In Java, a thread is a lightweight process or a path of execution within a program. Java uses threads to perform multiple tasks concurrently. Each thread in Java is an instance of the **Thread** class or implements the **Runnable** interface.

Creating Threads:

Extending the Thread Class: You can create a new thread by defining a class that extends Thread and overriding its run() method. When you create an instance of this class and call start() on it, the run() method will execute in a new thread.

class MyThread extends Thread {

public void run() {

System.out.println("Thread is running");

}

}

public class Main {

public static void main(String[] args) {

MyThread thread = new MyThread();

thread.start(); // Start the thread

}

}

Implementing the Runnable Interface: Alternatively, you can create a class that implements the Runnable interface and provides an implementation for its run() method. Then you can pass an instance of this class to a Thread object and start the thread.

class MyRunnable implements Runnable {

public void run() {

System.out.println("Thread is running");

}

}

Using Lambda Expressions with Runnable

lambda expressions to create threads in a more concise and readable manner. Lambda expressions are particularly useful when implementing functional interfaces like Runnable, which has a single method run() that you need to override.

Here's how you can create and start a thread using a lambda expression:

public class Main {

public static void main(String[] args) {

// Using a lambda expression to implement the Runnable interface

Runnable task = () -> {

System.out.println("Thread is running with lambda expression");

};

// Create a new Thread with the Runnable task

Thread thread = new Thread(task);

// Start the thread

thread.start();

}

}

Using Callable and Future for Tasks

In Java, the Callable interface and Future Interface provide a way to perform tasks asynchronously and retrieve their results. They are part of the java.util.concurrent package and offer a more flexible alternative to Runnable and Thread for concurrent programming

The Callable interface is similar to Runnable, but it can return a result and throw a checked exception. It has a single method:

public interface Callable<V> {

V call() throws Exception;

}

The Future interface represents the result of an asynchronous computation. It provides methods to check if the computation is complete, retrieve the result, and handle exceptions. Key methods include:

V get() – Retrieves the result of the computation, blocking if necessary until it is available.

V get(long timeout, TimeUnit unit) – Retrieves the result, waiting up to the specified timeout.

boolean isDone() – Checks if the computation is complete.

boolean isCancelled() – Checks if the computation was canceled.

Here's an example of using Callable and Future to execute tasks asynchronously and retrieve their results:

import java.util.concurrent.Callable;

import java.util.concurrent.Future;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.ExecutionException;

public class CallableFutureExample {

public static void main(String[] args) {

// Create an ExecutorService to manage the threads

ExecutorService executor = Executors.newFixedThreadPool(2);

// Define a Callable task

Callable<Integer> task = () -> {

System.out.println("Task is running");

// Simulate a long-running task

Thread.sleep(2000);

return 123;

};

// Submit the task and get a Future object

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

try {

// Retrieve the result of the computation

Integer result = future.get();

System.out.println("Task result: " + result);

} catch (InterruptedException | ExecutionException e) {

e.printStackTrace();

}

// Shut down the ExecutorService

executor.shutdown();

}

}

**Explanation:**

**ExecutorService:**

ExecutorService executor = Executors.newFixedThreadPool(2); creates a thread pool with two threads. ExecutorService provides a higher-level replacement for using Thread directly.

**Callable Task:**

Callable<Integer> task = () -> { ... }; defines a Callable task that returns an Integer result. It simulates a long-running computation by sleeping for 2 seconds.

**Submitting the Task:**

Future<Integer> future = executor.submit(task); submits the Callable task for execution. submit() returns a Future object that can be used to retrieve the result later.

**Getting the Result:**

Integer result = future.get(); retrieves the result of the computation. If the result is not yet available, get() will block until it is. It can also throw InterruptedException or ExecutionException.

**Shutting Down the ExecutorService:**

executor.shutdown(); shuts down the ExecutorService after all tasks have been completed. It does not accept new tasks but allows previously submitted tasks to finish.

