thread).

Remember that to kick off the execution of a new thread **concurrently**, you need to call the `start` method on the `Thread` object.

So, for example: the following code prints main because the `run` method gets executed on the main thread and not on a separate thread:

```
public static void main(String[] args) {
    Thread t = new Thread(){
        public void run(){
            System.out.println(Thread.currentThread().getName());
        }
    };
    t.run(); //should call start() here
}
```

Using `Thread.Builder` to create threads

The way of creating threads discussed above is the pre-Java 21 way of creating threads. It remains valid even now in Java 21 and works the same as before. But it is meant to create platform threads only. Now that Java 21 has introduced virtual threads, a new `Thread.Builder` interface has been defined in the Java standard library for creating platform as well as virtual threads. The `Thread` class too has been enhanced to include two new static methods named `ofPlatform` and `ofVirtual`

that return `Thread.Builder` objects to create platform and virtual threads respectively. Here is how the new API is used:

```
public static void main(String[] args) {
  Factorial f = new Factorial(100000); //create a task
  Thread.Builder tb = Thread.ofPlatform(); //create a builder for creating platform
    threads
  Thread t = tb.unstarted(f); //create a new platform thread with the task
  t.start(); //start the thread
  System.out.println("All Done. Result: "+f.result);
}
```

I have used three lines of code above to start a platform thread only to show the return types of these methods but typically, this is done using just one line of code:
`Thread.ofPlatform().start(f); //the start method creates as well as starts the thread`

You should go through the JavaDoc description of `Thread.Builder` but its following four method are used more often than others:

1. `Thread.Builder name(String name)`: Sets the thread name.
2. `Thread.Builder name(String prefix, long start)`: Sets the thread name to be the concatenation of a string prefix and the string representation of a counter value. This method is useful when the `Thread.Builder` object is used to create multiple threads each having a different name.
3. `Thread start(Runnable task)`: Creates a new thread from the current state of the builder and schedules it to execute.
4. `Thread unstarted(Runnable task)`: Creates a new thread from the current state of the builder to run the given task.

Daemon Threads

When you run the above program, you will notice that the program doesn't terminate even when the end of the main method is reached. You will see the cursor blinking for a while after the print statement in the `main` method executes. The program terminates only after the print statement in the `run` method of the `Factorial` class is executed. That is because a Java program is not composed of just the main thread. A Java program is composed of all the threads that are started explicitly by our code as well as the threads started by the JVM on its own such as the main thread. Thus, a program cannot terminate until all the running threads terminate. However, there is a exception.

Some of the threads that the JVM starts are merely meant for "housekeeping" purpose and they do not execute any logic that is relevant to the user (called "business logic"). Such threads that exist only to play a supporting role in the program are called **daemon threads**. For example, when you launch a Java program, the JVM automatically starts a thread for garbage collection as

well. Now, garbage collection is needed only if the program is executing some business logic. It doesn't make sense to keep this thread running if there is no business logic left to execute. For this reason, the JVM terminates all the daemon threads automatically after all non-daemon threads terminate. It follows then that **a Java program terminates when all non-daemon threads terminate**.

By default, all **platform threads** are non-daemon threads but any thread can be marked as a daemon thread **before that thread is started**, by calling `setDaemon(true)` on that thread. Which means, you too can use daemon threads for executing business logic that you want to execute only if other parts of your program are running. For example, you might want to create a thread that loops every 30 minutes and removes all those entries of a cache that have not been used in the past 30 minutes. You can mark this thread as a daemon thread because you wouldn't want this thread to prevent the program from terminating if nothing else is running.

The `Thread` class also has an `isDaemon()` method that lets you check whether a thread is a daemon thread or not. It returns a `boolean`.

Creating Virtual Threads

Virtual threads are represented by the `java.lang.VirtualThread` class, which is a subclass of `java.lang.Thread`. However, unlike `Thread`, `VirtualThread` is not public. Therefore, let alone instantiating a `VirtualThread` object, you cannot even refer to this class in your code. This is done purposefully because usage of virtual threads is supposed to be transparent once they are created. Virtual threads are threads and they support the same API as platform threads.

So, you must use either `Thread.Builder`'s `start`/`unstarted` methods or `Thread`'s `startVirtualThread(Runnable r)` method to create a virtual thread:

```
public static void main(String[] args) {
  Factorial f = new Factorial(100000); //create a task
  Thread t = Thread.ofVirtual().start(f);
  //Thread t = Thread.startVirtualThread(f); //this is equivalent to the above line
  t.join(); //call to join is important
  System.out.println("All Done. Result: "+f.result);
}
```

An important fact about virtual threads is that as of Java 21 a virtual thread is always a daemon thread. Calling `setDaemon(false)` on a virtual thread throws a `java.lang.IllegalArgumentException`. Why? Good question. While I couldn't find an officially documented reason for it, I did read comments from the Java language designers that since virtual threads are meant to execute short-lived business tasks that can be GCed like any other object, they should be deamon threads. Furthermore, that if a good reason to make them non-daemon is found, this restriction might be lifted in the future. Personally, I find this restriction unreasonable. If a virtual thread is executing a business task, no matter how short-lived, it must be given a chance to finish that task instead of abruptly terminating it as that could leave the business state inconsistent.

Okay, so, given that a virtual thread is always a daemon thread, it is easy to see why the call to `t.join()` is required in the above code. Without this call, the main thread will terminate after the print statement is executed. Once the main thread is terminated, there is no non-daemon thread left in the given program. Since daemon threads cannot keep a program from terminating, the above program will terminate while the factorial thread is still running. Thus, the print statement from the factorial thread will most likely not be executed. The call to join causes the main thread to wait for the factorial thread to end.

20.1.3 Thread API

Below, I am listing a few methods of the `Thread` class that are used often in real life or are important for the exam. You have already seen some of these in the above examples. You will see the usage of the others in the rest of the chapter.

Static methods of `java.lang.Thread`

1. `Thread currentThread()` - Returns the Thread object for the current thread. This method is very useful when you want to print the id or the name of the thread that is currently executing that piece of code.

2. `boolean interrupted()` - Tests whether the current thread has been interrupted. This method **clears** the interrupted status of the current thread.

3. `void sleep(long milliseconds)` - Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds. Throws `InterruptedException`. There are two more overloaded sleep methods that do the same thing: `sleep(long millis, int nanos)` and `sleep(Duration duration)`. This method **does not** release the locks that have been acquired by the current thread.

4. `Thread.Builder.OfPlatform ofPlatform()` - Returns a builder for creating a platform Thread or ThreadFactory that creates platform threads.

5. `Thread.Builder.OfVirtual ofVirtual()` - Returns a builder for creating a virtual Thread or ThreadFactory that creates virtual threads.

6. `Thread startVirtualThread(Runnable task)` - Creates a virtual thread to execute a task and schedules it to execute.

Instance methods of `java.lang.Thread`

1. `long threadId()` - Returns the identifier of this Thread. The thread ID is a positive long number generated when this thread was created. The thread ID is unique and **remains unchanged** during its lifetime.

2. `String getName()/void setName(String name)` - Returns/sets this thread's name.

3. `boolean isDaemon()/void setDaemon(boolean on)` - Tests if this thread is a daemon thread. / Marks this thread as either a daemon or non-daemon thread. Remember that virtual threads are always daemon threads as of Java 21.
4. `boolean isVirtual()` - Returns `true` if this thread is a virtual thread.
5. `void start()` - Schedules this thread to begin execution.
6. `void run()` - This method is run by the thread when it executes.
7. `void join()` - Waits for this thread to terminate. Throws `InterruptedException`.
8. `void join(long millis)` - Waits at most millis milliseconds for this thread to terminate. Throws `InterruptedException`. There are two more overloaded join methods that work similarly: `join(long millis, int nanos)` and `join(Duration duration)`.
9. `void interrupt()` - Interrupts this thread. If this thread is blocked in an invocation of one of the `wait`, `join`, or `sleep` methods, then its interrupt status will be cleared and it will receive an `InterruptedException`. Otherwise, thread's interrupt status will be set. Meaning, calling `isInterrupted()` on this thread will return `true`. This method is used to coordinate the execution of two threads.
10. `Thread.State getState()` - Returns the state of this thread.
11. `boolean isAlive()` - Tests if this thread is alive.
12. `boolean isInterrupted()` - Tests whether this thread has been interrupted. It **does not clear** the interrupted status of this thread.

Observe that `sleep` and `join` methods may throw a `java.lang.InterruptedException`, which is a **checked exception**.

Thread states

A thread's state tells us what is currently going on with the thread and what can we expect from it next. These states are represented by the six constants defined in an enum named `java.lang.Thread.State` and are applicable to platform as well as virtual threads. These are as follows:

1. `NEW` - A thread that has not yet started is in this state. It will not do anything until its start method is invoked.
2. `RUNNABLE` - A thread becomes runnable once its start method is invoked. This means, nothing is now holding back the thread from executing the run method of the task except the availability of CPU time.
3. `TERMINATED` - A thread that has exited is in this state. This happens when a thread has finished executing the run method of its task. Once a thread is terminated, it cannot be restarted.

4. `BLOCKED` - A thread that is blocked waiting for a monitor lock is in this state. It becomes `RUNNABLE` again when it gets the lock it was trying to get.

5. `WAITING` - A thread that is waiting indefinitely for another thread to perform a particular action is in this state. Typically, it goes into this state when it calls `wait()` on an object or `join()` on another runnable thread. It becomes `RUNNABLE` again when the other thread performs that particular action. That action is typically a call to `notify()` or `notifyAll()` on the same object or when the thread terminates.

6. `TIMED_WAITING` - A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state. Typically, it goes into this state when it calls wait with a timeout argument on an object or join with a timeout argument on another runnable thread. It becomes `RUNNABLE` again when the other thread performs that particular action or when the waiting time is over. A thread could also put itself in this state by calling one of the overloaded `sleep` methods.

The following diagram illustrates how a thread transitions between these states.

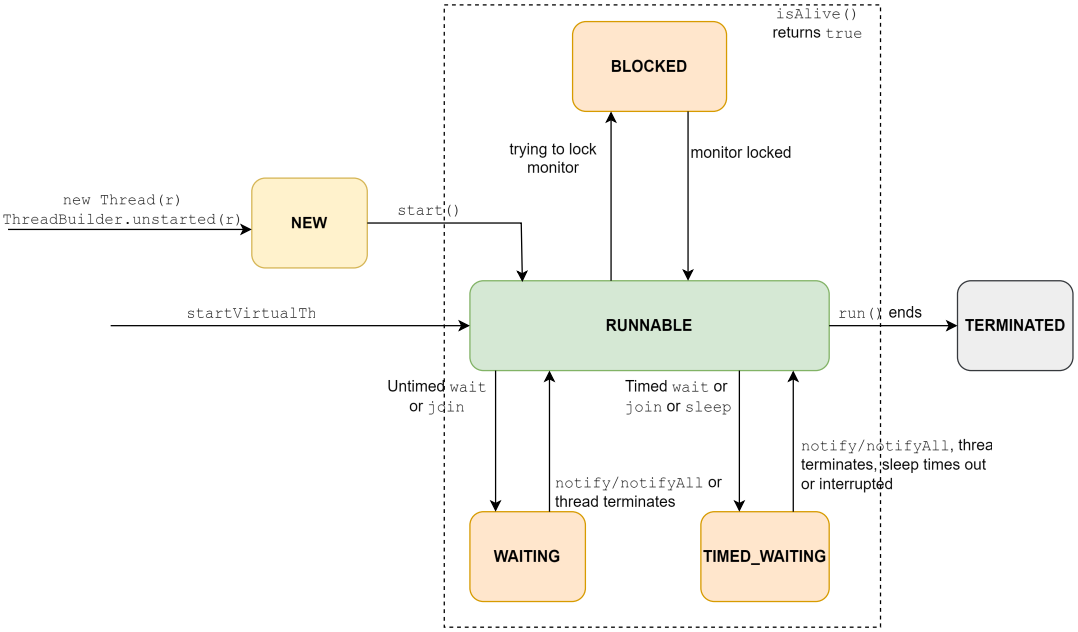

Figure 20.2: Thread States and thread transition

When a thread is `BLOCKED`, `WAITING`, and `TIME_WAITING` state, the scheduler will not schedule it to run, which implies that a thread does not consume CPU cycles when it is in one of these states. As you will soon learn, a thread transitions to these states while trying to gain exclusive access to shared variables or while coordinating its execution with one or more threads but the important point to note is that a thread can move back and forth between `RUNNABLE` and `BLOCKED`, `RUNNABLE` and `WAITING`, or `RUNNABLE` and `TIMED_WAITING` states multiple times during its execution as determined by the business logic of that task. Thread's `isAlive` method returns `true` only when it is in one of these four states.

