Java Concurrency

TABLE OF CONTENTS

PREFACE

This Java Concurrency Cheatsheet is crafted with the intention of providing developers, both novice and experienced, with a concise yet comprehensive resource to navigate the intricacies of concurrent programming in Java. Whether you are just beginning your journey into concurrent programming or seeking to refine your existing skills, this cheatsheet aims to be your reliable companion, offering quick access to essential concepts, best practices, and code snippets.

INTRODUCTION

Java is a powerful and versatile programming language known for its support for concurrent programming. Concurrency allows you to execute multiple tasks in parallel, making your applications more efficient and responsive. However, concurrent programming introduces challenges such as synchronization, thread safety, and avoiding common pitfalls like deadlocks and race conditions.

This Java Concurrency Cheatsheet serves as a quick reference guide to essential concepts, classes, and techniques for writing concurrent Java applications. Whether you're a beginner looking to grasp the basics of multithreading or an experienced developer aiming to optimize performance, this cheatsheet provides a comprehensive overview of key topics.

BASIC CONCEPTS

Let's start by providing a foundation for understanding and working with concurrent programming in Java. Concurrent programming is essential for leveraging the power of modern multi-core processors and creating responsive and efficient applications that can perform tasks concurrently and in parallel.

Concept	Description
Thread	A thread represents an independent path of execution within a Java program. Threads allow for concurrent and parallel execution of code. Java supports multithreading through the `Thread` class.
Runnable	The `Runnable` interface is used for defining the code that can be executed by a thread. It provides a way to encapsulate the task or job that a thread should perform.
Synchronization	Synchronization mechanisms like `synchronized` blocks and methods are used to control access to critical sections of code, preventing multiple threads from accessing them simultaneously.
Locks and Mutexes	Locks (e.g., `ReentrantLock`) are explicit mechanisms used to manage access to shared resources, allowing threads to acquire and release locks for controlled access.
Race Conditions	A race condition occurs when two or more threads access shared data concurrently, and the final result depends on the order of execution, leading to unpredictable behavior. Proper synchronization prevents race conditions.

Concept	Description
Data Race	A data race is a specific type of race condition where two or more threads concurrently access shared data, and at least one of them modifies the data. Data races can result in undefined behavior and should be avoided.
Deadlocks	Deadlocks occur when two or more threads are blocked, waiting for resources that will never be released. Identifying and avoiding deadlocks is essential in concurrent programming.
Atomic Operations	Atomic operations are thread-safe operations that can be performed without interference from other threads. Java provides atomic classes like `AtomicInteger` and `AtomicReference`.
Thread Local Storage	Thread-local storage allows each thread to have its own copy of a variable, which is isolated from other threads. It's useful for storing thread-specific data.
Volatile Keyword	The `volatile` keyword ensures that changes to a variable are visible to all threads. It's used for variables accessed by multiple threads without synchronization.

Concept	Description
Java Memory Model (JMM)	JMM defines the rules and guarantees for how threads interact with memory, ensuring visibility of changes made by one thread to other threads.

JMM "HAPPENS-BEFORE" RELATIONSHIP

The "happens-before" relationship describes the guarantees and constraints JMM applies regarding the order of actions and visibility of memory changes in a multi-threaded environment. It is critical for establishing a consistent and predictable order of operations when we have multiple threads accessing the same resources. It helps prevent issues like data races, ensures that memory changes are visible to other threads when necessary, and provides a foundation for reasoning about the behavior of concurrent Java programs.

A "happens-before" relationship has the following properties:

- **Guarantee of Order**: The "happens-before" relationship establishes a guarantee that actions performed before an action "happens-before" another action, will be seen by other threads in the expected order. It ensures that certain operations are observed as occurring sequentially.
- **Program Order**: Actions within a single thread, as defined by the program order, are always considered to have a "happens-before" relationship. This means that actions within the same thread occur in the order specified by the program, as expected.
- **Synchronization**: Synchronization actions, such as acquiring and releasing locks via `synchronized` blocks or `ReentrantLocks`, create "happens-before" relationships. When a thread releases a lock, all actions performed within the synchronized block are guaranteed to be visible to other threads that subsequently acquire the same lock.
- **Thread Start and Termination**: When a thread starts (via `Thread.start()`) or terminates (via `Thread.join()`), there is a "happens-before"

relationship between the thread's start or termination and actions that occur within that thread.