Using Callable and Future allows you to handle asynchronous tasks more effectively, especially when you need to get a result or handle exceptions.

Thread Lifecycle:

**New:** A thread that has been created but not yet started is in the new state.

**Runnable**: After the start() method is called, the thread moves to the runnable state, meaning it’s ready to run but may not be currently executing.

**Blocked**: A thread can enter this state if it is waiting for a resource or a lock to be released.

**Timed** **Waiting**: A thread that is waiting for a specified period.

**Terminated**: A thread that has completed its execution or has been terminated is in the terminated state.

**CompletableFuture** is a powerful class in the java.util.concurrent package introduced in Java 8 that provides a more flexible and comprehensive approach to asynchronous programming compared to Future. It is part of the Java Concurrency framework and allows you to write non-blocking, asynchronous code in a more readable and maintainable way.

Key Features of **CompletableFuture**

**Asynchronous Computation:** start computations asynchronously and combine them using a fluent API.

**Non-blocking:** It supports non-blocking operations and allows chaining multiple tasks.

**Exception Handling:** It provides mechanisms to handle exceptions that might occur during asynchronous computations.

**Completion Stages:** It supports creating pipelines of tasks that execute in sequence or in parallel.

Basic Usage of CompletableFuture

Here are some common operations you can perform with CompletableFuture:

1. **Creating a CompletableFuture**

You can create a CompletableFuture either by using its static methods or by constructing it directly.

import java.util.concurrent.CompletableFuture;

public class CompletableFutureExample {

public static void main(String[] args) {

// Using supplyAsync to start a computation asynchronously

CompletableFuture<Integer> future = CompletableFuture.supplyAsync(() -> {

System.out.println("Computing...");

// Simulate some computation

try { Thread.sleep(1000); } catch (InterruptedException e) { Thread.currentThread().interrupt(); }

return 42;

} );

// Get the result

future.thenAccept(result -> System.out.println("Result: " + result));

}

}

**2. Combining Futures**

You can combine multiple futures to create complex pipelines.

import java.util.concurrent.CompletableFuture;

public class CompletableFutureCombineExample {

public static void main(String[] args) {

CompletableFuture<Integer> future1 = CompletableFuture.supplyAsync(() -> {

return 5;

});

CompletableFuture<Integer> future2 = CompletableFuture.supplyAsync(() -> {

return 10;

});

// Combine results from future1 and future2

CompletableFuture<Integer> combinedFuture = future1.thenCombine(future2, (result1, result2) -> {

return result1 + result2;

});

combinedFuture.thenAccept(result -> System.out.println("Combined Result: " + result));

}

}

3. **Handling Exceptions**

CompletableFuture allows you to handle exceptions using methods like exceptionally.

import java.util.concurrent.CompletableFuture;

public class CompletableFutureExceptionHandlingExample {

public static void main(String[] args) {

CompletableFuture<Integer> future = CompletableFuture.supplyAsync(() -> {

throw new RuntimeException("An error occurred");

});

future.exceptionally(ex -> {

System.out.println("Exception: " + ex.getMessage());

return -1; // Provide a default value in case of an exception

}).thenAccept(result -> System.out.println("Result: " + result));

}

}

4. **Waiting for Multiple Futures**

You can wait for all or any of multiple futures to complete.

import java.util.concurrent.CompletableFuture;

import java.util.concurrent.ExecutionException;

public class CompletableFutureAllOfExample {

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

CompletableFuture<Void> future1 = CompletableFuture.runAsync(() -> {

try { Thread.sleep(1000); } catch (InterruptedException e) { Thread.currentThread().interrupt(); }

System.out.println("Task 1 completed");

});

CompletableFuture<Void> future2 = CompletableFuture.runAsync(() -> {

try { Thread.sleep(2000); } catch (InterruptedException e) { Thread.currentThread().interrupt(); }

System.out.println("Task 2 completed");

});

// Wait for all tasks to complete

CompletableFuture.allOf(future1, future2).join();