Note

Difference between BLOCKED and WAITING/TIMED_WAITING states

A thread cannot be scheduled for execution when it is in any of the BLOCKED, WAITING, and TIMED_WAITING states. In that sense, these states are similar and are called "blocking states". But there is a fundamental difference between BLOCKED and WAITING/TIMED_WAITING states. A BLOCKED thread becomes RUNNABLE and starts contending for the lock again automatically when the lock that it was trying to acquire becomes available. While a WAITING/TIMED_WAITING thread is simply waiting on either a notification from another thread or on timeout. It becomes RUNNABLE when these events occur. This difference plays an important role during thread coordination.

In general though, program logic should not rely on knowing exactly which one of the blocking states a thread is in because these state transitions are highly implementation dependent. Knowing the exact state of a thread is useful for debugging purpose only.

Thread Priority

A **platform thread** has a priority, which can be anywhere between `Thread.MIN_PRIORITY` and `Thread.MAX_PRIORITY` and can managed using the `int getPriority()` and `void setPriority(int p)` methods. Thread priority is just an extra bit of information that the OS task scheduler may use to optimize the scheduling of threads but there is no guarantee that the OS will use it to any affect and so, programs that rely upon it for thread coordination may not always perform as expected.
Virtual threads always have the same priority and it is set to `Thread.NORM_PRIORITY`. A call to change its priority is ignored.

20.1.4 Quiz

Q. What output can be expected from the following code?

```
public class ThreadStates {
  public static void main(String[] args) throws InterruptedException{
    Runnable r = ()->{
      Thread t = Thread.currentThread();
      try{
        Thread.sleep(5000);
      }catch(InterruptedException, i.e., ){
        System.out.println("1: "+t.getState());
      }
    };

    Thread t  = Thread.ofPlatform().start(r);
    t.interrupt();
    System.out.println("2: "+t.getState());
```

```
  }
}
```

Select **2** correct options.

A.

```
2: RUNNABLE
1: RUNNABLE
```

B.

```
2: TIMED_WAITING
1: RUNNABLE
```

C.

```
2: RUNNABLE
1: INTERRUPTED
```

D.

```
2: BLOCKED
1: RUNNABLE
```

E.

```
2: RUNNABLE
```

Correct answer is: **A**, **B**, and **E**.

Consider the following two possibilities:
Possibility 1: The main thread launches the new thread `t` and executes `t.interrupt()` **after** the new thread executes `Thread.sleep(5000)`. Since, in this case, thread `t` is sleeping when it is interrupted, it will transition to the `RUNNABLE` state and receive an `InterruptedException`, which will be caught by the catch clause. Thus, the print statement in the catch clause will print `1: RUNNABLE`. Since the print statement in the main thread may either execute before or after the thread `t` is made `RUNNABLE`, it could print `2: TIMED_WAITING` or `2: RUNNABLE`. Hence, **options A** and **B** are correct. Not that the print statement in the catch clause and the print statement in the main method could be executed in any order depending on which thread gets to run run first. In fact, if thread `t` gets to execute the catch clause before the main thread gets to execute the print statement, the main thread could even find that thread `t` has finished execution and so, it could also print `2: TERMINATED`.
Possibility 2: The main thread launches the new thread `t` and executes `t.interrupt()` **before** the new thread gets a chance to execute `Thread.sleep(5000)`. Since, in this case, thread `t` is not sleeping when it is interrupted, only its interrupted status will be set to true and it will not receive any exception. Therefore, the `catch` clause will not be executed. The print statement in the main thread may get to execute either before or after thread `t` executes `t.sleep(5000)`. It may, therefore, print `2: RUNNABLE` or `2: TIMED_WAITING`. Hence, **option E** is correct.
Note that there is no state called `INTERRUPTED`. So, option C is incorrect. A sleeping thread is in the `TIMED_WAITING` state and not in `BLOCKED` state. So, option D is also incorrect.

20.2 Sharing memory with multiple threads

The example programs that I showed in the previous section create threads that execute independently of each other. Meaning, they go about doing their jobs without caring what the other threads are doing. They don't depend on another thread for any data and they don't produce any data that is required by another thread. Those programs are fine for introducing multi-threading, but most non-trivial multi-threaded programs require some sort of data sharing as well as coordination among themselves. For example, in a simple producer-consumer program where one thread consumes the data that is produced by another thread, has to have some coordination so that the consumer thread can know when the producer thread has produced the data and can then use the data. Although "sharing" and "coordination" are not exactly the same thing, there is lot of overlap between the two. Let us look at sharing first.

Since all threads in a Java program belong to the same OS process, they share the memory allocated to the process and since all objects exist on the heap space, which is common to all threads, sharing an object between multiple threads is as easy as making its reference accessible from different classes. Even a public static field of a class can be used for this purpose.

However, in this simplicity lies the **second of the two** main causes of bugs in multithreaded programs. Let's see how by modifying our factorial program a little bit:

```
public static void main(String[] args) throws Exception{
  Factorial f = new Factorial(100000); //create a task
  Thread t = Thread.ofPlatform().start(f);
  while(f.result == null){ } //this line of code is new
  System.out.println("Factorial is : "+f.result); //this line of code now prints
    f.result
}
```

The `Factorial` class is the same as before but I have modified the `main` method to print the factorial. Observe that now the `result` field of the `Factorial` object is set by one thread (I am calling it the "factorial thread") but it is read by another (the main thread) and so, this has now become a classic producer-consumer program. The factorial thread is the producer of the `result` field and the main thread is the consumer of the same.

Since the computation takes a while, I don't want the print statement in the main method to execute until the result is computed. To ensure this, I have put a loop that keeps checking whether the result has been computed or not. Once the result is computed, the while condition becomes false and the loop ends. The next statement after the while loop prints `result`. Looks good. Right? Wrong. There are two issues with this solution.

The first problem is kind of obvious. Checking the value of a variable in a loop until its value changes will cause the CPU usage to spike. It is called a "busy-waiting" (or busy-looping or spinning) and is an highly inefficient coordination technique. But at least it doesn't cause the program to produce incorrect output.

The second problem is a lot more serious and actually makes the program entirely unusable. In fact, the program doesn't even terminate on my machine and there is a good chance that it will not terminate on yours either. The problem is that the main thread keeps seeing the original value (`null`) of the `result` field even after the factorial thread assigns a new value to it. It is not a bug. It is how the Java language is designed to work. To fix this problem, you need know about the Java "Memory Model". Although the OCP exam objectives do not include this topic explicitly, you can write bug free multi-threaded Java programs without understanding it only accidentally. So, I am going to the go through the important points about this topic briefly.

20.2.1 Memory Model

As I mentioned before, all threads share the heap space of the program. They access objects within this heap space through object references, which are nothing but addresses of **virtual memory locations**. When a Java program reads or writes an object field, this call is translated by the JVM and the OS to a CPU instruction to read from or write to an **actual memory location** as per the CPU's addressing scheme. Now, recall that a system may have multiple logical CPUs and multiple OS threads may run in parallel on those CPUs. To ensure that multiple CPUs can execute instructions simultaneously with high efficiency, CPUs tend to store the contents of memory locations in multilevel caches as well as in CPU registers. So, it is possible that while one CPU has cached the value of a memory location into its register, another CPU changes the value at the actual memory location. Thus, a value cached by one CPU can become "out of sync" with the value at the memory location or with the value cached by another CPU. At the top most layer, i.e., in the Java program, an out of sync cached value results in one or more threads seeing a stale value of the shared variable. This is exactly what happens in the above program. The factorial thread updates the value of the `result` variable but the main thread never sees the updated value because both the threads are most likely being run on different CPUs and their caches are out of sync. This situation is very common on multi-core CPUs.

Another reason why a thread may not observe the updated value of a variable immediately after being updated by another thread is "instruction reordering". Ideally, you would expect that a line of code that appears earlier in a method will execute before another line of code that appear later in the same method. However, to optimize the performance, a compiler may chose to reorder some lines of code, or even omit some lines from the compiled code, if it is able to determine that such restructuring will not affect the overall semantics of the given method. But such restructuring may produce unwanted side effects in a multi-threaded environment.

Going into the details of such optimizations is beyond the scope of this book but the point is that there are multiple reasons for observing the stale and thus, incorrect, value of a shared variable.

How can this be prevented? Well, the OS and the CPU do provide techniques to prevent caches from going out of sync, but preventing this from happening at all times and for all memory locations is very expensive and a blanket use of those techniques degrades the performance of the system substantially. Therefore, many programming languages such as Java do not apply those techniques unless a programmer explicitly declares their intent that syncing a particular set of

values is indeed required by the program.

The **Memory Model** of a programming language describes when the language runtime keeps the caches in sync and how a program written in that language can control this behavior. It also limits the extent to which a compiler can reorder instructions. A memory model allows a high level programming language such as Java to provide a uniform set of platform independent techniques by wrapping the use of the underlying OS and CPU specific techniques. A memory model thus, provides programmers with options that can help them control the cost of cache synchronization.

20.2.2 The Java Memory Model

Java does not expose the OS level cache synchronization techniques to the Java programmer directly. In other words, there is no API that you can use to manage or even observe the caches used by the platform that exists underneath the JVM. There is no API to prevent reordering of instructions either. Instead, Java defines a "memory model", known as the **Java Memory Model**, that abstracts this problem away into a set of rules that, if followed, establish a predictable, consistent, and dependable, order of execution of the statements irrespective of the thread from which they are executed. Effectively, the Java memory model defines how the shared memory behaves when accessed by multiple threads. More specifically, it defines the conditions in which updates to a shared variable by one thread can be seen by another thread. Here, shared memory essentially means static and instance fields only because method local variables are always local to one thread and they are never shared with any other thread.

Looking back at the factorial example above from this perspective, although the factorial thread may have executed the assignment statement, the main thread failed to see the update to the `result` field because the Java memory model does not promise that the assignment operation performed by one thread, i.e., the factorial thread, on the `result` variable will actually be seen as executed from another thread, i.e., the main thread. In other words, since the Java memory model does not promise that the assignment done by the factorial thread will happen **before** the main thread tries to read the same field, the main thread may not observe that the update is completed by the factorial thread even if the main thread keeps checking the `result` variable forever. Since the assignment may, theoretically, appear to happen **after** one, two, or any number of checks performed by the main thread, the main thread cannot depend on the fact that the assignment will definitely be executed before any number of checks. Thus, the main thread may never come out of the while loop.

To ensure that an action is seen as happened before another action, there needs to exist a "happens before" relationship between the two actions. The Java memory model defines the conditions in which such a relationship exits and it is up to the JVM implementers how they implement these relationships internally. As Java programmers, we just need to create such conditions in our code so that we can rely on the visibility guarantees provided by the Java memory model.

Although there are numerous ways in which a happens-before is established but don't worry, there are only six of them that we care about the most at the moment.

Establishing a happens-before relationship

1. **Program order** - Each action in a thread happens before every action in that thread that comes later in the program. This is the simplest of the guarantees and it means that if a line of code appears before another line of code in a method then a thread executing that code will observe that the first line of code has indeed been executed before the second. In other words, a thread always observes the execution order to be the same as the program order. For example, consider the following code:

```
//assume that the variable x is accessible in this context
  //x could be a static field or an instance field
  x = 0;
  x = 10;
  System.out.println(x);
```

   In the above code, a value of `10` is assigned to `x` before a call to the `println` method is made. Thus, the `println` method will observe the value of `x` to be `10`. This is kind of obvious because the sequential execution order of instructions by a thread forms the bedrock of programming but this guarantee restricts the compiler and the JVM from changing the apparent order of the three statements.

2. **Starting a thread** - A call to `start()` on a thread happens before any actions in the started thread. This rule implies that updates made by a thread cannot be seen by other threads unless that thread has already been started. This rules also kind of states the obvious but needs to be stated to avoid overzealous optimizations and reordering that might make the updates visible to other threads even before the thread is started.

3. **Observing the termination of a thread** - All actions in a thread happen before any other thread observes that this thread has terminated. A thread can observe the termination of another thread by calling the `join` or the `isAlive` methods on that thread.

   Calling the `join()` method makes the calling thread wait until the other thread terminates. Thus, this rule ensures that if a thread joins on another thread then the calling thread will see all the updates made to the shared variables by the other thread when the call to `join()` returns. For example, this rule can be used to fix our factorial program:

```
//while(f.result == null){ } //no need to busy wait
t.join(); //can throw an InterruptedException, which is a checked exception, so
    declare it in the throws clause
System.out.println("Factorial is : "+f.result); //will print the result correctly
```

   Observe that calling `t.join()` allows us to get rid of the busy-waiting while loop.