- **Volatile Variable Access**: Accesses to volatile variables create "happens-before" relationships. When a thread writes to a volatile variable, it guarantees that subsequent reads by other threads will see the most recent write.
- **Transitivity**: "happens-before" relationships are transitive. If action A "happens-before" action B, and action B "happens-before" action C, then action A also "happens-before" action C.

THREADS AND RUNNABLE

The `Thread` class is a fundamental class for creating and managing threads. It allows you to define and run concurrent tasks or processes within your application. Threads represent lightweight, independent paths of execution that can perform tasks concurrently, making it possible to achieve parallelism in your programs.

```
public class MyThread extends Thread
{
    public void run() {
        // Code to be executed by
the thread
        for (int i = 1; i <= 5; i++)
{
            System.out.println
("Thread: " + Thread.
currentThread().getId() + " Count: "
+ i);
        }
    }

    public static void main(String[]
args) {
        // Create two threads
        MyThread thread1 = new
MyThread();
        MyThread thread2 = new
MyThread();

        // Start the threads
        thread1.start();
        thread2.start();
    }
}
```

The `Runnable` interface is a functional interface that represents a task or piece of code that can be executed concurrently by a thread. It provides a way to define the code that a thread should run without the need to explicitly extend the `Thread` class. Implementing the `Runnable` interface allows for better separation of concerns and promotes reusability of code.

```
public class MyRunnable implements
Runnable {
    public void run() {
        // Code to be executed by
the thread
        for (int i = 1; i <= 5; i++)
{
            System.out.println
("Thread: " + Thread.
currentThread().getId() + " Count: "
+ i);
        }
    }

    public static void main(String[]
args) {
        // Create two Runnable
instances
        MyRunnable runnable1 = new
MyRunnable();
        MyRunnable runnable2 = new
MyRunnable();

        // Create threads and
associate them with Runnable
instances
        Thread thread1 = new Thread
(runnable1);
        Thread thread2 = new Thread
(runnable2);

        // Start the threads
        thread1.start();
        thread2.start();
    }
}
```

Thread states represent the different phases or conditions that a thread can be in during its lifecycle:

Thread State	Description
NEW	A thread is in the `NEW` state when it has been created but has not yet started executing. It is not yet eligible to run and has not yet acquired any system resources.
RUNNABLE	A thread is in the `RUNNABLE` state when it is eligible to run, and the Java Virtual Machine (JVM) has allocated resources for its execution. However, it may not be currently executing.
BLOCKED	A thread is in the `BLOCKED` state when it is waiting to acquire a monitor lock to enter a synchronized block or method. It is blocked by another thread holding the lock.
WAITING	A thread is in the `WAITING` state when it is waiting for a specific condition to be met before it can proceed. It may be waiting indefinitely until notified by another thread.
TIMED_WAITING	Similar to the `WAITING` state, a thread in the `TIMED_WAITING` state is waiting for a specific condition. However, it has a timeout and will automatically transition to `RUNNABLE` after the timeout expires.
TERMINATED	A thread is in the `TERMINATED` state when it has completed its execution or has been explicitly terminated. Once terminated, a thread cannot be restarted or run again.

Thread lifecycle methods:

Method	Description
`start()`	Initiates the execution of a thread by invoking its `run()` method. When `start()` is called, the thread transitions from the `NEW` state to the `RUNNABLE` state and begins execution concurrently. It's the primary method for starting a new thread.
`wait()`	Used to make a thread voluntarily give up the monitor lock it holds. It should be called from within a synchronized block or method. The thread enters the `WAITING` state and releases the lock until it's notified by another thread.
`notify()` / `notifyAll()`	Used to wake up one / all of the threads that are waiting using the `wait()` method on the same object. It allows one / all waiting threads to transition back to the `RUNNABLE` state, giving them a chance to proceed.