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

}

}

Summary

supplyAsync(() -> {}) // to submit a task

future.thenAccept(result -> System.out.println("Result: " + result));

multiple futures to create complex pipelines

future1.thenCombine(future2, (result1, result2) -> { return relt1 + ret2;});

allOf(futures …).join()wait for all or any of multiple futures to complete

CompletableFuture provides a powerful and flexible API for working with asynchronous programming in Java. It supports various operations such as combining futures, handling exceptions, and waiting for multiple futures to complete. The fluent API and non-blocking nature make it a valuable tool for building complex asynchronous workflows.

The Java Executor Framework, introduced in Java 5 as part of the java.util.concurrent package, is a robust framework for managing and controlling the execution of asynchronous tasks. It abstracts away low-level thread management, allowing you to focus on task execution rather than thread creation and lifecycle management.

**Core Components of the Executor Framework**

**Executor Interface**

**ExecutorService Interface**

**ScheduledExecutorService Interface**

**ThreadPoolExecutor Class**

**Executors Class**

**1. Executor Interface**

The Executor interface is the simplest and most basic interface in the framework. It defines a single method for executing tasks:

public interface Executor {

void execute(Runnable command);

}

The execute() method accepts a Runnable task and executes it asynchronously.

**2. ExecutorService Interface**

ExecutorService extends Executor and provides more advanced features such as task submission and lifecycle management:

public interface ExecutorService extends Executor {

<T> Future<T> submit(Callable<T> task);

Future<?> submit(Runnable task);

List<Runnable> shutdownNow();

void shutdown();

boolean isShutdown();

boolean isTerminated();

boolean awaitTermination(long timeout, TimeUnit unit) throws InterruptedException;

}

**submit()**: Submits a task for execution and returns a Future object.

**shutdown()**: Initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks are accepted.

**shutdownNow()**: Attempts to stop all actively executing tasks and halts the processing of waiting tasks.

**3. ScheduledExecutorService Interface**

ScheduledExecutorService extends ExecutorService and adds methods for scheduling tasks:

public interface ScheduledExecutorService extends ExecutorService {

ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit unit);

<V> ScheduledFuture<V> schedule(Callable<V> callable, long delay, TimeUnit unit);

ScheduledFuture<?> scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit);

ScheduledFuture<?> scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit);

}

**schedule()**: Schedules a task to execute after a specified delay.

**scheduleAtFixedRate()**: Schedules a task to execute at a fixed rate, with the first execution delayed.

**scheduleWithFixedDelay()**: Schedules a task to execute with a fixed delay between the end of one execution and the start of the next.

**4. ThreadPoolExecutor Class**

ThreadPoolExecutor is a flexible implementation of ExecutorService that provides a pool of threads to execute tasks. It is highly configurable and can be customized using various parameters:

public class ThreadPoolExecutor extends AbstractExecutorService {

public ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue) {

// Constructor implementation

}

}

**corePoolSize**: The number of core threads in the pool.

**maximumPoolSize**: The maximum number of threads allowed in the pool.

**keepAliveTime**: The time that excess idle threads will wait for new tasks before terminating.

**workQueue**: The queue used to hold tasks before they are executed.

**5. Executors Class**

The Executors class provides factory methods for creating various types of thread pools and other executor services:

public class Executors {

public static ExecutorService newFixedThreadPool(int nThreads) {

return new ThreadPoolExecutor(nThreads, nThreads, 0L, TimeUnit.MILLISECONDS, new LinkedBlockingQueue<Runnable>());

}

public static ExecutorService newCachedThreadPool() {

return new ThreadPoolExecutor(0, Integer.MAX\_VALUE, 60L, TimeUnit.SECONDS, new SynchronousQueue<Runnable>());

}

public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize) {

return new ScheduledThreadPoolExecutor(corePoolSize);