   Calling `isAlive` provides the same guarantee as `join`, but only if it returns `true`. Since it returns immediately with the status of the other thread, you need to use this method in a busy-waiting loop:

```
while(f.isAlive()){ } //not a great option but better than checking for result ==
    null
```

```
System.out.println("Factorial is : "+f.result); //will print the result correctly
```

Unlike `join`, `isAlive` doesn't throw any exception.

4. **Observing the interrupted status of a thread** - If thread T1 interrupts thread T2, all actions performed by T1 just before interrupting T2 happen before any other thread (including T2) determines that T2 has been interrupted (by having an `InterruptedException` thrown or by invoking the `interrupted()`/`isInterrupted()` methods). I am not giving an example here because I haven't discussed thread coordination yet and it is not too important for the exam either but I encourage you to write one after finishing the next section on thread coordination similar to the one I have given above for `join()`.

5. **Using a volatile field** - Everything that happens in a thread before it writes to a **volatile field** (remember, only static and instance fields can be marked `volatile`), appears to have happened before for another thread that reads the same volatile field. The following example will make it clear:

```
class Student{
    int id;
    volatile String name;
}
public class TestClass {
  public static void main(String[] args) {
    Student s = new Student();
    //writer thread
    Thread.ofPlatform().start(()-> {
      s.id = 10;
      s.name = "bob";
      s.id = 20;});

    //reader thread
    Thread.ofPlatform().start(()-> {
      while(s.name == null){ }
      System.out.println(s.id);
      });
    }
}
```

In the above code, one thread writes to the instance fields of a `Student` object and another thread reads them. Since any thread could get a chance to run at any time, we can't assume that the writer thread will always be able to set the instance fields before the reader thread gets to read them. So, the reader thread uses a while loop to check whether the writer thread has gotten a chance to update the `name` field. While the reader thread is busy testing the `name` field to be not `null`, the OS will ultimately schedule the writer thread to run and the writer thread will set the `id` field to `10`, the `name` field to `"bob"`, and then the `id` field again to `20`. Observe that at the time of setting the `name` field to `"bob"`, the value of the `id` field, as seen by the writer thread, is `10`.

Now, since the `name` field is marked `volatile`, the JVM guarantees that any thread that tries to read this field after it has been written to, will observe that the write has really been executed, i.e., it will see the updated value. Furthermore, that the reader thread will see the same or more recent values of other fields also (even if they are not marked `volatile`) that the writer thread has seen at the time of writing to the volatile field. Thus, the reader thread is guaranteed to see the `name` field change from `null` to not `null` at some point. It is also guaranteed to NOT see the value of the `id` field as `0` because at the time of writing to the `name` field, the value of the `id` field is `10`. Note that the reader thread is not guaranteed to see the update made to the `id` field **after** the writer thread has updated the `name` field because the `id` field is not marked `volatile`.

From the above discussion, you can see that the factorial program can be fixed by marking the `result` field `volatile`.

6. **Locking the same monitor as the one unlocked by another thread** - In Java, every object is associated with a monitor (also known as an intrinsic lock). This monitor can be locked and unlocked by a thread by using a "synchronized" block. At most one thread can lock the monitor at a time. If a monitor is already locked by a thread, other threads have to wait until the thread that has already locked the monitor unlocks it. Although monitors are designed to solve a bigger problem than just making updates visible to other threads, they can, technically, be used to establish a happens-before relationship as well.
The following example shows how but remember that I am showing this example only for the sake of completeness. I do not recommend this approach at all because besides being a lot more expensive than marking a variable volatile, it is too convoluted:

```
class Student{  int id;  String name; //not using volatile here
}
public class WorksButNotRecommened{
  public static void main(String[] args) {
    Student s = new Student();
    Thread.ofPlatform().start(()-> {
      s.id = 10; s.name = "bob";
      synchronized(s){} //locking and unlocking
      });

    Thread.ofPlatform().start(()-> {
      while(s.name == null){ synchronized(s){} }
      System.out.println(s.id);
      });
  }
}
```

When the writer thread encounters `synchronized(s)`, it locks the monitor associated with the object referred to by the variable `s`. It then enters the code block (which is empty, in this case), executes the code, and unlocks the monitor while leaving the code block.
On the other hand, the busy-waiting while loop in the reader thread ensures that the reader

thread will ultimately get to execute the `synchronized(s){ }` statement after the writer thread has updated the fields and has unlocked the same monitor. This establishes a happens-before relationship between the two threads at that point and so the values of the `id` and `name` fields that the writer thread has seen just before unlocking the monitor will be seen by the reader thread after it locks the same monitor.

The Java Memory Model is applicable to all threads and so, the above mentioned guarantees apply to both **platform threads** as well as **virtual threads**.

You will see later that invoking the methods of some of the common multi-threading related classes such as `Future` and `ExecutorService` also establish a happens-before relationship. The OCP exam does not expect you to remember all of them by heart. But while programming professionally, whenever your code depends on such a relationship to exist after calling a method, you should verify it from the JavaDoc of that class and method. If the description says something like, "Memory consistency effects: Actions taken by the asynchronous computation happen-before actions following the corresponding Future.get() in another thread.", then that means you can depend on the guarantees provided by such a relationship after the method returns.

20.2.3 Sharing data using monitors

Making shared variables `volatile` or observing the termination of threads using `join`/`isAlive` work well when you want to make sure that a thread is able to see the most recent values of one or more shared variables. However, these mechanisms fall short when you want to update multiple shared variables "atomically". The following example illustrates what I mean:

```
class Student{
  volatile int id;  volatile String name;
}
public class TestClass {
  public static void main(String[] args) throws InterruptedException{
    Student s = new Student();
    Thread t1 = Thread.ofPlatform().start(()->{ s.id = 1; s.name = "Amy"; });
    Thread t2 = Thread.ofPlatform().start(()->{ s.id = 2; s.name = "Bob"; });
    Thread.sleep(5000); //causes the current thread to sleep for 5 seconds
    //t1.join(); t2.join(); //this will not solve the problem either
    System.out.println("Id = "+s.id+" Name = "+s.name);
  }
}
```

Ideally, I would like the above program to print either `Id = 1 Name = Amy` or `Id = 2 Name = Bob`. But imagine the following scenario:

The program starts the main thread. The main thread creates a new `Student` object and starts two new threads **T1** and **T2.** The main thread then sleeps for 5 seconds (or joins on the two threads). Now, the OS schedules thread T1 to run. T1 executes the statement `s.id = 1;` but before T1 could execute the second statement, i.e., `s.name = "Amy";`, the OS preempts T1 and schedules the thread T2 to run. The second thread executes two statements, i.e., `s.id = 2;` and

`s.name = "Bob"` and terminates. The OS schedules T1 to run again. T1 executes the remaining statement, i.e., `s.name = "Amy";` and terminates. So, now `id` is `2` and `name` is `"Amy"`. Finally, the main thread wakes up and since both `id` and `name` fields are volatile, the main thread is able to see the updated values of both the fields. Therefore, it prints `Id = 2 Name = Amy`. The following diagram shows how the three threads (main, T1, and T2) could potentially get CPU time to execute.

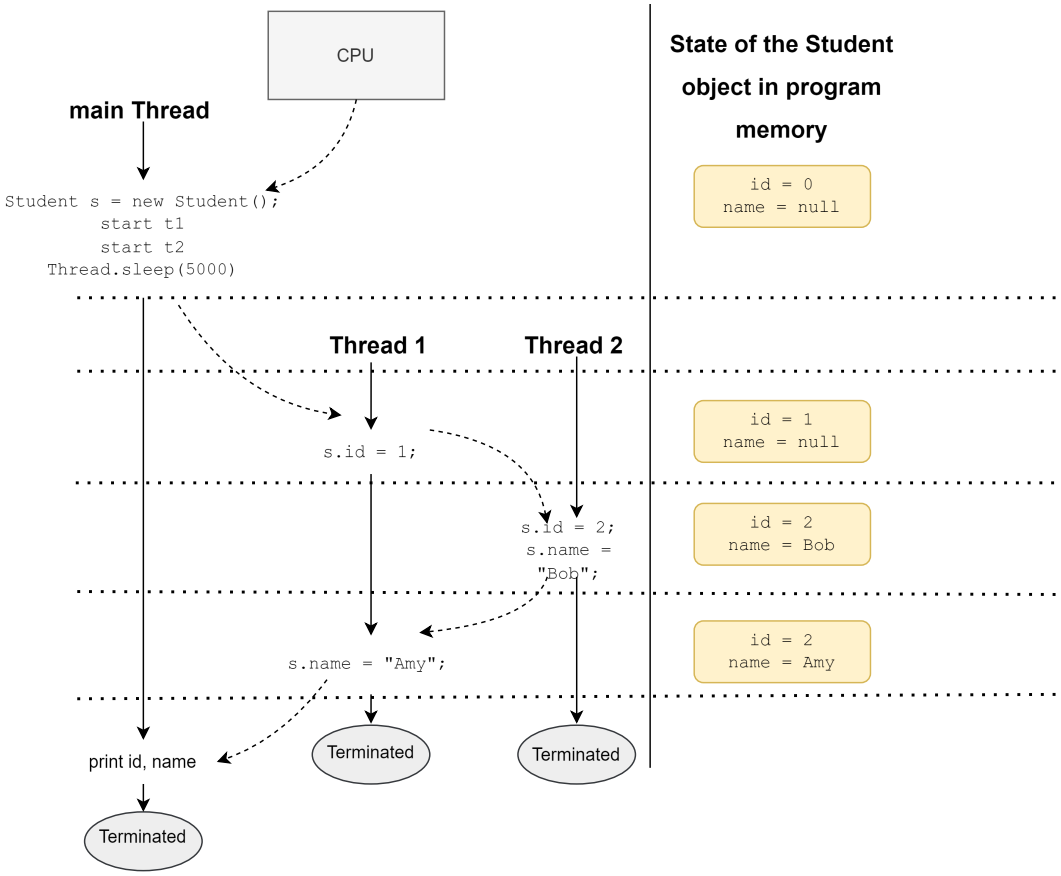

Figure 20.3: Thread scheduling on a single CPU scenario

The output in this scenario highlights the problem with the program. The `id` and `name` fields were assigned values that are inconsistent with the business logic that I had in mind. It mixed up the updates from two threads. In technical parlance, this situation is called "**race condition**". A race condition refers to a situation where the output of a program depends on the timing of execution of the threads and is thus, unpredictable. The unpredictability of the output in itself is not always a problem unless the outcome is undesirable. In this case, the outcome `Id = 2 Name = Amy` is undesirable as per the business logic. Note that it is possible to get the same output even if I had used non-volatile fields and the `join` method instead of volatiles fields and the `sleep` method.

There are several variations of this problem but they all have the same root cause - the business logic expects a set of instructions to be executed as one single instruction but in reality,

the instructions are executed independently and one thread is able to interfere or is able to observe intermediate values while another thread is in the middle of executing that set of instructions. The set of instructions during which a program is vulnerable to producing undesirable outcomes is called **critical section**.

The solution to this problem is to avoid letting multiple threads execute the critical section simultaneously. If one thread is already in the middle of the critical section, then other threads should wait until this thread leaves the critical section. In Java, this is achieved using a "monitor".

Protecting critical section using a monitor

As I mentioned earlier, every object is associated with a monitor that a thread can lock and unlock. If a thread has already locked a monitor, other threads have to wait if they try to lock the monitor again until the thread that has locked the monitor unlocks it. This is, therefore, ideal for protecting the critical section from being executed by multiple threads simultaneously. The locking and unlocking of the monitor happens automatically when a thread enters and exits out of a synchronized block of code. If you are not from Computer Science background, you can imagine a monitor as a physical door bolt on the inside of a door of a room that contains the code in the critical section. If a thread wants exclusive access to the code, it must lock the bolt after entering so that other threads cannot enter the room. Other threads have to wait at the door if they find it locked. Once a thread is done with the code, it unlocks the bolt and exits the room, giving a chance to another thread to enter. Here is how a monitor can be used to fix the previous example:

```
class Student{
  volatile int id;
  volatile String name;
  void update(int id, String name){
    synchronized(this){
      this.id = id; this.name = name;
    }
  }
}
public class TestClass {
  public static void main(String[] args) throws InterruptedException{
    Student s = new Student();
    Thread t1 = Thread.ofPlatform().start(()->{ s.update(1, "Amy"); });
    Thread t2 = Thread.ofPlatform().start(()->{ s.update(2, "Bob"); });
    Thread.sleep(5000);
    System.out.println("Id = "+s.id+" Name = "+s.name);
  }
}
```

There are a couple of changes in this version. First, I have moved the business logic to a single method in `Student` class (it was duplicated in two lambda expressions earlier) so that there is just one block of code that I need to protect from simultaneous execution from multiple threads instead of two. Second, I have used a **synchronized** block to ensure that only one thread enters the critical section at a time. I have used the monitor associated with the same `Student` object that I want to

update while synchronizing. I will explain why this is important soon. So now, whichever thread locks the monitor first will get exclusive access to the code inside the synchronized block. Even if the OS preempts this thread and schedules another thread to run and if that thread tries to get into the same synchronized block, it will have to wait. In effect, since the other thread cannot run due to the monitor being locked by the first thread, the OS will schedule the first thread to run again and the first thread will get to finish executing the synchronized block. Thus, there is no chance of the two fields getting assigned illogical values. Well, at least by the `update` method.