Method	Description
`join()`	Allows one thread to wait for the completion of another thread. When a thread calls `join()` on another thread, it will block until the target thread finishes its execution.
`yield()`	Suggests to the JVM that the current thread is willing to yield its current CPU time to allow other threads to run. It's a hint, and the actual behavior depends on the JVM's implementation.
`sleep()`	Pauses the execution of the current thread for a specified amount of time (in milliseconds). It allows you to introduce delays in your program, often used for timing purposes.
`interrupt()`	Interrupts the execution of a thread by setting its interrupt status. It can be used to request a thread to gracefully terminate or to handle the interruption in a custom way. If the thread is waiting, sleeping, or blocked an `InterruptedException` is thrown. In case you catch the exception at the interrupted thread level, set its interrupt status manually by calling `Thread.currentThread().interrupt()` and throw the exception in order to be handled at a higher level.

SYNCHRONIZATION

THE SYNCHRONIZED KEYWORD

The `synchronized` keyword is used to create synchronized blocks of code, which ensure that only one `Thread` can execute them at a time. It provides a way to control access to critical sections of your program, preventing multiple threads from accessing them simultaneously.

To enter a synchronized block, one must acquire a lock on an object's monitor. An object's monitor is a synchronization mechanism that provides locking functionality on `Object` instances. After doing so, all code included in the block can be manipulated exclusively and atomically. Upon exiting the `synchronized` block the lock is returned to the object's monitor for other threads to acquire. If the lock cannot be acquired immediately, the executing `Thread` waits until it becomes available.

```
public class
SynchronizedBlockExample {
    private int count = 0;
    private Object lock = new
Object(); // A lock object for
synchronization

    public void performTask() {
        synchronized (lock) { //
Synchronized block using the 'lock'
object
            for (int i = 0; i <
1000; i++) {
                count++;
            }
        }
    }
}
```

The `synchronized` keyword can be also specified on a method level. For non static methods, the lock is acquired from the monitor of the `Object` instance that the method is a member of, or for static methods, the `Class` object monitor of the `Class` with the method.

```
public class SynchronizedExample {
```

```
    private int count = 0;

    // Synchronized instance method
    public synchronized void
increment() {
        count++;
    }

    // Synchronized static method
    public static synchronized void
decrement() {
        count--;
    }

}
```

The lock is reentrant, so if the thread already holds the lock, it can successfully acquire it again.

```
class Reentrantcy {
    private int count = 0;

    public synchronized void doAll()
{
        increment();
        decrement();
    }

    public synchronized void
increment() {
        count++;
    }

    public synchronized void
decrement() {
        count--;
    }
}
```

WAIT()/NOTIFY()/NOTIFYALL()

The most common pattern for synchronizing access to functionality/resources using `wait()`, `notify()`, `notifyAll()` methods is a condition loop. Let's see an example that demonstrates the usage of `wait()` and `notify()` to coordinate two threads to print alternate numbers:

```
public class WaitNotifyExample {
    private static final Object lock
= new Object();
    private static boolean isOddTurn
= true;

    public static void main(String[]
args) {
        Thread oddThread = new
Thread(() -> {
            for (int i = 1; i <= 10;
i += 2) {
                synchronized (lock)
{
                    while
(!isOddTurn) {
                        try {
                            lock
.wait(); // Wait until it's the odd
thread's turn
                        } catch
(InterruptedException e) {
                            Thread
.currentThread().interrupt();
                        }
                    }
                    System.out
.println("Odd: " + i);
                    isOddTurn =
false; // Satisfy the waiting
condition
                    lock.notify();
// Notify the even thread
                }
            }
        });

        Thread evenThread = new
Thread(() -> {
            for (int i = 2; i <= 10;
i += 2) {
                synchronized (lock)
{
                    while (
isOddTurn) {
                        try {
                            lock
.wait(); // Wait until it's the even
```

```
thread's turn
                                } catch
(InterruptedException e) {
                                    Thread
.currentThread().interrupt();
                                }
                            }
                            System.out
.println("Even: " + i);
                            isOddTurn =
true; // Satisfy the waiting
condition
                            lock.notify();
// Notify the odd thread
                        }
                    }
        });

        oddThread.start();
        evenThread.start();
    }
}
```

Things to notice:

- In order to use `wait()`, `notify()`, `notifyAll()` on an object, you need to acquire the lock on this object first - both our threads synchronize on the `lock` object to acquire its lock.
- Always wait inside a loop that checks the condition being waited on. This addresses the timing issue if another thread satisfies the condition before the wait begins and also protects your code from spurious wake-ups - both our threads wait inside a loop governed by the `isOddTurn` flag.
- Always ensure that you satisfy the waiting condition before calling `notify()` / `notifyAll()`. Failing to do so will cause a notification but no thread will ever be able to escape its wait loop - both our threads satisfy the `isOddTurn` flag for the other thread to continue.