}

public static ExecutorService newSingleThreadExecutor() {

return new FinalizableDelegatedExecutorService(new ThreadPoolExecutor(1, 1, 0L, TimeUnit.MILLISECONDS, new LinkedBlockingQueue<Runnable>()));

}

}

**Example Usage**

Here is a simple example demonstrating the use of the ExecutorService and ScheduledExecutorService:

import java.util.concurrent.\*;

public class ExecutorFrameworkExample {

public static void main(String[] args) {

// Creating a fixed thread pool

ExecutorService executor = Executors.newFixedThreadPool(2);

// Submitting tasks

Future<Integer> future1 = executor.submit(() -> {

Thread.sleep(1000);

return 1;

});

Future<Integer> future2 = executor.submit(() -> {

Thread.sleep(2000);

return 2;

});

try {

// Retrieving results

System.out.println("Result 1: " + future1.get());

System.out.println("Result 2: " + future2.get());

} catch (InterruptedException | ExecutionException e) {

e.printStackTrace();

}

// Shutting down the executor

executor.shutdown();

// Creating a scheduled executor service

ScheduledExecutorService scheduledExecutor = Executors.newScheduledThreadPool(1);

// Scheduling tasks

scheduledExecutor.scheduleAtFixedRate(() -> System.out.println("Scheduled Task"), 0, 1, TimeUnit.SECONDS);

// Shutting down the scheduled executor service

// Note: In real applications, use shutdown appropriately

// scheduledExecutor.shutdown();

}

}

**Summary**

The Executor Framework in Java simplifies the management of asynchronous tasks by providing a high-level API for thread management and task execution. It includes various interfaces and classes for different use cases, such as managing thread pools, scheduling tasks, and handling task results and exceptions. By leveraging the Executor Framework, you can write more efficient and maintainable concurrent code.

Thread Management:

Synchronization: Java provides mechanisms such as synchronized blocks, methods, and high-level concurrency utilities (e.g., ReentrantLock, Semaphore) to manage access to shared resources and avoid issues like race conditions.

Concurrency Utilities: Java's java.util.concurrent package provides classes like ExecutorService, CountDownLatch, and CyclicBarrier to simplify thread management and coordination.

Thread Priorities and Daemon Threads:

Thread Priorities: Threads can have priorities that affect their scheduling. You can set a thread's priority using setPriority() and get it with getPriority(). Priorities range from Thread.MIN\_PRIORITY (1) to Thread.MAX\_PRIORITY (10), with the default being Thread.NORM\_PRIORITY (5).

Daemon Threads: Daemon threads are background threads that do not prevent the JVM from exiting when all user threads have finished. You can make a thread a daemon by calling setDaemon(true) before starting it.

Threads are a fundamental part of Java's concurrency model, enabling efficient multitasking and parallel processing in applications.

What is the difference between start() and run() methods in threads?

The start() method is used to create a new thread and the run() method contains the code that is executed in the thread. If you call run() directly, the code will run in the current thread and not in a new thread.

What is the use of join() in threads?

The join() method waits for a thread to die. In other words, it causes the currently running threads to stop executing until the thread it joins with completes its task.

What is synchronization in the context of multithreading?

Synchronization is the capability to control the access of multiple threads to shared resources. Without synchronization, it is possible for one thread to modify a shared object while another thread is in the process of using or updating that object's value, leading to significant errors.

What is a deadlock?

A deadlock is a situation where two or more threads are blocked forever, waiting for each other. This usually happens when multiple threads need the same locks but obtain them in different order.

What are the methods used for inter-thread communication?

wait(), notify(), and notifyAll() methods are used for inter-thread communication in Java.

What is the difference between a process and a thread?

A process is a self-contained execution environment and it can contain multiple threads. A thread is a subset of the process.

What is the purpose of Thread.sleep() in Java?

Thread.sleep() method causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.

Remember, understanding multithreading can help you write more efficient and effective Java program Sure, here are some frequently asked questions (FAQs) about multithreading in Java:

Frequently asked questions (FAQs) about the **Executor** **Framework in Java**:

What is the Executor Framework in Java?