Using a synchronized method

If you need to protect the entire code inside a method from simultaneous access (as was the case with the update method shown above), you can make the whole method synchronized as follows:

```
synchronized void update(int id, String name){
  this.id = id; this.name = name;
}
```

A synchronized method synchronizes on the monitor associated with object on which the method is invoked. Thus, the above method is equivalent in all respects to the synchronized block used in the earlier example. You may now wonder what happens if a synchronized method calls another synchronized method in the same class. Will the thread be blocked because the monitor is already locked? No. Intrinsic locks are **reentrant**, which means, if a thread has already locked the monitor, it can lock the same monitor again. It would have been a big problem if it were not so. We wouldn't be able to split a long piece of code into multiple small methods as calling the second synchronized method would block the thread permanently. So, each locking action by a thread merely increases the lock count for that monitor and each unlock action, i.e., exiting a synchronized method or a synchronized block, decrements it. Thus, the monitor will be unlocked when the lock count goes back to zero.

You may make a static method synchronized as well. In that case, the method uses the monitor associated with the `java.lang.Class` object that the JVM creates internally to represent the class to which the static method belongs. The JVM creates only one object of Class per class, which means, all synchronized static methods of a class synchronize on the same monitor.

Note that it is common to say, "locking the object" or "synchronizing on the object" instead of saying, "locking the monitor associated with an object" or "synchronizing on the monitor associated with the object". They imply the same thing.

Note

Synchronizing constructors
Marking constructors of a class as synchronized is **not** allowed. As per JLS, "There is no practical need for a constructor to be synchronized, because it would lock the object under construction, which is normally not made available to other threads until all constructors for the object have completed their work."

Although this is not important for the exam, but be aware that it is possible for a reference to a partially constructed object to leak from the thread that is constructing it and be accessible from another thread.

Deciding which lock to use

Deciding which object to use for protecting a critical section is important because a monitor is helpful only if all the threads that are going to execute the critical section synchronize on the same monitor. It doesn't matter which monitor that is but it must be the same. Using different monitors to protect a critical section is like having multiple doors to a room. Locking one door and leaving another open will not prevent other people from entering the room from the other door. For example, the lock in the following update method is technically valid but completely useless:

```
void update(int id, String name){
  synchronized(new Object()){
    this.id = id; this.name = name;
  }
}
```

Since every thread that executes the method creates a new `Object` to lock the critical section, effectively, no thread checks whether the critical section is already locked or not.

On the other hand, using the same monitor to protect multiple unrelated critical sections will cause the program to slow down because in that case, even the threads that do not need to be synchronized with each other will be waiting for the same monitor. It is like one guest locking the entire hotel instead of locking their own room. For example, the lock in the following code is too restrictive:

```
void update(int id, String name){
  synchronized(Student.class){
    this.id = id; this.name = name;
  }
}
```

In the above code, `Student.class` refers to the single `Class` object created automatically by the JVM when it loads the `Student` class. Using this object's monitor is too restrictive because it will prevent threads from updating even different `Student` objects simultaneously, while all we want is to prevent the same `Student` object from being updated by multiple threads simultaneously.

20.3 Coordinating thread execution

Merely executing the two threads simultaneously is of little use unless their execution is well coordinated. Indeed, a consumer thread cannot do much even if it gets a chance to execute unless the producer thread has already produced something. Therefore, it is important to know about the basic techniques to coordinate the execution of threads in a multi-threaded program.
Java provides several means to coordinate the execution of two or more threads. You have already seen a couple of them in the previous section while learning how to share memory with multiple threads. Let's go over them quickly.

20.3.1 Using join

You have already seen the usage of the join method. This is a great way of coordination when one thread requires the termination of another thread before it can continue. This approach is not useful in situations when multiple threads need to coordinate their execution while they are alive.

20.3.2 Using sleep

A sleeping thread does not consume any resource. So, one way for a consumer thread to reduce busy-waiting is to go to sleep for small chunks of time. For example:

```
public static void main(String[] args) throws Exception{
  Factorial f = new Factorial(100000);
  Thread t = Thread.ofVirtual().start(f);
  while(f.result == null){
        Thread.currentThread().sleep(5000);
  }
  System.out.println("Factorial is : "+f.result);
}
```

In the above program, the main thread launches the factorial thread and goes to sleep for 5 seconds. The while loop ensures that if the factorial threads takes longer than 5 seconds to compute, the main thread will go back to sleep again. The only thing better in this approach than the busy-waiting approach is that it doesn't burn CPU as much but the major disadvantage that it has is that it takes a minimum of five seconds even when computing factorial for small numbers. The sleep time can be optimized based on the problem at hand.

Sleep and locks

A thread does not automatically unlock any monitor that it might have locked when it goes to sleep. This limitation makes it hard to use the sleep method for coordination when multiple threads require to lock the same monitor to access a shared resource.

20.3.3 Using interrupt

The `Thread` class has an instance method named `interrupt`, which can be used to grab the attention of a thread. This method works in two ways - if a thread is already in the `RUNNABLE` or `BLOCKED`

state, then this method merely sets the interrupted flag of that thread to `true` and if a thread is in `WAITING` or `TIMED_WAITING` state (such as when the thread is sleeping) then that thread comes out of that state with a `java.lang.InterruptedException` and becomes `RUNNABLE`. The following is how we can use it to address the shortcoming of the previous example:

```
class Factorial implements Runnable{
  int n;
  volatile BigInteger result;
  Thread consumer; //will interrupt this thread when done
  Factorial(int n, Thread consumer){
    this.n = n; this.consumer;
  }
  public void run(){
    BigInteger start = BigInteger.ONE;
    for(int i=1; i<n;i++) start = start.multiply(BigInteger.valueOf(i));
    result = start;
    consumer.interrupt(); //interrupt it up now
  }
}
public class TestClass{
public static void main(String[] args){
  Factorial f = new Factorial(100000, Thread.currentThread()); //pass the current
    thread's reference
  Thread t = Thread.ofVirtual().start(f);
  while(f.result == null){
      try{
        Thread.currentThread().sleep(5000);
      }catch(InterruptedException, i.e., ) {
          //exception is expected here, so, not logging it
      }
  }
  System.out.println("Factorial is : "+f.result);
}
```

In the above program, we pass the main thread's reference to the factorial thread so that the factorial thread can awaken it once the computation is complete. The main thread launches the factorial thread and goes to sleep. The factorial thread, which now has a reference to the sleeping thread, interrupts that thread when it is done with the computation by calling `interrupt()` on that thread. Upon interruption, the main thread comes out of the sleep immediately with an `InterruptedException`. This ensures that the main thread doesn't sleep for full 5 seconds if the computation is finished early.

However, this approach too has a couple of problems:

1. **Clunky coordination** - Since any thread can be scheduled by the JVM to run at any time, it is possible for the factorial thread to finish computation after the main thread checks the result field but before the main thread goes to sleep. In this case, the interrupt signal sent by the factorial will effectively be wasted because the main thread hasn't gone to sleep yet.

This means, the main thread will sleep for at least 5 seconds even though the computation has already been done. Extra code is required to eliminate this inefficiency.

2. **Tight coupling** - By far, the biggest disadvantage of this approach is that the `Factorial` class is now tightly coupled to another `Thread` object. In this particular example, there is only one thread that waits for the result, but what if there are an unknown number of threads that are interested in knowing when the task is complete? Piling on the responsibility of keeping track of the interested parties on the producer thread would be a terrible design.

It is a bit harder to coordinate with a thread that is already in the `RUNNABLE` state using the `interrupt()` method because a `RUNNABLE` thread does not receive an `InterruptedException` when it is interrupted. So, it is the responsibility of the `RUNNABLE` thread to check its interrupted status often and decide what it wants to do when it finds that it has been interrupted. For example, the `run` method of the `Factorial` class can abort the computation if it notices that it has been interrupted as follows:

```
public void run(){
  BigInteger start = BigInteger.ONE;
  for(int i=1; i<n; i++) {
    start = start.multiply(BigInteger.valueOf(i));
    if(i%1000 == 0 && Thread.interrupted()){
      System.out.println("Computation interrupted!");
      break;
    }
  }
  result = start;
  System.out.println("Factorial "+n+" computed.");
  consumer.interrupt();
}
```

I could have used `Thread.currentThread().isInterrupted()` instead of `Thread.interrupted()` in this example, but in some cases `interrupted()` may be more suitable because it clears the interrupted status while `isInterrupted()` does not.

20.3.4 Using wait and notify

Although **wait**, **notify**, and **notifyAll** methods are not on the OCP exam, they provide an efficient technique for thread coordination. Since they are also important for technical interviews, I will cover this topic briefly.

The `wait`, `notify`, and `notifyAll` methods are instance methods of the `Object` class and are thus, available in all objects. Only a thread that has already locked the monitor associated with the object is allowed to call these methods on that object. Let's first see what these methods do and then see their usage.

1. `void wait()` and `void wait(long timeoutMillis)` and `wait(long timeoutMillis, int nanos)` - When a thread calls any of these methods on an object, **three** things happen. **First**, the thread is put on a "waiting list" of threads maintained by the JVM for that object, **second**, the monitor of the object on which this method is called is unlocked, and **third**, this thread

goes into the WAIT state (or the TIMED_WAIT state, if a timeout argument is passed as well). This means, the scheduler will not schedule the calling thread for execution until it is brought back to the RUNNABLE state.
As a consequence of calling this method, any RUNNABLE thread that wants to lock the same monitor can lock it because the monitor is now unlocked.

2. void notify() and void notifyAll() - When a thread calls notify() on an object, the JVM the makes **any one** of the other threads that were put on the waiting list of this object RUNNABLE. It is not guaranteed which thread will be made RUNNABLE. It could be any one of them. The notifyAll() method works the same way except that the JVM makes **all** of the threads that were waiting on this object RUNNABLE.

The trick in using these methods is to understand that just because a thread becomes RUNNABLE doesn't mean it has acquired the locked the monitor and will be able to continue executing the critical section. It just means that the thread is now in contention for locking the same monitor along with other threads that are trying to lock the same monitor. Only one thread will be able to lock the monitor and all others will be put in the BLOCKED state. They will become RUNNABLE again when the monitor is unlocked.

Let me show you how this technique can be used to implement our factorial program:

```
class Factorial implements Runnable{
  int n;
  BigInteger result;
  Factorial(int n){
    this.n = n;
  }
  public synchronized void run(){ //lock this object using synchronized
    BigInteger start = BigInteger.ONE;
    for(int i=1; i<n;i++) {
      start = start.multiply(BigInteger.valueOf(i));
    }
    result = start;
    notifyAll(); //make all threads that are waiting on this object RUNNABLE
  }
}

public class TestClass{
  public static void main(String[] args) {
    Factorial f = new Factorial(200000);
    synchronized(f){ //acquire the lock of the Factorial object
      Thread ft = Thread.ofVirtual().start(f); //start the computation thread
      while(f.result == null){ //use while instead of if
        try{
          f.wait(); //put self on the wait list of this Factorial object
        }catch(InterruptedException e){ e.printStackTrace(); }
      }
    }
```

```
    System.out.println("Factorial is : "+f.result);
  }
}
```

The following is how the above program proceeds:

1. The main thread creates a `Factorial` object and locks it. Since no other thread is trying to lock this `Factorial` object at this point, the main thread is able to lock it.

2. The main thread starts a new thread with the same `Factorial` object as its `Runnable`. However, the `run` method of `Factorial` is synchronized, which means, as soon as the factorial thread is executed, it tries to lock the same `Factorial` object. Since this `Factorial` object is already locked by the main thread, the factorial thread goes from `RUNNABLE` to the `BLOCKED` state immediately.