THE VOLATILE KEYWORD

When a variable is declared as `volatile`, it guarantees that any read or write operation on that variable is directly performed on the main memory, ensuring atomic updates and visibility of changes to all threads. In other words, there JMM applies a "happens-before" relationship for the events "write to a volatile variable" and any subsequent "read from the volatile variable". Therefore, any subsequent reads of the variable will see the value that was set by the most recent write.

```
public class VolatileExample {
    private static volatile boolean
flag = false;

    public static void main(String[]
args) {
        Thread writerThread = new
Thread(() -> {
            try {
                Thread.sleep(1000);
// Simulate some work
            } catch
(InterruptedException e) {
                Thread.
currentThread().interrupt();
            }
            flag = true; // Set the
flag to true
            System.out.println("Flag
set to true by writerThread.");
        });

        Thread readerThread = new
Thread(() -> {
            while (!flag) {
                // Busy-wait until
the flag becomes true
            }
            System.out.println("Flag
is true, readerThread can
proceed.");
        });

        writerThread.start();
        readerThread.start();
    }
}
```

THE THREADLOCAL CLASS

`ThreadLocal` is a class that provides thread-local variables. A thread-local variable is a variable that is unique to each thread, meaning that each thread accessing a `ThreadLocal` variable gets its own independent copy of that variable. This can be useful when you have data that needs to be isolated and maintained separately for each thread, but also reduce contention for shared resources, which usually leads to performance bottlenecks. It's commonly used to store values like user sessions, database connections, and thread-specific state without explicitly passing them between methods.

Here's a simple example of how to use `ThreadLocal` to store and retrieve thread-specific data:

```
public class ThreadLocalExample {
    private static ThreadLocal
<Integer> threadLocal = ThreadLocal
.withInitial(() -> 0);

    public static void main(String[]
args) {
        // Create and start three
threads
        Thread thread1 = new
Thread(() -> {
            threadLocal.set(1); //
Set a thread-specific value
            System.out.println
("Thread 1: " + threadLocal.get());
// Get the thread-specific value
        });

        Thread thread2 = new
Thread(() -> {
            threadLocal.set(2);
            System.out.println
("Thread 2: " + threadLocal.get());
        });

        Thread thread3 = new
Thread(() -> {
            threadLocal.set(3);
            System.out.println
("Thread 3: " + threadLocal.get());
        });

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

IMMUTABLE OBJECTS

An immutable object is an object whose state cannot be modified after it is created. Once an immutable object is initialized, its internal state remains constant throughout its lifetime. This property makes immutable objects inherently thread-safe because they cannot be modified by multiple threads simultaneously, eliminating the need for synchronization.

Creating an immutable object involves several key steps:

- **Make the class `final`**: To prevent inheritance and ensure that the class cannot be subclassed.
- **Declare all fields as `final`**: Mark all instance variables as `final` to make sure they are initialized only once, typically within the constructor.
- **No setter methods**: Do not provide setter methods that allow the modification of the object's state.
- **Safe publication**: `this` reference does not escape during construction.
- **No mutable objects**: If the class contains references to mutable objects (objects that can change their state), ensure that those references are not exposed or allow external modification.
- **Make all fields private**: Encapsulate the fields by making them private to restrict direct access.
- **Return a new object in methods that modify state**: Instead of modifying the existing object, create a new object with the desired changes and return it.

```
public final class ImmutablePerson {
    private final String name;
    private final int age;
    private final List
<ImmutablePerson> family;
```

```
    public ImmutablePerson(String
name, int age, List<ImmutablePerson>
family) {
        this.name = name;
        this.age = age;
        // Defensive copy
        List<ImmutablePerson> copy =
new ArrayList<>(family);
        // Making mutable collection
unmodifiable
        this.family = Collections
.unmodifiableList(copy);
        // 'this' is not passed to
anywhere during construction
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    // No setter methods, and fields
are final

    // Instead of modifying the
object, return a new object with the
desired changes
    public ImmutablePerson withAge
(int newAge) {
        return new ImmutablePerson
(this.name, newAge);
    }