Is a framework provided by Java that allows you to create and manage threads. It provides a pool of threads and an API for assigning tasks to them.

The key components are the *Executor* interface, the *ExecutorService* interface (which extends Executor), the *ScheduledExecutorService* interface (which extends *ExecutorService*), and the *Executors* utility class.

What is the difference between Executor and ExecutorService?

The Executor interface provides a single method, execute(Runnable), for launching new tasks. ExecutorService is a more complete interface that provides additional methods for lifecycle management of both the tasks and the executor itself.

What is a ThreadPoolExecutor?

ThreadPoolExecutor is an implementation of the ExecutorService interface. It executes each submitted task using one of possibly several pooled threads.

What is a ScheduledExecutorService?

ScheduledExecutorService is an ExecutorService that can schedule commands to run after a given delay, or to execute periodically.

What is the Executors class?

Executors is a utility class that provides factory methods for creating different kinds of executor services, like newSingleThreadExecutor(), newFixedThreadPool(int), newCachedThreadPool(), and newScheduledThreadPool(int).

How can we shut down an ExecutorService?

We can shut down an ExecutorService by calling its shutdown() or shutdownNow() methods. The shutdown() method will allow previously submitted tasks to execute before terminating, while the shutdownNow() method attempts to stop all actively executing tasks and returns a list of the tasks that were awaiting execution.

Remember, using the Executor Framework can help you manage threads more effectively and write more efficient multithreaded code.

What is CompletableFuture?

CompletableFuture is a Future that may be explicitly completed (setting its value and status), and may be used as a CompletionStage, supporting dependent functions and actions that trigger upon its completion.

How is CompletableFuture different from Future?

CompletableFuture implements the Future interface but also includes a number of additional methods for chaining and combining tasks, handling exceptions, and manually completing the Future.

How do I use CompletableFuture?

You can create a CompletableFuture with CompletableFuture.supplyAsync(() -> method()), where method() is the method you want to run asynchronously. You can then chain methods like thenApply(), thenAccept(), or thenRun() to execute additional tasks after the initial task completes.

What is the difference between thenApply and thenAccept in CompletableFuture?

thenApply() is used when you need to return a value after processing the previous stage. thenAccept() is used when you don't need to return a value and just want to consume the result from the previous stage.

These features help to write more efficient and cleaner code in Java

How can I handle exceptions in CompletableFuture?

You can handle exceptions using the exceptionally() method, which lets you define a function to return a default value when an exception occurs. Alternatively, you can use handle(), which lets you handle both the result and the exception, if one occurs.

How can I combine multiple CompletableFutures?

You can use methods like thenCompose() to chain multiple CompletableFutures, or allOf() to create a new CompletableFuture that is completed when all of the given CompletableFutures complete.

How can I make a CompletableFuture run synchronously?

You can use the join() method to block the current thread until the CompletableFuture completes. However, this defeats the purpose of using CompletableFuture for asynchronous processing, so use it sparingly.

Remember, the CompletableFuture API is a powerful tool for asynchronous programming in Java, but it also comes with complexity. Make sure to handle exceptions properly and be aware of potential pitfalls, like blocking the current thread with join().

**4. Thread Synchronization**

Synchronized Methods and Blocks

volatile Keyword

Thread Safety and Concurrency Control

**6. Thread Communication**

wait(), notify(), and notifyAll()

Using Monitors and Locks

Condition Interface

**7. Advanced Synchronization**

Explicit Locks (ReentrantLock, ReadWriteLock)

Semaphores

CountDownLatch

CyclicBarrier

Phaser

**8. Thread Coordination**

Barriers and Latches

Synchronizers and Utility Classes

Managing Thread Lifecycles

**9. Concurrent Collections**

ConcurrentHashMap

CopyOnWriteArrayList

BlockingQueue (e.g., ArrayBlockingQueue, LinkedBlockingQueue)

ConcurrentLinkedQueue

**10. Thread Safety Patterns**

Immutable Objects

Thread-Local Variables

Atomic Variables (AtomicInteger, AtomicReference, etc.)