3. The main thread now checks the `result` field, which, of course, is `null`, because the factorial thread hasn't gotten a chance to do the computation yet. The main thread now calls `wait()` on the same `Factorial` object. It is allowed to do that because it already has the lock of that `Factorial` object. As I mentioned about the `wait` method earlier, the main thread now goes from `RUNNABLE` to the `WAITING` state and the `Factorial` object becomes unlocked.

4. As soon as the `Factorial` object is unlocked, the factorial thread, which was in the `BLOCKED` state, is moved automatically to the `RUNNABLE` state and is also able to lock the `Factorial` object because no other thread is trying to lock it. The scheduler will be able to schedule the factorial thread now.

5. The factorial thread executes the `run` method, completes the computation, and calls `notifyAll()`. This call brings all the threads that were in the waiting list of this object's monitor from `WAITING` to the `RUNNABLE` state. Since there is just one thread, the main thread, on this list, it is made `RUNNABLE` and its call to `wait()` returns. Remember that the call to `wait()` is inside the synchronized block and even if that call returns, the main thread still needs to acquire the lock of the `Factorial` object to execute the next statement in the synchronized block. Since the `Factorial` object is not yet unlocked by the factorial thread, the main thread goes to the `BLOCKED` state immediately after the call to `wait()` returns.

6. The factorial thread exits the synchronized block after the call to `notifyAll()`, thereby unlocking the `Factorial` object.

7. As soon as the `Factorial` object is unlocked, the main thread is moved from `BLOCKED` to the `RUNNABLE` state and locks the `Factorial` object.

8. The main thread runs, exits the synchronized block, and prints `result`. Note that the updated value of `result` is visible to the main thread (even though it is not volatile) due to the guarantee provided by the Java Memory Model that I explained earlier.

This approach has a quite a few advantages over the earlier approaches:

1. The `Factorial` class is not tightly coupled with any other thread. It doesn't even care about who is interested in knowing about the completion of the computation and is not responsible for notifying any thread in particular.

2. It is possible for multiple threads to wait on the completion of the computation.

3. There is no busy-waiting and the result is available as soon as it is computed in all situations.

You might now be wondering about the usage of the `while` loop in the main method instead of an `if` statement. There are four reasons for that:

1. **Spurious wakeups** - Normally, a call to `wait()` returns as a result of another thread calling `notify` or `notifyAll`. However, the specification allows the call to return even without any such trigger. This is called a "spurious wakeup", which means the thread is awakened for no apparent reason.

2. **Timeout** - A wait with a timeout parameter will return after the timeout expires.

3. **Unavailability of the resource** - Remember that there could be multiple threads waiting on an object and all of them will wake up upon a call to `notifyAll`. However, if the number of resources that the threads require to proceed with their business logic are limited, some of them may not be able to get a hand on the resource required to continue their execution. This could happen, for example, when a producer thread produces only one item while there are multiple consumer threads waiting for an item each to become available. Only one consumer will be able to get the item and proceed. Other consumers need to start waiting again.

4. **Receiving an interrupt signal** - The call to `wait` may end up with an `InterruptedException` if any other interrupts this thread. This is the reason for wrapping a call to `wait()` in a try/catch block in the above code.

In all of the four situations described above, the waiting thread wakes up but condition on which the thread was waiting isn't actually fulfilled. Therefore, it is imperative to check that condition again after waking up and start waiting again if the condition is not satisfied.

20.4 Writing thread safe code

As I explained earlier, the actual output of a program cannot always be predicted due to race conditions. But not all such programs are thread unsafe. For example, you cannot predict which user will get the seat if two users try to book the same seat but that doesn't make the application problematic. The application would be problematic if it allowed two people to book the same seat. As a programmer, you need to be able to foresee the possibility of the application generating an undesirable outcome when it is used by multiple people.

Thus, sharing class and instance fields with multiple threads is only one part of writing multithreaded programs. Ensuring that they are shared "safely" is the other. A class or a method is can be deemed thread unsafe if an undesirable outcome can be predicted when it is used from multiple threads. For example, the `Student` class that you saw earlier is not thread safe because its `id` and `name` fields can be modified independently from any thread, rendering the business logic captured in its update method ineffective. Similarly, the `Factorial` class is not thread safe either because any thread can read or modify its `result` field while the factorial is still being computed.

Note that whether an outcome is deemed desirable or not is determined by the business process that is being modelled and implemented.

Thread safety is a major concern in non-trivial applications and while anticipating every possible execution scenario is not always possible, there are several coding techniques that reduce the possibility of multithreading related bugs from creeping into the code. I am listing a few of the important ones here.

20.4.1 Use immutable objects

If an object's state never changes after it is constructed, it can be read from multiple threads without any possibility of reading incorrect or inconsistent values. Thus, immutable classes are inherently thread safe.

Creating immutable classes, however, is easier said than done. If you think that making the instance fields of a class final should be enough to make a class immutable, consider the following `Student` record:
`public record Student(String name, java.util.Date dob){ };` The `Student` record itself doesn't allow any of its fields to be changed but what about the `Date` object pointed to by its `dob` field? Since the `Date` class has several setter methods, it is mutable. Thus, a `Student` object cannot really be considered immutable. This means that just making the fields of a class final or omitting setter methods for its fields may not be enough to make the class immutable. You need to check whether the types of its fields are also immutable or not.

Furthermore, since it is possible for multiple threads to access the `dob` field and modify the same `Date` object simultaneously, the thread safety of the `Student` object depends on the thread safety of the `Date` object. Since the `Date` class is not thread safe (because its documentation doesn't say that it is thread safe), the `Student` record cannot be considered thread safe either.

The above is just one example where a seemingly immutable class turns out be mutable and thread unsafe but there are several things that you need to keep in mind while creating a truly immutable class. Since the OCP exam does not require you to know all those details, I will not go into them in this book. However, I strongly suggest you to read more on this topic from the **Java Secure Coding Guidelines** document published by Oracle in your spare time.

20.4.2 Use final or volatile fields

If you do not expect a field of a class to ever change, then make it final. This prevents accidental misuse of the field. On the other hand, if you expect a field to be mutated by multiple threads directly, then make it volatile to ensure that no thread will see a stale value for that field. Remember that volatile does not ensure atomicity for operations involving multiple fields or for multiple operations on the same field.

20.4.3 Use encapsulation

Although using immutable classes as much as possible is good for thread safety, it would be unrealistic to expect an application to consist only of immutable classes. In cases where object fields can be mutated, you should always use accessor and mutator methods with appropriate access control for its fields. Allowing non-private access to the instance fields of a class is a recipe for disaster because such fields can be modified by any thread at any time without any concern for the business logic of the class. Providing appropriate accessor and mutator methods for instance fields not only allows checks for valid values but also allows incorporation of locks to prevent their modification from multiple threads simultaneously. For example, the following `Square` class is thread safe because the getter and the setter methods synchronize on the same lock:

```
public class Square{
  private int side;
  public synchronized void setSide(int side){
    if(side<0) throw new IllegalArgumentException("Invalid side");
    this.side = side;
  }
  public synchronized int getSide(){return side;}
}
```

20.4.4 Isolate and guard class invariants

An invariant refers to an assertion defined by the business logic that holds true throughout the life of an object of a class. For example, consider the following class:

```
public class Square{
  private int side;
  private int area;
  //synchronized getters and setters for side and area not shown
}
```

If my business logic dictates that the value of the `area` field should always be equal to the value given by the formula side*side, then this rule is an invariant of the `Square` class. Now, what if one thread updates the `side` field and before it could update the `area` field with a value equal to side*side, another thread comes in and modifies the `side` field to something else? In this case, the `side` and the `area` field will become out of sync in terms of the class invariant. The `Square` class, therefore, is not thread safe.

To make a class thread safe, any code that affects the class invariants should be isolated and be guarded with a common lock. For example, the `Square` class can be updated as follows to make it thread safe:

```
class Square{
  private int side;  private int area;
  public synchronized int getSide(){ return side; }
  public synchronized int getArea(){ return area; }
  public synchronized void setSide(int side){
```

```
      this.side = side;  this.area = side*side;
  }
}
```

In the above code, all of the methods are synchronized using the same monitor (the one associated with the same `Square` object). This ensures that the `area` field is always in sync with the `side` field at any instant as demanded by the business logic of the class.

20.4.5 Identify critical sections and guard them with a lock

A common source of bugs in multithreaded code is the logic where an action is performed based on the value of a predicate. Such logic is prone to race conditions. As I explained earlier, a race condition may produce undesirable outcomes. For example: let's say the business logic that you are trying to implement is to check whether the `passengerId` field of a ticket is `null`, and if it is, set it to the current `userId`. Something like this:

```
if(ticket.getPassengerId() == null){
   ticket.setPassengerId(userId);
}
```

Since the checking of the condition and the setting of the `passengerId` are two different operations, one thread could find the `passengerId` of a ticket to be `null` but before it could assign the current `userId` to the `passengerId` field, another thread could come in, find the `passengerId` to be `null` and assign another `userId` to it. If the first thread now proceeds with the assignment, it would be overwriting the value assigned by the second thread. This is clearly undesirable because two users may now be under the impression that they got the ticket.

To fix such bugs we need to identify the actions that form the critical section and execute that critical section as one single **atomic operation**. An atomic operation precludes the possibility of others thread seeing the intermediate states of the objects involved in that operation. They either see the state as it was just before the start of the operation or as it is right after the end of the operation. An atomic operation also prevents the execution of the thread from being interleaved with the execution of other threads that access the same objects.

Identifying the actions that form the critical section is not easy but, basically, a critical section spans the complete chain of actions where every subsequent action requires that the state of the objects involved in the operation is still the same as it was immediately after the previous action. For example, the statement `ticket.setPassengerId(userid)` requires `ticket.getPassengerId()`to stay `null` until it finishes setting the `passengerId`. Thus, the two actions are parts of the same critical section.

A critical section can be made atomic if the objects whose state is being modified are locked at the beginning of the critical section. This will ensure that if one thread has already started executing the critical section another thread will not be allowed to modify the state of that object in any way until the first thread finishes. Something like this:

```
//this method could be in any class
```

```
void bookTicket(Ticket t, User u){
  synchronize(t){ //lock the Ticket object at the beginning of the critical section
    if(ticket.getPassengerId() == null){
      ticket.setPassengerId(userId);
    }
  }
}
```

Note that guarding the critical section is in addition to the guards placed on the methods of the object that is being updated. So, for example, synchronizing the above critical section will be helpful only if the `getPassengerId`and `setPassengerId` methods of the `Ticket` class are also synchronized.

You may wonder at this point that if the ticket object has already been locked at the beginning of the critical section then how it is possible to invoke the `getPassengerId` and `setPassengerId` methods from inside the critical section when they both need to lock the same ticket object. This is possible because intrinsic locks are "**reentrant**". Meaning, if a thread has already locked an object, it can relock that object any number of times. Locking the same object again just increases the lock count of that object and the thread must unlock the monitor the same number of times to make the monitor available for locking to other threads. Thus, in the above example, when a thread calls `ticket.getPassenterId()`, it simply relocks the ticket object and that causes the lock count of the ticket object to increase to two. When the thread exits the `getPassengerId` method, the lock count is reduced back to one. Since the lock count is not zero yet, the ticket object remains locked by the same thread. Other threads will be able to lock the same ticket object once the first thread exits the synchronized block.