    // No toString, hashCode and
equals methods for simplicity
}
```

DEADLOCK, LIVELOCK AND THREAD STARVATION

DEADLOCK

Deadlock is a situation where two or more threads are unable to proceed with their execution because they are each waiting for the other(s) to release a resource or a lock. This results in a standstill where none of the threads can make progress. Deadlocks are typically caused by improper synchronization or resource allocation against resources that causes blocking. Lest see an example of a deadlock scenario involving two threads and two locks:

```
public class DeadlockExample {
    private static final Object
lock1 = new Object();
    private static final Object
lock2 = new Object();

    public static void main(String[]
args) {
        Thread thread1 = new
Thread(() -> {
            synchronized (lock1) {
                System.out.println
("Thread 1: Holding lock 1...");
                try { Thread.sleep
(100); } catch (InterruptedException
e) {}
                System.out.println
("Thread 1: Waiting for lock 2...");
                synchronized (lock2)
{
                    System.out
.println("Thread 1: Acquired lock
2.");
                }
            }
        });

        Thread thread2 = new
Thread(() -> {
            synchronized (lock2) {
                System.out.println
("Thread 2: Holding lock 2...");
                try { Thread.sleep
(100); } catch (InterruptedException
e) {}
                System.out.println
("Thread 2: Waiting for lock 1...");
                synchronized (lock1)
{
                    System.out
.println("Thread 2: Acquired lock
1.");
```

```
                }
            }
        });

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

In this example:

- thread1 acquires lock1 and then waits for lock2.
- thread2 acquires lock2 and then waits for lock1.

Both threads are now waiting for a resource held by the other, resulting in a deadlock. The program will hang indefinitely.

OVERCOMING DEADLOCK

Deadlocks can be avoided or resolved by various techniques:

- **Use a Timeout**: Set a timeout for acquiring locks. If a thread cannot acquire a lock within a specified time, it can release any locks it holds and retry or abort. This functionality can be easily implemented using ReentrantLock from the java.util.concurrent.locks package.
- **Lock Ordering**: Establish a consistent order for acquiring locks across all threads to prevent circular waiting as seen in the example below.
- **Resource Allocation Graph**: Use algorithms like the resource allocation graph to detect and recover from deadlocks.
- **Design for Deadlock Avoidance**: Design your multi-threaded code to minimize the potential for deadlocks, such as using higher-level abstractions like the java.util.concurrent classes.

```
import
java.util.concurrent.TimeUnit;
import
java.util.concurrent.locks.Lock;
import
java.util.concurrent.locks.Reentrant
Lock;
```

```
public class
DeadlockResolutionExample {
    private static final Lock lock1
= new ReentrantLock();
    private static final Lock lock2
= new ReentrantLock();

    public static void main(String[]
args) {
        Runnable acquireLocks = ()
-> {
            lock1.lock();
            try {
                System.out.println
(Thread.currentThread().getName() +
": Holding lock 1...");
                try {
                    Thread.sleep(
100);
                } catch
(InterruptedException e) {
                }
                System.out.println
(Thread.currentThread().getName() +
": Waiting for lock 2...");

                // Attempt to
acquire lock2 with a timeout of 500
milliseconds
                boolean
acquiredLock2 = lock2.tryLock(500,
TimeUnit.MILLISECONDS);
                if (acquiredLock2) {
                    try {
                        System.out
.println(Thread.currentThread().getN
ame() + ": Acquired lock 2.");
                    } finally {
                        lock2.
unlock();
                    }
                } else {
                    System.out
.println(Thread.currentThread().getN
ame() + ": Timeout while waiting for
lock 2.");
                }
            } finally {
                lock1.unlock();
```

```
            }
        };

        // Consistent order for
acquiring locks and use of timeouts
        Thread thread1 = new Thread
(acquireLocks);
        Thread thread2 = new Thread
(acquireLocks);

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

LIVELOCK

Livelock is a situation where two or more threads are actively trying to resolve a conflict but end up causing repeated state changes without making any progress. In a livelock, threads are not blocked but are busy responding to each other's actions, and the system remains in an undesirable state.

THREAD STARVATION

Thread starvation occurs when a thread is unable to make progress because it is constantly waiting for a resource or access to a critical section that is always being acquired by other threads. This results in the affected thread not getting a fair share of CPU time.

THE JAVA.UTIL.CONCURRENT PACKAGE

The `java.util.concurrent` package provides a wide range of classes and interfaces that support concurrent and multithreaded programming. These classes offer high-level abstractions for managing threads, synchronization, and concurrent data structures, making it easier to write efficient and thread-safe code. Here's an overview of some of its most popular classes and interfaces.

EXECUTOR & EXECUTORSERVICE

`Executor` is an interface that represents an object capable of executing tasks asynchronously. It decouples the task submission from task execution.

`ExecutorService` is a subinterface of `Executor` that extends the functionality by providing methods for managing the lifecycle of the executor and controlling the execution of tasks. In other words, `ExecutorService` is the core interface for thread pools.

`ExecutorService` implementation classes offer various ways to manage and execute tasks concurrently, each with its own advantages and use cases. You can find the most commonly used in the table below. Choose the appropriate implementation based on your specific requirements, but remember, when sizing thread pools, it is often useful to base their size on the number of logical cores the machine running your code has. You can get that value by calling `Runtime.getRuntime().availableProcessors()`

Executorservice Implementation	Description
`ThreadPoolExecutor`	A versatile and customizable executor service that allows you to create thread pools with specified core and maximum thread counts, custom thread factory, and more.
`ScheduledThreadPoolExecutor`	Extends `ThreadPoolExecutor` to provide scheduling capabilities for executing tasks at specific times or intervals.
`ForkJoinPool`	A specialized `ExecutorService` designed for parallel execution, particularly suited for recursive tasks and algorithms using the Fork-Join framework.
`WorkStealingPool`	An implementation of `ForkJoinPool` that uses a work-stealing algorithm for efficiently distributing tasks among worker threads.

Executorservice Implementation	Description
`SingleThreadExecutor`	Creates an executor service with a single worker thread, suitable for sequentially executing tasks one at a time.
`FixedThreadPool`	A fixed-size thread pool executor that manages a predetermined number of worker threads, ideal for a fixed workload.
`CachedThreadPool`	A thread pool executor that can adaptively adjust the number of threads based on task demand, suitable for short-lived and bursty tasks.
`SingleThreadScheduledExecutor`	Creates a single-threaded scheduled executor, which allows scheduling tasks for execution at specific times or with fixed-rate intervals.
`FixedScheduledThreadPool`	A fixed-size thread pool with scheduling capabilities, combining features of a fixed-size thread pool with task scheduling.

Additionally, `java.util.concurrent` provides the `Executors` class which contains static factory methods for easily creating the aforementioned thread pool types and more.

Available task types are shown in the table below.

Task Type	Description
Runnable Tasks	Runnable tasks are simple, non-returning tasks that implement the `Runnable` interface and perform actions without producing a result.
Callable Tasks	Callable tasks are similar to runnables but can return a result or throw an exception. They implement the `Callable<V>` interface.
Asynchronous Tasks	Asynchronous tasks are often represented by the `Future<V>` interface and can run independently of the calling thread. `FutureTask` is a concrete implementation of `Future` that allows you to wrap a `Callable` or `Runnable` and use it with executors.

Tasks are submitted to the executor service using `ExecutorService#submit`, `ExecutorService#invokeAll`, or `ExecutorService#invokeAny`.

Most methods of the `ExecutorService` return `Future<V>` instances. `Future` is an interface that represents the result of an asynchronous computation. It exposes methods to examine if the computation is complete or block until the result is available. Below is an example.

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

public class ExecutorServiceExample {

    public static void main(String[] args) {
        // Create an ExecutorService using a fixed-size thread pool with 2 threads.
        ExecutorService executorService = Executors.newFixedThreadPool(2);

        // Define a Runnable task
```

```
        Runnable runnableTask = ()
-> {
            String threadName =
Thread.currentThread().getName();
            System.out.println("Task
1 executed by " + threadName);
        };

        // Define a list of Callable
tasks
        List<Callable<String>>
callableTasks = List.of(
            () -> {
                String threadName =
Thread.currentThread().getName();
                return "Task 2
executed by " + threadName;
            },
            () -> {
                String threadName =
Thread.currentThread().getName();
                return "Task 3
executed by " + threadName;
            }
        );

        // Submit the task to the
ExecutorService
        executorService.submit
(runnableTask);

        try {
            // Use invokeAll to
submit a list of Callable tasks and
wait for all tasks to complete.
            List<Future<String>>
futures = executorService.invokeAll
(callableTasks);

            // Print the results of
the Callable tasks, call to get()
waits until the result is available
            for (Future<String>
future : futures) {
                System.out.println
(future.get());
            }

            // Use invokeAny to
submit a list of Callable tasks and
wait for the first completed task.
            String firstResult =
executorService.invokeAny(callableTa
sks);
            System.out.println
("First completed task: " +
firstResult);
        } catch (Exception e) {
            e.printStackTrace();
        }

        // Shutdown the
ExecutorService to stop accepting
new tasks
        executorService.shutdown();
    }
}
```