Singleton Design Pattern with Lazy Initialization

**11. Handling Concurrency Issues**

Deadlock

Starvation

Livelock

Race Conditions

**12. Fork/Join Framework**

Introduction to Fork/Join

RecursiveTask and RecursiveAction

Fork/Join Pool and Work Stealing Algorithm

**13. Performance and Best Practices**

Optimizing Thread Performance

Avoiding Excessive Synchronization

Profiling and Debugging Multithreaded Code

**14. Java Memory Model (JMM)**

Visibility of Variables

Happens-Before Relationship

Thread Visibility and Caching

**15. Thread Pools and Executors**

Fixed Thread Pools

Cached Thread Pools

Single Thread Executor

Scheduled Executor Service

**16. Thread-Safe Design**

Designing Thread-Safe Classes

Using synchronized Blocks Efficiently

Best Practices for Concurrency Design

**17. Asynchronous Programming**

Future and CompletableFuture API

Reactive Programming and Libraries (e.g., RxJava, Project Reactor)

Non-Blocking I/O with NIO and AIO

**18. Testing Multithreaded Code**

Techniques for Testing Concurrency

Tools for Concurrency Testing

JUnit and Multithreading

Each CPU core will have core cached share and cached share below it.

All the changes to instance variable will be done in core cached share and not in cached share.

In case of multi threading it will not get the latest change to that variable in the changed by the thread run on the other core.

We need somebody to flush the changes to shared cache below.. This is done using the volatile keyword

And if it is compounded operation it should use synchronized block

synchronize(object) {

}

Lock lock = new ReentrantLock()

Condition condition = lock.newCondition()

lock.lock()

// code

lock.unlock()

Multiple threads wait on the same condition

condition.await() <- similar to wait() method suspend the thread

condition.signal() <-- similar to notify this with the awake

condition.signalAll() <-- notifyall

Perform await in loop always to fix the spurious wake up

Indicate completion of a condition.

Co

//instance of executer service

ExecuterService service = Executors.newFixedThreadPool(10)

// create a thread pool

service.

FixedNothread poolnumber will be based on no of cores present

or thread will be some good number based on current criteria if they ware IO intensive operation

ForkJoinPool - task producing subtask

blocking and non bloking

Future<Integer> future = service.submit(new Task());

Here the task should implement Callable interface not the runnable

Callable will return something for their call() method

Integer integer = future.get() // this is blocking operation

This approach has issues to when the depending

CompletableFuture - perform posible asynchronous (non-blocking) computation and trigger depenedant computation which could also be asynchronous.

CompletableFuture.supplyAsync(() -> getOrder)

.thenApply(objec -> method())

.thenApplyAsyn(objec -> method())

.thenAccept(0-> sendmail())

concurency vs parallaisam

"parallasim is doing lot of things at once" so that we can speed up our programs

Java enable parallelism using

Raw Threads- > using Thread and Runnable interfaces

ThreadPool

ExecutorService

ForkJoinPool

Custom ThreadPools (eg: webservers)

-- Requires more then one CPU core

Concurrency -- using shared resources - "dealing lot of things at once"

to fix the issues with shared resources

Tools to deals with concurency

- Locks/Syncronization

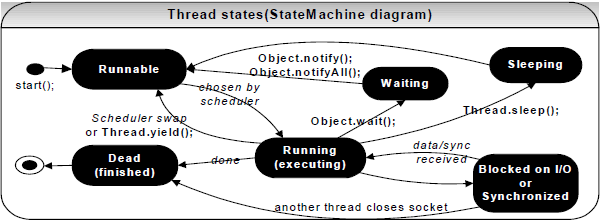
- Atomic classes

- ConcurrentData structure(eg : ConcurrentHashMap, BlockingQueue)

- CompletableFuture

- CountdownLatch/Phasers/Cyclic Barier/Semaphore etc

Briefly explain high-level thread states?