20.4.6 Use atomic classes

Many programs routinely use an `int` field for keeping a count of something such as the number active threads, the number of queries fired, or the number of connections, and so on. They also have statements such as `count++;`that contain pre and post increment and decrement operators to manage the count. Such statements actually involve three actions - read the current value of the variable, add 1 to the value, and assign the new value back to the variable. However, these three actions are not executed atomically and so, they may be interleaved with actions performed on the same variable by another thread. For example, there is no guarantee that the following code will print `2` in spite of `counter` being volatile:

```
public class TestClass{
  static volatile int counter = 0;
  public static void main(String[] args) throws Exception{
    Thread t1 = new Thread(()->counter++);
    Thread t2 = new Thread(()->counter++);
    //Use the following two lines instead of the above two lines if you really want to
   test the theory
    //Thread t1 = new Thread(()->{ for(int i = 0; i<20000; i++) { counter++; }});
    //Thread t2 = new Thread(()->{ for(int i = 0; i<20000; i++) { counter++; }});
    t1.start(); t2.start(); t1.join(); t2.join();
    System.out.println("counter = "+counter);
```

```
  }
}
```

As discussed before, if you want multiple threads to increase the counter without losing any increments, you should treat the three actions as a critical section and guard it with a lock. Something like this:

```
class Counter{
  private int i;
  public synchronized void increment(){ i++;  }
  public synchronized int get(){ return i; }
}
public class TestClass{
  static Counter counter = new Counter();
  public static void main(String[] args) throws Exception{
    Thread t1 = new Thread(()->counter.increment());
    Thread t2 = new Thread(()->counter.increment());
    t1.start(); t2.start(); t1.join(); t2.join();
    System.out.println("counter = "+counter.get());
  }
}
```

But why create a new class when you can just use the `java.util.concurrent.atomic.AtomicInteger` class, which comes free with the JDK and provides a lot more functionality than just incrementing an integer value atomically. The `AtomicInteger` class extends `Number` and wraps an `int` value that can be manipulated atomically in many ways. You may check out its API description to know all of its methods but you need to be aware of only the following ones for the exam:

1. `int get()` - Returns the current value.
2. `int getAndIncrement()` - Increments the value but returns the old value. Simulates a post increment, i.e., i++.
3. `int getAndDecrement()` - Decrements the value but returns the old value. Simulates a post decrement, i.e., i--.
4. `int incrementAndGet()` - Increments the value and returns the new value. Simulates a pre increment, i.e., ++i.
5. `int decrementAndGet()` - Decrements the value and returns the new value. Simulates a pre decrement, i.e., --i.

The above program can be updated to use this class as follows:

```
import java.util.concurrent.atomic.AtomicInteger;
public class TestClass{
  static AtomicInteger counter = new AtomicInteger(); //default value is 0
  public static void main(String[] args) throws Exception{
```

```
    Thread t1 = new Thread(()->counter.getAndIncrement()); //ignore the value returned
    by getAndIncrement
    Thread t2 = new Thread(()->counter.getAndIncrement()); //ignore the value returned
    by getAndIncrement
    t1.start(); t2.start(); t1.join(); t2.join();
    System.out.println("counter = "+counter.get());
  }
}
```

Note that an `AtomicInteger` need not be used when it is required only locally within a method such as for a `for` loop because local variables are never shared with other threads.

Besides `AtomicInteger`, the `java.util.concurrent.atomic` package has many other classes including `AtomicLong`, `AtomicBoolean`, and `AtomicReference` that work similarly. You don't need to know the details of all of those classes for the exam but just be aware of them so that you won't be tempted to code one yourself when you need the desired functionality.

20.4.7 Watch out for liveness problems

Synchronizing access to shared objects may sometimes cause a multithreaded program to either appear as stuck or perform sluggishly. Such programs are not exactly thread unsafe in the sense that they don't produce wrong outcomes but are not too useful either because they don't produce any outcome. The culprit is usually irresponsible or uncoordinated acquisition of locks in the application.

Starvation

A thread is considered "starved" when it is unable to proceed with its work for a long time due to lack of availability of a shared object on account of the shared object being locked by other threads most of the time. This could happen due to bad scheduling by the thread scheduler or due to one thread keeping the object locked for a long time. For example, let us say the user asks to refresh the price of a ticket by clicking a button on the GUI and it causes the following code to execute at the backend:

```
synchronized(ticket){
     ticket.setPrice(fetchPriceFromWebsite(ticket.getId()));
}
```

Since this operation locks the ticket, no other thread will be able to book it. This might be okay if this is what the business use case dictates. But what if the fetch call takes a long time or what if the user keeps clicking the Refresh Price button? A thread that tries to book this ticket will have a hard time locking the ticket object.

There is no magic bullet for fixing starvation related bugs but keeping objects locked for short durations will reduce their possibility.

Deadlock

A deadlock occurs when two or more threads are waiting for an action to be taken by each other. For example, if thread `T1` has locked object `O1` and is waiting for object `O2` to become available, while thread `T2` has locked object `O2` and is waiting for object `O1` to become available, neither of the threads will be able to proceed. Such a situation can occur when multiple threads try to acquire multiple locks in different order as illustrated by the following code:

```
void bookTicket(Ticket t, User u){
  synchronize(t){ //lock the Ticket object
    synchronize(u){  //lock the user object
      ticket.setPassengerId(userId);
      u.addTicket(t);
    }
  }
}

void cancelTicket(Ticket t, User u){
  synchronize(u){ //lock the User object
    synchronize(t){  //lock the Ticket object
      ticket.setPassengerId(null);
      u.removeTicket(t);
    }
  }
}
```

If two threads execute the above two methods for the same Ticket and User objects at about the same time, due to an unfortunate interleaving of the two threads, thread 1 could lock the ticket and before it could lock the user, thread 2 could lock the user and proceed to lock the ticket. It is clear that neither of the threads will be able to proceed and the application will appear stuck.

Java does not provide any standard way to prevent deadlocks but a common strategy used to avoid them is to always acquire locks in the same order. In the above example, if both the methods lock the Ticket and the User objects in the same order, a deadlock would not happen.

Livelock

A Livelock is similar to a deadlock except that in a livelock, threads keep executing the same sequence of actions in a loop without actually making any headway in executing the business logic. For example, let's say thread `T1` locks object `O1` and finds that `O2` is already locked. Similarly, thread `T2` locks `O2` and finds that `O1` is already locked. To prevent a deadlock, the threads `T1` and `T2` unlock objects `O1` and `O2` respectively and try to lock objects `O2` and `O1` respectively. So, now, `T1` has `O2` and `T2` has `O1`. This sequence of execution keeps repeating. Thus, the threads appear to be running but neither of the threads is able to lock both the objects and neither of the threads is able to execute the business logic.

Although this situation is not possible while using synchronized blocks because a thread always gets BLOCKED at the synchronized statement if the lock is not available (i.e., there is no

way to take a different path of execution if the object is already locked), it is possible when using the `ReentrantLock` class, which you will learn about soon.

20.5 Using the java.util.concurrent.locks package

The intrinsic locking provided by the `synchronized` keyword is sufficient for most programs but there are a few situations where intrinsic locking is either too rigid in its form or is simply not enough. Specifically, intrinsic locking has the following disadvantages:

1. It can only be used in a block structure such as a code block or a method. This means, you cannot lock a monitor in one method and unlock it in another, or if a block of code locks multiple monitors, they must be unlocked in the reverse order of their acquisition.

2. Intrinsic locks do not allow any control over "fairness". Meaning, out of all the threads that are competing for an intrinsic lock, any thread can lock the monitor, irrespective of how long other threads have been waiting for that lock. A fair lock prioritizes the thread that has been waiting the longest and thus, may prevent thread starvation.

3. Once a thread attempts to lock the monitor, it cannot back out. It will be stuck in the `BLOCKED` state until it gets the lock no matter how long it takes. Even interrupting that thread will not cause it to quit waiting for the lock.

4. Thread pinning - As of Java 21, a virtual thread cannot be unmounted when it is executing a synchronized block. This behavior is called "pinning". Although pinning poses no serious issue if the synchronized block contains short-lived operations but if it includes a long running operation or if the synchronized block is executed frequently, the platform thread will not be available to execute other virtual threads and that will defeat the purpose of using virtual threads.

> **Note**
>
> Although not important for the exam, be aware that a virtual thread is pinned to a platform thread when it executes native code as well just like it is when it executes a synchronized block.

The `java.util.concurrent.locks` package provides `Lock` and `ReadWriteLock` interfaces and their implementations such as `ReentrantLock`and `ReentrantReadWriteLock`, that are more flexible to use than intrinsic locks and also provide a few extra features that one might need while crafting complicated thread coordination logic. They do not cause a virtual thread to be pinned to a platform thread either. These are called **explicit locks**. The OCP exam covers `Lock` and its implementation class `ReentrantLock` only.

The basic usage of an explicit lock is as follows:

```
lock.lock(); //lock refers to a Lock instance
try{
```

```
  //critical section code goes here
}finally{
  lock.unlock();
}
```

Any code that exists between the calls to `lock()` and `unlock()` is executed just like the code within a synchronized block. If the `lock` is not free, a thread will go into one of the blocking states (i.e., `WAITING`, `TIMED_WAITING`, or `BLOCKED`) depending on the `Lock` implementation and thus, will not be scheduled to execute after the call to `lock()`. It will become `RUNNABLE` again only after it acquires the lock, which is when the call to `lock()` returns. Threads that use the same `Lock` instance to coordinate their execution can rely on the **happens-before** guarantees provided by the **Java Memory Model**. Although the `lock()` and `unlock()` methods work about the same as a `synchronized` block, the extra features provided by explicit locks, as evident from the following methods of the `Lock` interface, are not present in intrinsic locks.

1. `void lockInterruptibly()` - Acquires the lock unless the current thread is interrupted.
2. `boolean tryLock()` - Acquires the lock only if it is free at the time of invocation.
3. `boolean tryLock(long time, TimeUnit unit)` - Acquires the lock if it is free within the given waiting time and the current thread has not been interrupted.
4. `Condition newCondition()` - Returns a new `Condition` instance that is bound to this `Lock` instance. A `Condition` provides a coordination mechanism similar to the wait and notify mechanism that I explained earlier.

20.5.1 Using ReentrantLock

The `ReentrantLock` class is a standard implementation of the `Lock` interface. It has two constructors - `ReentrantLock()` and `ReentrantLock(boolean fair)`. The second constructor creates an instance of `ReentrantLock` with the given fairness policy. If it is `true`, the thread waiting the longest will get the lock.

The word "**reentrant**" in its name signifies that if a thread already has a hold on the lock, that thread can lock it again. This is just like the intrinsic locks you saw earlier. Each lock action increments the hold count of the lock and each unlock action decrements it. The `ReentrantLock` class defines a `int getHoldCount()` method that can be called to find out how many holds does the current thread have on this lock. A `ReentrantLock` is not unlocked unless the hold count goes to zero. Thus, a thread must invoke `unlock()` **exactly** the same number of times as it has invoked `lock()` to unlock a `ReentrantLock`. Calling `unlock()` an extra time will cause an `IllegalMonitorStateException` to be thrown.

The following program shows how a `ReentrantLock` can be used:

```
public class TestClass {
  static Lock lock = new ReentrantLock();
  static int count = 0;
```

```
  static Runnable r = () -> {
    lock.lock();
    try{
      count++;
    }finally{
      lock.unlock();
    }
  };

  public static void main(String[] args) throws Exception {
    Thread t1 = Thread.startVirtualThread(r);
    Thread t2 = Thread.startVirtualThread(r);
    boolean gotTheLock = lock.tryLock(1, TimeUnit.SECONDS);
    if(gotTheLock){ //should wrap this block in try/finally
        System.out.println("main thread got the lock");
        lock.unlock();
    }else{
        System.out.println("main thread did not get the lock");
    }
    t1.join(); t2.join();
    System.out.println("Count is "+count);
  }
}
```

The following points are worth noting about the above program:

1. The two virtual threads synchronize using the same `Lock` object and so, only one of them is able to update the `count` at a time. Therefore, we are sure that `count` is incremented atomically.

2. Observe that the block of code after `lock.lock()` is wrapped in a try block and the call to `lock.unlock()` is in the finally block. Although it is not necessary in this case because there is little chance that the code in the try block will ever throw an exception, it is recommended to use explicit locks in this manner to ensure that a lock is not left unlocked if the code in the critical section throws an exception, as that might leave other threads waiting for the lock to be unlocked forever. An intrinsic lock is definitely advantageous in this respect because it is always unlocked automatically whenever and however the control exits a `synchronized` block.

3. Since the main thread uses the `tryLock` method with the time out arguments, it may or may not get the lock depending on whether the lock has already been acquired by another thread and if the other thread releases the lock within the next 1 second. This means, the main thread will not be scheduled for execution for at most 1 second only. I have used `tryLock` in this program for illustration purpose only.

The `ReentrantLock` class defines several useful methods in addition to the methods declared in the `Lock` interface such as `getHoldCount`, `isLocked`, and `isHeldByCurrentThread()` but the exam does not have questions on them.