SEMAPHOR

The `Semaphore` class is a synchronization primitive that allows a fixed number of threads to access a resource or a section of code concurrently. This is especially useful for scenarios where you want to limit concurrency, manage access to a pool of resources, or protect a critical section of code. `Semaphore` is initialized with a count, a set of permits. Threads may call `acquire()` to acquire a permit. Each `acquire()` blocks if necessary until a permit is available, and then takes it. Threads may call `release()` to add a permit, potentially releasing a blocking acquirer.

COUNTDOWNLATCH

`CountDownLatch` is a synchronization construct that allows one or more threads to wait for a set of operations to complete before they proceed. `CountDownLatch` is initialized with a count, the number of operations needed to be completed before a thread is allowed to continue. Threads may call `await()` to wait for the count to reach 0 and then proceed. Threads may call `countDown()` to reduce the count by one when they complete an operation.

CYCLICBARRIER

`CyclicBarrier` is a synchronization barrier that allows a set of threads to wait for each other to reach a common point before continuing execution. It's commonly used to synchronize multiple threads that perform different subtasks and need to wait for each other before proceeding. `CyclicBarrier` is initialized with a count, the number of threads to wait before continuing, and a function called when the count is reached and threads are allowed to continue. Threads may call `await()` to wait for the count to reach the designated number before allowed to proceed operations.

CONCURRENT COLLECTIONS

These concurrent collection classes provide thread-safe data structures for various use cases, allowing multiple threads to access and modify data concurrently while ensuring data consistency and minimizing contention. The choice of which class to use depends on the specific needs of your concurrent application.

Concurrent Collection Class	Description
`ConcurrentHashMap`	A highly concurrent, thread-safe implementation of the `Map` interface, designed for efficient read and write operations in multithreaded environments.
`ConcurrentSkipListMap`	A concurrent, sorted map that is based on a skip list data structure, providing concurrent access and sorted order.
`BlockingQueue` (`LinkedBlockingQueue`, `DelayQueue`, `PriorityBlockingQueue`, `SynchronousQueue`)	Blocking queues are thread-safe, bounded or unbounded queues that support blocking operations for producer-consumer scenarios. In `DelayQueue` elements are removed based on their delay, in `PriorityBlockingQueue` based on a `Comparator` and in `SynchronousQueue` an element is removed only when a new one has arrived.
`ConcurrentLinkedQueue`	A thread-safe, non-blocking, and unbounded queue based on a linked node structure, suitable for high-concurrency producer-consumer scenarios.
`ConcurrentLinkedDeque`	A thread-safe, non-blocking, double-ended queue that supports concurrent access and modifications from both ends.
`CopyOnWriteArrayList`	A list that creates a new copy of its internal array whenever a modification is made, ensuring thread safety for read-heavy workloads.
`CopyOnWriteArraySet`	A thread-safe set that is backed by a `CopyOnWriteArrayList`, providing thread safety for read-heavy sets.
`ConcurrentSkipListSet`	A concurrent, sorted set that is based on a skip list data structure, providing concurrent access and sorted order.

ATOMICS

The `java.util.concurrent.atomic` package provides classes that support atomic operations on single variables. These classes are designed to be used in multi-threaded applications to ensure that operations on shared variables are performed atomically without the need for explicit synchronization. This helps avoid data races and ensures thread safety.

Common Atomic Classes:

- `AtomicInteger`: An integer value that can be atomically incremented, decremented, or updated.
- `AtomicLong`: A long value that supports atomic operations.
- `AtomicBoolean`: A boolean value with atomic operations for setting and getting.
- `AtomicReference`: A generic reference type that supports atomic updates.
- `AtomicStampedReference`: A variant of `AtomicReference` that includes a version stamp to detect changes.
- `AtomicIntegerArray`, `AtomicLongArray`, `AtomicReferenceArray`: Arrays of atomic values.