**Runnable** — waiting for its turn to be picked for execution by the thread schedular based on thread priorities.

**Running**: The processor is actively executing the thread code. It runs until it becomes blocked, or voluntarily gives up its turn with this static method *Thread.yield().* Because of context switching overhead, *yield()* should not be used very frequently.

**Waiting**: A thread is in a **blocked state** while it waits for some external processing such as file I/O to finish.

**Sleeping**: Java threads are forcibly put to sleep (suspended) with this overloaded method:

Thread.sleep(milliseconds), Thread.sleep(milliseconds, nanoseconds);

**Blocked on I/O**: Will move to runnable after I/O condition like reading bytes of data etc changes.

**Blocked on synchronization**: Will move to Runnable when a **lock is acquired.**

**Dead**: The thread is finished working.

What is the difference between yield and sleeping?

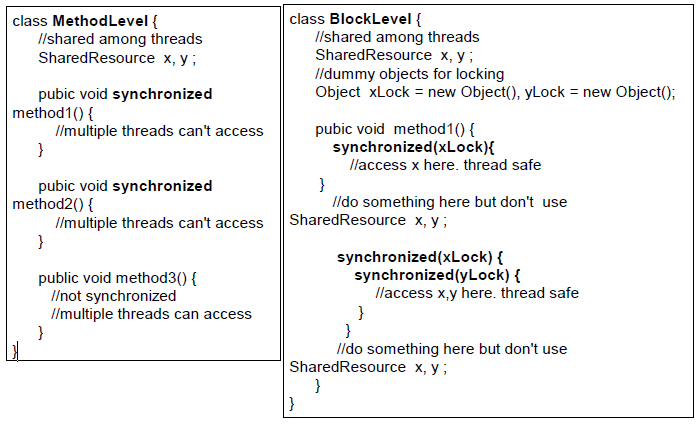
When a task invokes yield(), it changes from running state to runnable state. When a task invokes sleep(), it changes from running state to waiting/sleeping state.

How does thread synchronization occurs inside a monitor? What levels of synchronization can you apply? What is the difference between synchronized method and synchronized block?

In Java programming, each object has a lock. A thread can acquire the lock for an object by using the

***synchronized***keyword. The synchronized keyword can be applied in **method level** (coarse grained lock – can affect performance adversely) or **block level of code** (fine grained lock). Often using a lock on a method level is too coarse. Why lock up a piece of code that does not access any shared resources by locking up an entire method. Since each object has a lock, dummy objects can be created to implement block level synchronization.

The block level is more efficient because it does not lock the whole method.



The JVM uses locks in conjunction with monitors. A monitor is basically a guardian who watches over a sequence of synchronized code and making sure only one thread at a time executes a synchronized piece of code. Each monitor is associated with an object reference. When a thread arrives at the first instruction in a block of code it must obtain a lock on the referenced object. The thread is not allowed to execute the code until it obtains the lock.

Once it has obtained the lock, the thread enters the block of protected code. When the thread leaves the block, no matter how it leaves the block, it releases the lock on the associated object.

***Why synchronization is important?*** Without synchronization, it is possible for one thread to modify a shared object while another thread is in the process of using or updating that object’s value. This often causes dirty data and leads to significant errors. **The disadvantage of synchronization** is that it can cause deadlocks when two threads are waiting on each other to do something. Also synchronized code has the overhead of acquiring lock, which can adversely the performance.

What is a daemon thread?

Daemon threads are sometimes called "service" threads. These are threads that normally run at a low priority and provide a basic service to a program or programs when activity on a machine is reduced. An example of a daemon thread that is continuously running is the garbage collector thread. This thread is provided by the JVM.

How can threads communicate with each other? How would you implement a producer (one thread) and a consumer (another thread) passing data (via stack)?