20.5.2 Quiz

Q. What are the possible outcomes of the following program?

```
public class Quiz {
  static Lock rlck = new ReentrantLock();
  static List items = new ArrayList();
  public static void main(String[] args) {
    Thread.ofPlatform().start(()->{
      rlck.lock();
      if(!items.isEmpty()) System.out.print(items.remove("A"));
      rlck.unlock();
    });

    Thread.ofPlatform().start(()->{
      if(rlck.tryLock()){
        items.add("A");
      } else rlck.unlock();
    });
  }
}
```

Select **2** correct options.
A. It prints `true`
B. It prints `true` as well as a stack trace for an `IllegalMonitorStateException`
C. Does not print `true` but prints a stack trace for an `IllegalMonitorStateException`
D. No output and the program hangs
E. It prints `true` and then hangs
Correct answer is C, D.
You need to keep in mind that the scheduler can schedule any thread to run at any time. So, consider the following three scenarios:

Scenario 1: The first thread started by main runs and finishes before the second thread runs. In this case, the first thread will get the lock. Since the list is empty at this time, it will not print anything. It will then unlock `rlck` and end. The second thread will call `tryLock`, and since `rlck` is unlocked at this point, `tryLock` will return `true`. Therefore, it will enter the `if` block, add `"A"` to the list, and end. Thus, the program will end without printing anything. Note that the second thread will not execute the else part and `rlck` will remain locked but that is not an issue because the program will end anyway.

Scenario 2: The second thread started by main runs and finishes before the first thread runs. In this case, the second thread will get the lock and add `"A"` to the list. However, this thread will end without unlocking `rlck` and so `rlck` will remain locked. Thus, when the first thread runs, it will keep waiting forever for `rlck` to become available. Therefore, in this scenario the program will not print anything and will hang.

Scenario 3: The first thread started by main runs and gets the lock upon calling `rlck.lock()`.

The scheduler now suspends the first thread and runs the second thread. The second thread calls `rlck.tryLock()`but this call will return `false` because the `rlck` is already locked by the first thread. So, the second thread executes `rlck.unlock()` in the `else` part. Since the second thread does not hold the lock for `rlck`, it will get an `IllegalMonitorStateException` and end. The first thread now runs and executes the remaining code without any issue. Thus, in this scenario, the `print` statement will not be executed but the stack trace for the exception generated in the second thread will be printed.

Try to work out the output in the scenario where the second thread runs first and executes `tryLock` but is then suspended by the scheduler immediately afterwards.

20.5.3 Using Condition for thread coordination

Although the OCP exam does not have questions on this topic, I suggest you to read on because the ability to use a `Condition` is a compelling reason for using an explicit lock instead of an intrinsic lock.

A `Condition` object allows one thread to suspend execution by calling its `await()` method until another thread calls `signal()` or `signalAll()`. It works very much like the `Object` class's `wait()` and `notify()`/`notifyAll()` methods. But the `Condition` interface also provides more flexibility in awaiting through its methods `boolean await(long time, TimeUnit unit)`, `long awaitNanos(long nanosTimeout)`, and `void awaitUninterruptibly()`. The names of these methods are self explanatory.
Let me modify the `Factorial` program to use a `ReentrantLock` and a `Condition` to coordinate the execution of the main thread and the factorial thread instead of an intrinsic lock. You should revisit the previous version if you have forgotten how it works because this version follows the same logic.

```
class Factorial implements Runnable{
  int n;
  BigInteger result;
  Lock lock = new ReentrantLock();
  Condition completed = lock.newCondition();
  Factorial(int n){ this.n = n; }
  public void run(){
    lock.lock();
    try{
      BigInteger start = BigInteger.ONE;
      for(int i=1; i<n;i++) {
        start = start.multiply(BigInteger.valueOf(i));
      }
      result = start;
    }finally{
      completed.signal(); //signalAll() will work too
      lock.unlock();
    }
  }
}
```

```
class TestClass{
  public static void main(String[] args) throws Exception{
    Factorial f = new Factorial(100000);
    f.lock.lock();
    try{
      Thread t = Thread.startVirtualThread(f);
      while(f.result==null) f.completed.await();
      System.out.println("Factorial computed "+f.result);
    }finally{
      f.lock.unlock();
    }
  }
}
```

Observe the following points in the above program:

1. A `Condition` object is created from a `Lock` object by calling its `newCondition()` method. Multiple `Condition` objects can be created from the same `Lock` but here, we need just one.

2. The main thread calls `await()` on the `Condition` object after it has already locked the `Lock`. It is typically required that the thread must have locked the `Lock` before it can call methods on the associated `Condition`.

3. Note that `await()` atomically releases the associated lock and suspends the current thread, just like `Object.wait`.

4. The main thread wraps the call to `await()` into a `while` loop because, just like `Object.wait`, `Condition.await` too suffers from spurious wake ups. Remember that even if the thread is woken up, it will not proceed with executing the critical section until it gets the lock.

5. The factorial thread signals the thread that is waiting on the same `Condition` by calling the `signal()` method. This makes main thread `RUNNABLE` again.

6. The main thread reacquires the lock and proceeds with the execution of the statements after the call to `await()` returns.

In this example, the explicit lock and `Condition` do not provide any benefit over the wait/notify approach. In fact, I like the wait/notify approach better because it does not require me to create separate `Lock` and `Condition` objects and share their references with the main thread in an ad hoc fashion. However, the explicit lock and Condition approach is handy and more efficient when you want different threads to wait on different conditions associated with the same lock. I encourage you to go through the example given in the Javadoc description of the `Condition` interface if you have time.

20.6 Using the java.util.concurrent package

Creating threads and implementing thread coordination logic using the Thread API is a low level coding activity that takes the focus of the developer away from coding business logic. It would

be helpful if the developer can focus more on packaging their business logic as "tasks" and use a readymade framework to manage the execution of those tasks instead of "reinventing the wheel" for every project. The **java.util.concurrent** package offers just that. As per its JavaDoc description, this package contains utility classes that are commonly useful in concurrent programming. It includes a few small standardized extensible frameworks, as well as classes that provide useful functionality and are otherwise tedious or difficult to implement. Understanding these frameworks can increase developer productivity tremendously.

This package provide five frameworks for different purposes - **Executors**, **Concurrent Collections**, **Synchronizers**, **Queues**, and **Timing**. Out of these, the OCP exam focuses mostly on Executors and a little bit on Concurrent Collections and Synchronizers. Note that the word "Executors" refers to the name of the framework here. As you will soon learn, there is a class named `Executors` too in the **Executors** framework.

20.6.1 Using the Executors framework

If a `Thread` is like an agent that executes a task, an `Executor` is like a manager that manages the execution of multiple tasks using one or more worker threads that it employs. Once you submit a task to an `Executor`, it is the `Executor`'s responsibility to get that task executed by one of the `Thread`s that it has. This is illustrated by the following code:

```
public static void main(String[] args) {
  Runnable myTask = ()->{ out.println("executing my task..."); };
  Executor e = Executors.newSingleThreadExecutor();
  e.execute( myTask );
  ((ExecutorService )e).shutdown();
}
```

Admittedly, `myTask` can be executed with a single statement `new Thread(myTask).start();` or even better, with `Thread.ofPlatform().start(myTask);` in Java 21, but trust me, you will see the advantages of this framework shortly. Let's first check out the important interfaces and classes of this framework.

Executor - `Executor` is an interface with just one method, `void execute(Runnable r)`, which executes the given `Runnable` at some time in the future. This is actually a root interface that describes what an `Executor` is meant for. It does not provide any method for managing the lifecycle of an `Executor` itself. There is no publicly defined class in the JDK that implements `Executor`. You can, of course, define one yourself but that is usually not required.

ExecutorService - `ExecutorService` is an interface that extends `Executor` (as well as `AutoCloseable` since Java 19). It enhances the `Executor` interface with methods such as `shutdown()`, `shutdownNow()`, and `close()` for managing its termination. There are subtle differences in how these three methods behave.

1. `void shutdown()` - Initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted.

2. `List<Runnable> shutdownNow()` - Attempts to stop all actively executing tasks (typically,

by calling `interrupt()` on its worker threads), halts the processing of waiting tasks, and returns a list of the tasks that were awaiting execution.

3. `void close()` - Initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted. This method waits until all tasks have completed execution and the executor has terminated.

Note that the `shutdown` and `shutdownNow` methods do not cause the current thread (i.e., the thread that calls one of these methods) to block or wait for the shutdown of the `ExecutorService`. But the `close` method, which has been available only since Java 19, does. If you want the current thread to wait for the termination of the `ExecutorService` after calling `shutdown` or `shutdownNow`, then you may call the `awaitTermination(long timeout, TimeUnit unit)` method on the `ExecutorService` instance. Not shutting down an `ExecutorService` through one of these methods may cause its worker threads to remain alive and that, in turn, will keep the Java program from terminating. Some executors such as the one created by the `newWorkStealingPool()` method, do not keep any idling thread and so, do not need shutting down but it is a good idea to do so anyway. Java 19 onwards, using an executor service with a try-with-resources block handles this requirement cleanly.

An `ExecutorService` also allows submission of tasks that return a value through its `Future<T> submit(Callable<T> c)`, `Future<T> submit(Runnable r)` and `Future<T> submit(Runnable r, T result)` methods, where `Callable<T>` and `Runnable` represent the task and `Future<T>` represents the result of type `T` that will be produced after the execution of the task is complete.

Exam Tip

Submission of a task to an `ExecutorService` using its `submit` methods forms the basis of **asynchronous computation** in the Java world. It is called asynchronous because the result of the task is produced only when the execution of the task is complete but the call to `submit` returns immediately with an empty `Future` object, which is filled with the result later. This makes the `submit` methods fundamentally different from the `execute` method.
Another difference, of course, is that the `execute` method accepts only a `Runnable`, but there are overloaded `submit` methods that accept a `Runnable` as well as a `Callable`.

There is no publicly defined class in the JDK that implements `ExecutorService` either. You have to get an instance of an `ExecutorService` through a factory method of the `Executors` class. In fact, all of the factory methods of the `Executors` class actually return an instance of an `ExecutorService`, which is why we are able to cast `e` to `ExecutorService` and invoke `shutdown()` on it in the above code.

Executors - `Executors` is a helper class that knows how to create different kinds of `Executors`. It has a static method for creating each kind. The following are the important ones:

1. `ExecutorService newSingleThreadExecutor()` - Creates an `Executor` that has just one worker thread and it uses that single thread to execute the tasks. Of course, since it uses just one thread, it can execute tasks only sequentially one by one.

2. `ExecutorService newCachedThreadPool()` - Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available.

3. `ExecutorService newFixedThreadPool(int nThreads)` - Creates a thread pool that reuses a fixed number of threads operating off a shared unbounded queue.

4. `ExecutorService newWorkStealingPool()` - Creates a work-stealing thread pool using the number of available processors as its target parallelism level. This executor is quite useful when the tasks that you want to execute are CPU intensive (such as the factorial task) because having more worker threads than the number of CPUs will only cause the extra threads to remain unscheduled for the want of a CPU.

5. `ExecutorService newVirtualThreadPerTaskExecutor()` - (Added In Java 21) Creates an `Executor` that starts a new virtual thread for each task. The number of threads created by the `Executor` is unbounded. This method is equivalent to invoking the `ExecutorService newThreadPerTaskExecutor(ThreadFactory)` method with a thread factory that creates virtual threads.
   Recall that virtual threads are meant to be disposable. You can create as many as you need. Since there is no need to "pool" virtual threads, there is no factory method that returns an `ExecutorService` of virtual threads other than this one.

The `Executors` class also has two overloaded `newScheduledThreadPool` methods to create a `ScheduledExecutorService`, which has methods to execute a task either after a given delay or periodically.

Callable - Just like `java.lang.Runnable`, `java.util.concurrent.Callable<V>` too is a functional interface that is used to represent a task. However, unlike `Runnable`, `Callable` is a generic class. Its functional method `call()` returns a value `V` and also declares `Exception` in its throws clause.

Future - A `Future<V>` represents the result of an asynchronous computation. It has several methods that allow us to check the status of the computation such as `boolean isCancelled()`, `boolean isDone()`, `V get()`, `V get(long timeout, TimeUnit unit)`, and `boolean cancel(boolean mayInterruptIfRunning)`. Its `get()`method is particularly important because it blocks the current thread until the result is ready and provides all of the happens-before guarantees so that the result is visible from any thread that calls this method. It can throw a `java.lang.InterruptedException` if the thread that is waiting for its completion is interrupted and a `java.util.concurrent.ExecutionException` if the execution of the task throws any exception. The `get` with the time out argument works similarly but it can additionally throw a `java.util.concurrent.TimeoutException` (which is also a checked exception) if the result is not ready before the timeout.