They are suitable for scenarios where you need to perform operations like increment, compare-and-set, and update on variables without risking data corruption due to concurrent access. Here's a simple example using `AtomicInteger` to demonstrate atomic operations.

```
import
java.util.concurrent.atomic.AtomicIn
teger;

public class AtomicExample {
    public static void main(String[]
args) {
        AtomicInteger atomicInt =
new AtomicInteger(0);

        // Increment the atomic
integer atomically
        int incrementedValue =
atomicInt.incrementAndGet();
        System.out.println
("Incremented value: " +
incrementedValue);

        // Add a specific value
atomically
        int addedValue = atomicInt
.addAndGet(5);
        System.out.println("Added
value: " + addedValue);

        // Compare and set the value
atomically
        boolean updated = atomicInt
.compareAndSet(10, 15);
        System.out.println("Value
updated? " + updated);

        // Get the current value
        int currentValue =
atomicInt.get();
        System.out.println("Current
value: " + currentValue);
    }
}
```

LOCKS

Locks provide more flexible and advanced locking mechanisms compared to `synchronized` blocks, including features like reentrancy, fairness, and read-write locking. The `java.util.concurrent.locks` package contains two interfaces, `Lock` and `ReadWriteLock` and their implementation classes `ReentrantLock` and `ReentrantReadWriteLock` respectively.

`ReentrantLock` is a reentrant mutual exclusion lock with the same basic behavior as `synchronized` blocks but with additional features. It can be used to control access to a shared resource and provides more flexibility and control over locking such as obtaining information about the state of the lock, non-blocking `tryLock()`, and interruptible locking. In this example, we use a `ReentrantLock` to protect a critical section of code.

```
import
java.util.concurrent.locks.Reentrant
```

```
Lock;

public class ReentrantLockExample {
    private static ReentrantLock
lock = new ReentrantLock();

    public static void main(String[]
args) {
        Runnable task = () -> {
            lock.lock(); // Acquire
the lock
            try {
                System.out.println
("Thread " + Thread.currentThread
().getId() + " has acquired the
lock.");
                // Perform some
critical section operations
                Thread.sleep(1000);
            } catch
(InterruptedException e) {
                Thread.
currentThread().interrupt();
            } finally {
                lock.unlock(); //
Release the lock
                System.out.println
("Thread " + Thread.currentThread
().getId() + " has released the
lock.");
            }
        };

        // Create multiple threads
to access the critical section
        for (int i = 0; i < 3; i++)
{
            new Thread(task).
start();
        }
    }
}
```

ReentrantReadWriteLock provides separate locks for reading and writing. It's used to allow multiple threads to read a shared resource simultaneously, while ensuring that only one thread can write to the resource at a time. Here's an example.

```
import
java.util.concurrent.locks.ReadWrite
Lock;
import
java.util.concurrent.locks.Reentrant
ReadWriteLock;

public class ReadWriteLockExample {
    private static ReadWriteLock
readWriteLock = new
ReentrantReadWriteLock();
    private static String sharedData
= "Initial Data";

    public static void main(String[]
args) {
        Runnable reader = () -> {
            readWriteLock.
readLock().lock(); // Acquire the
read lock
            try {
                System.out.println
("Reader Thread " + Thread
.currentThread().getId() + " is
reading: " + sharedData);
                // Reading shared
data
            } finally {
                readWriteLock
.readLock().unlock(); // Release the
read lock
            }
        };

        Runnable writer = () -> {
            readWriteLock.
writeLock().lock(); // Acquire the
write lock
            try {
                sharedData = "New
Data";
                System.out.println
("Writer Thread " + Thread
.currentThread().getId() + " is
writing: " + sharedData);
                // Writing to the
shared data
            } finally {
```

```
                readWriteLock
.writeLock().unlock(); // Release
the write lock
            }
        };

        // Create multiple reader
and writer threads
        for (int i = 0; i < 3; i++)
{
            new Thread(reader).
start();
        }
        for (int i = 0; i < 2; i++)
{
            new Thread(writer).
start();
        }
    }
}
```