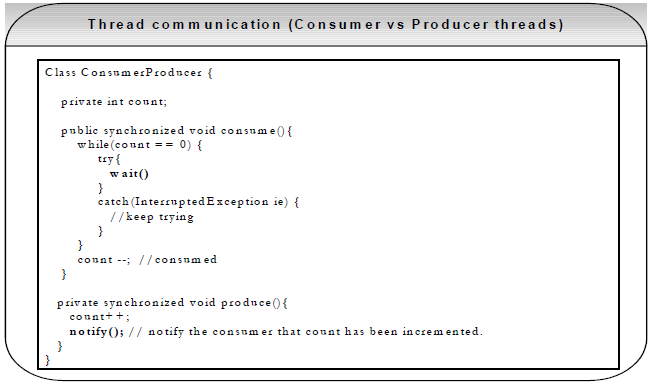
The **wait(), notify()**, and **notifyAll()** methods are used to provide an efficient way for threads to communicate with each other. This communication solves the ‘**consumer-producer problem**’. This problem occurs when the producer thread is completing work that the other thread (consumer thread) will use.

**Example**: If you imagine an application in which one thread (the producer) writes data to a file while a second

thread (the consumer) reads data from the same file. In this example the concurrent threads share the same resource file. Because these threads share the common resource file they should be synchronized. Also these two threads should communicate with each other because the consumer thread, which reads the file, should wait until the producer thread, which writes data to the file and notifies the consumer thread that it has completed its writing operation.

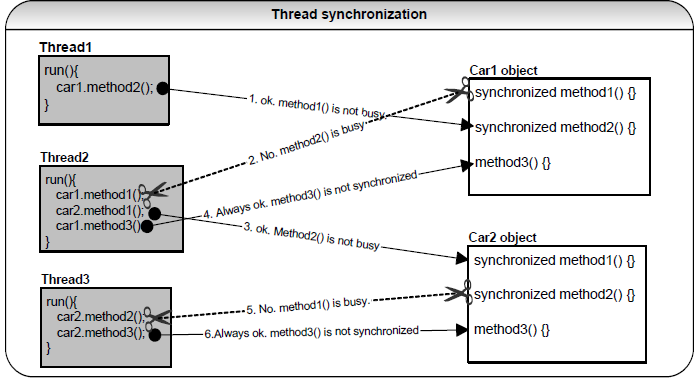
Let’s look at a sample code where **count** is a shared resource. The consumer thread will wait inside the

consume() method on the producer thread, until the producer thread increments the count inside the produce() method and subsequently notifies the consumer thread. Once it has been notified, the consumer thread waiting inside the consume() method will give up its waiting state and completes its method by consuming the count (i.e. decrementing the count).



If 2 different threads hit 2 different synchronized methods in an object at the same time will they both continue?

No. Only one method can acquire the lock.



Explain threads blocking on I/O?

Occasionally threads have to block on conditions other than object locks. I/O is the best example of this. Threads block on I/O (i.e. enters the waiting state) so that other threads may execute while the I/O operation is performed.

When threads are blocked (say due to time consuming reads or writes) on an I/O call inside an object’s

synchronized method and also if the other methods of the object are also synchronized then the object is essentially frozen while the thread is blocked.

**Be sure to not synchronize code that makes blocking calls**, or make sure that a non-synchronized method exists on an object with synchronized blocking code. Although this technique requires some care to ensure that the resulting code is still thread safe, it allows objects to be responsive to other threads when a thread holding its locks is blocked.

**Note:** The **java.nio.\*** package was introduced in JDK1.4. The coolest addition is nonblocking I/O (aka NIO that

stands for New I/O). Refer **Q20** in Java section for NIO.

Sure, here are some frequently asked questions (FAQs) about multithreading in Java:

What is multithreading in Java?

Multithreading in Java is a feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU.

How to create a thread in Java?

There are two ways to create a thread in Java: by extending the Thread class and by implementing the Runnable interface.