Let me now rewrite the factorial example using the concurrent package:

```
class Factorial implements Callable<BigInteger>{
  int n;
  BigInteger result;
```

```
  Factorial(int n){ this.n = n; }
  public BigInteger call(){
    BigInteger start = BigInteger.ONE;
    for(int i=1; i<n;i++) {
      start = start.multiply(BigInteger.valueOf(i));
    }
    result = start;
    return result;
  }
  public static void main(String[] args) throws Exception {
    ExecutorService es = Executors.newWorkStealingPool();
    Factorial fact = new Factorial(100000);
    Future<Biginteger> future = es.submit(fact); //returns immediately
    BigInteger result = future.get(); //blocks until result is ready
    System.out.println(result+"\r\n"+fact.result);
    es.close();
  }
}
```

Observe that there is no special logic in the above code to coordinate the execution of the main thread and the factorial thread. In fact, there is no explicit creation of any thread to compute the factorial. The creation of a new thread and the coordination required to get the result back to the main method is handled transparently by `ExecutorService` and `Future` objects. Since `Future.get()` provides all of the happens-before guarantees, the main thread is able to view the updated value of `fact.result` as well.

`ExecutorService` also allows submission of multiple tasks for execution through its two overloaded `invokeAll`methods. For example:

```
List<Callable<BigInteger>> tasks = new ArrayList<>();
tasks.add(new Factorial(100000));
tasks.add(new Factorial(100000));
List<Future<BigInteger>> results = es.invokeAll(tasks);
System.out.println("All done!");
for(Future f : results) System.out.println(f.get());
```

The call to `invokeAll` blocks until all execution of all of the given tasks is complete. Depending on what type of `ExecutorService` is being used, the tasks may be executed in parallel by multiple threads. If you have time, you should check out the method descriptions of `invokeAll` and `invokeAny` methods in the JavaDoc.

Note

Although not important for the exam, using the right `ExecutorService` for a given situation is crucial for application performance.
In general, to gain maximum throughput from a machine, the number of threads should be about the same as the number of logical cores in the system, if the tasks are CPU intensive. If the details of the production machine are not known in advance, you can rely on `newWorkStealingPool` as it figures out the optimum number of threads for the underlying machine. If the tasks are mostly I/O bound, you can use a `newCachedThreadPool` or a `newFixedThreadPool` with a large number of threads. The best way to arrive at the optimum number of threads is to benchmark the application.
In Java 21, `newVirtualThreadPerTaskExecutor` is the best for tasks that block a lot on I/O and do not use a lot of time in `synchronized` blocks.

Protecting shared state while using an executor

An executor can only free you from managing the creation and execution of threads. It cannot automatically ensure thread safety of any shared state that your tasks may be using. It is your responsibility to write the code for your tasks in a way that accesses shared state in a thread safe manner. This is illustrated by the following code:

```
public class WorkPool {
  int workCount = 0;
  public static void main(String[] args) {
    WorkPool wp = new WorkPool();
    ExecutorService es = Executors.newCachedThreadPool();
    for(int i=0; i<10; i++)  es.submit(()->{
      wp.workCount++;
    });
    es.close();
    System.out.println("Total work done = "+wp.workCount);
  }
}
```

There is no guarantee that the above code will print `10` because the `workCount` field is not thread safe and `wp.workCount++;` is not an atomic operation. To fix the issue, you need to first decide whether you want the shared state to be thread safe or you want the tasks to access the shared state in a thread safe manner. If the former, you can use `AtomicInteger` instead of `int` and if the later, you can wrap the critical section into a synchronized block such as `synchronized(wp) { wp.workCount++; }`. I prefer the first approach but if the code for the shared state is not in your control, the you have to use the second approach.

20.6.2 Using concurrent collections

Commonly used collection classes such as `ArrayList` and `HashMap` from the `java.util` package are not thread safe. This has serious implications for multithreaded applications because these

classes cannot be used directly for some of the frequently required tasks. For example, let's say an online shopping application uses a `HashMap` for storing item quantities and makes the map instance accessible from all the threads through a static field. Something like this:

```
static final Map<String, Integer> itemQuantities = new HashMap<>();
public static void main(String[] args) throws Exception{
  Thread loadQuantitiesT = Thread.ofPlatform().start(()->{
    for(int i = 0; i<1000; i++) itemQuantities.put("ITM"+i, 100);
  });

  Thread printQuantitiesT = Thread.ofPlatform().start(()->{
    itemQuantities.forEach((i, q)->System.out.println(i+" = "+q));
  });
  Thread.sleep(5000);
  int qnt = itemQuantities.get("ITM1");
  if(qnt>0){
    itemQuantities.put("ITM1", qnt-1);
  }
}
```

In the above code, I am doing everything in the same method but remember that, in a real application, these activities will be done from threads created in different classes coded by different people. So, a thread may be using the map while being totally oblivious to the existence of other threads or to what the other threads might be doing with the map. You can expect at least three problematic behaviors from this code:

1. If the second thread starts execution while the first thread is still loading the quantities, the second thread will receive a `java.util.ConcurrentModificationException` because these classes are designed to be "fail-fast", meaning, they refuse to proceed with an operation if they detect even a slight possibility of incorrect results or of corruption of their internal data structures. This is a problem because the quantities may be refreshed or updated at any time while the application is up.

2. Even if the second thread starts execution after the first thread is finished, the second thread may not see any products in the map at all. If you recall the discussion on the visibility of shared memory, there is no guarantee that an update performed by one thread will be visible to another unless a happens-before relationship is established between them. (Pop quiz: What can you do to establish a happens-before relationship between thread 1 and thread 2 in the above code?)

3. The check and update being done by the main thread is not atomic. This means, after the main thread confirms that an item is available, other threads could come in and reduce the quantity of that item to `0` before the main through could complete the order and reduce the quantity by `1` itself.

It is possible to fix these issues by guarding all operations involving the map with one lock. However, to do that, you would either have to expose the lock and then hope that everyone uses the lock before accessing the map or define a thread safe class that wraps the map and use that

class instead. The first option not only requires extra lines of code littered all over the code base but is also way too risky. The second option too requires writing extra code but at least the extra code is in one place and this option also eliminates the possibility of others forgetting to lock the map before using it.

Thankfully, the `java.util.concurrent` package contains a `ConcurrentHashMap` class that solves all the three problems without requiring a single extra line of code. `ConcurrentHashMap` implements `java.util.concurrent.ConcurrentMap`, which, in turn, extends `java.util.Map`. Thus, `ConcurrentHashMap` is a drop in thread safe replacement for a `Map` but in addition to that, it has the following methods (since Java 1.8) that allow you to do check and update operations **atomically**:

1. `V computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction)` - If the specified key is not already associated with a value (or is mapped to null), attempts to compute its value using the given mapping function and enters it into this map unless null.
2. `V computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)` - If the value for the specified key is present and non-null, attempts to compute a new mapping given the key and its current mapped value.
3. `V merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction)` - If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.
4. `V putIfAbsent(K key, V value)` - If the specified key is not already associated with a value, associates it with the given value.

All I have to do now to get the desired behavior in the above code is to replace `HashMap` with `ConcurrentHashMap` and use the following code to update the quantity atomically:

```
Integer newQ = itemQuantities.computeIfPresent("ITM1", (item, currentQty)->{
    int newQty = currentQty - 1;
    if(newQty<0){
      throw new RuntimeException("Item sold out!");
    }else return newQty;
  });
  if(newQ == null)  throw new RuntimeException("No such Item!");
```

Besides `ConcurrentHashMap`, the `java.util.concurrent` package also has `ConcurrentSkipListMap`, `ConcurrentSkipListSet`, `CopyOnWriteArrayList`, and `CopyOnWriteArraySet` classes that provide similar thread safe implementations for `Map`, `Set`, and `List`. The exam does not expect you to know the details of all of these classes but it is good to be aware of them professionally as well as for interview purpose.

Note

You can replace a `HashMap` with a `ConcurrentHashMap` in most cases but not when you need to store `null` key or values because `ConcurrentHashMap` does not allow those.

Using synchronized wrappers for collections

The `java.util.Collections` class has static methods such as `List synchronizedList(List l)` and `Map synchronizedMap(Map m)` that wrap a given list or a map into a thread safe `List` or `Map` implementation class. Using these synchronized wrappers instead of the concurrent versions you saw above solves the first two problems that I explained at the beginning but do not solve the third problem because the return types of these methods is the same as the type of the object you pass in. Therefore, you will not get the methods for performing check and update operations atomically through these wrappers.

Another drawback with the wrapper classes is that they synchronize all read and write operations using a single lock. This technique provides a lower performance as compared to the concurrent versions, which use better techniques internally to provide thread safety as well as performance. Therefore, there is no compelling reason to use the synchronized wrappers instead of the concurrent classes except when you receive an thread unsafe collection (may be as a method parameter) and want to make it accessible through multiple threads.

20.6.3 Using Synchronizers

While developing multi-threaded Java applications, you may find similar requirements across applications and you may end up writing the same kind of code again and again. That's right, they are patterns and they exist while synchronizing multiple threads as well. The java.util.concurrent package provides a few higher level classes, namely, `CyclicBarrier`, `CountdownLatch`, `Semaphore`, `Phaser`, and `Exchanger`, that implement those patterns. These classes provide additional functionality on top of lower level synchronization tools such as the `synchronized`keyword and the explicit Lock classes.

CyclicBarrier

A `CyclicBarrier` allows a predefined number of threads to wait for each other to reach a common point. It has two constructors:

1. `CyclicBarrier(int parties)` - Creates a new `CyclicBarrier` that will trip when the given number of parties (threads) are waiting upon it, and does not perform a predefined action when the barrier is tripped.

2. `CyclicBarrier(int parties, Runnable barrierAction)` - Creates a new `CyclicBarrier` that will trip when the given number of parties (threads) are waiting upon it, and which will execute the given barrier action when the barrier is tripped, performed by the last thread entering the barrier.

It is like a few friends deciding to meet at a common point. All of them reach and wait at that common point until the last friend arrives, after which they all proceed with their remaining activity. Here is the same situation expressed in code:

```
public class Test {
  public static void main(String[] args) {
```

```
    CyclicBarrier cb = new CyclicBarrier(2, ()->{
      System.out.println("Barrier Lifted");
    });
    Thread.ofPlatform().start(new Friend("Paul", cb));
    Thread.ofPlatform().start(new Friend("Peter", cb));
  }

  static record Friend(String name, CyclicBarrier cb) implements Runnable{
    public void run(){
      System.out.println(name+" Arrived.");
      try{
        cb.await();
      }catch(InterruptedException|BrokenBarrierException e){ }
      System.out.println(name+" Proceeding.");
    }
  }
}
```

The above program produces the following output:

```
Paul Arrived.
Peter Arrived.
Barrier Lifed
Peter Proceeding.
Paul Proceeding.
```

Observe the following points in the above program:

1. A `CyclicBarrier` with a party count of `2` is being created. A `Runnable` that will be executed after both the parties arrive is also passed to the constructor. .
2. Each friend thread starts waiting upon calling `cb.await()`.
3. This `Runnable` passed as the barrier action is executed as soon as the last party arrives and before the parties continue to execute. That is why `Barrier Lifted` is printed before the last two lines.
4. Note that the order of arrival of friends is indeterministic as that depends on which thread gets to run first. Thus, the order of the first two lines and the last two lines can be different.

The word **cyclic** in `CyclicBarrier` signifies that the barrier is ready to block the threads again after the specified number of parties have arrived. The exam may test you on this aspect. So, for example, you should know what will happen if three more friends try to await on the same barrier:

```
public static void main(String[] args) {
    ...same as before
  Thread.ofPlatform().start(new Friend("Paul", cb));
  Thread.ofPlatform().start(new Friend("Peter", cb));
  Thread.ofPlatform().start(new Friend("Amy", cb));
  Thread.ofPlatform().start(new Friend("Bella", cb));
```

```
  Thread.ofPlatform().start(new Friend("Cathy", cb));
}
```

The barrier will be lifted after the first two friends arrive and then lifted again after the next two friends arrive. Thus, the barrier action will be executed twice, which means `Barrier Lifted` will be printed twice. The barrier will remain lowered after the fifth friend arrives. Since there are only five friends, the fifth friend will not be able to proceed and the barrier will wait to lift again for ever. There is a lot more to this class than what the above example illustrates, so, try to go through its JavaDoc.

Although all five synchronizers are useful, the exam has questions only on `CyclicBarrier`. So, I am not going to discuss the rest here. But I encourage you to browse through the JavaDoc of all of them because using them whenever the situation warrants will definitely save you a lot of time professionally.

20.7 Exercise

1. Given an int field name sum, start five threads that would each add a random number to this sum. Print the sum when all the threads are done.
2. Do the same as above using an ExecutorService.
3. Enhance the `Factorial` class shown in the chapter using the sleep and interrupt methods such that if the computation is not complete within about five seconds, the computation will be aborted.