Multithreading in Java

Kindly explain below points with respect to multithreading in Java:Creating Threads
 - Extending the Thread Class
 - Implementing the Runnable Interface
 - Thread Lifecycle

Multithreading in Java

Multithreading allows a program to execute multiple threads concurrently, which is beneficial for performing tasks in parallel and improving application performance. Here's an explanation of the requested points:

Creating Threads

In Java, you can create threads in two primary ways:

1. Extending the `Thread` Class

- The `Thread` class represents a thread in Java.
- You can create a thread by subclassing the `Thread` class and overriding its `run()` method.

Steps:

1. Create a class that extends the `Thread` class.
2. Override the `run()` method to define the code that should execute in the new thread.
3. Create an instance of your class and call the `start()` method to begin execution.

Example:

```java
class MyThread extends Thread {
    @Override
    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
    }
}
```

Notes:

- Subclassing `Thread` is straightforward but not ideal if your class also needs to extend another class, as Java does not support multiple inheritance.

---

2. Implementing the `Runnable` Interface

- This is the preferred method because it allows your class to inherit from other classes while still representing a thread.

Steps:

1. Create a class that implements the `Runnable` interface.
2. Implement the `run()` method.
3. Pass an instance of your class to a `Thread` object and call its `start()` method.

Example:

```java
class MyRunnable implements Runnable {
    @Override
    public void run() {
        System.out.println("Runnable thread is running...");
    }
```

```
}

public class Main {
    public static void main(String[] args) {
        MyRunnable myRunnable = new MyRunnable();
        Thread thread = new Thread(myRunnable);
        thread.start(); // Start the thread
    }
}
```

Notes:

- This approach is more flexible because the thread logic is decoupled from the thread management.

Thread Lifecycle

A thread in Java goes through the following states during its lifecycle:

1. **New (Created)**:
    - A thread object is created using the `Thread` class but has not yet started.
    - Example: `Thread t = new Thread();`
2. **Runnable (Ready to run)**:
    - The thread is ready to run and waiting for CPU time.
    - This state is achieved after calling the `start()` method.
    - Example: `t.start();`
3. **Running**:
    - The thread is actively executing its task in the `run()` method.
    - Only one thread can be in this state at a time per CPU core.
4. **Blocked/Waiting**:
    - The thread is waiting for a monitor lock (e.g., due to synchronized block) or is waiting for a signal.
    - Example: `wait()`, `sleep()`, or `join()` methods.
5. **Terminated (Dead)**:
    - The thread finishes its execution or is stopped explicitly. Once terminated, a thread cannot be restarted.
    - Example: After the `run()` method completes.

Thread Lifecycle Diagram

```plaintext
New --> Runnable --> Running --> Terminated
           ^    |
           |    v
        Blocked/Waiting
```

Key Methods Influencing Thread States

- `start()`: Moves thread from New to Runnable.
- `yield()`: Suggests that the current thread give up CPU to other threads.
- `sleep(milliseconds)`: Pauses the thread for a specified time.
- `join()`: Makes the current thread wait for another thread to complete.
- `wait()` / `notify()`: Used for thread synchronization.

By understanding these concepts, you can create, control, and synchronize threads effectively in Java.

How to create threads through lambda expression

Creating threads through lambda expressions in Java is a modern and concise way to implement multithreading. Lambda expressions are particularly useful when you need to create lightweight threads without explicitly creating separate classes or implementing the `Runnable` interface in a traditional way.

Here's how you can use lambda expressions to create threads:

Steps to Create Threads Using Lambda Expressions

1. **Create a `Thread` instance.**

2. Pass a lambda expression to the `Thread` constructor. The lambda expression serves as the implementation of the `Runnable` interface's `run()` method.
3. Call the `start()` method on the `Thread` instance to begin execution.

Example: Using Lambda Expression

```java
public class Main {
    public static void main(String[] args) {
        // Creating a thread using a lambda expression
        Thread thread = new Thread(() -> {
            System.out.println("Thread is running using lambda...");
        });

        // Start the thread
        thread.start();
    }
}
```

Breaking Down the Code

- The `Thread` constructor accepts a `Runnable` as an argument.
- Instead of creating a separate class or object for `Runnable`, you can use a lambda expression: `() -> { /* code */ }`.
- The `run()` method is implicitly implemented inside the lambda.

Example with Multiple Threads

```java
public class Main {
    public static void main(String[] args) {
        // Creating multiple threads using lambda expressions
        Thread thread1 = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                System.out.println("Thread 1 - Count: " + i);
```

```
            }
        });

        Thread thread2 = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                System.out.println("Thread 2 - Count: " + i);
            }
        });

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

Advantages of Using Lambda for Threads

1. **Conciseness**: Eliminates the need for boilerplate code like creating separate classes for `` `Runnable` ``.
2. **Improved Readability**: Code becomes cleaner and easier to understand.
3. **Flexibility**: Allows you to define thread behavior inline, making it easier to customize tasks dynamically.

Example with Thread Pool

You can also use lambda expressions in thread pools for concurrent tasks:

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

public class Main {
    public static void main(String[] args) {
        // Create a thread pool with 3 threads
        ExecutorService executor = Executors.newFixedThreadPool(3);

        // Submit tasks using lambda expressions
        for (int i = 1; i <= 5; i++) {
            int taskNumber = i;
            executor.submit(() -> {
                System.out.println("Task " + taskNumber + " is running...");
            });
```

```
        }

        // Shut down the executor
        executor.shutdown();
    }
}
```

Key Points

- Lambda expressions in Java require Java 8 or later.
- Lambda expressions simplify multithreading code but should still follow good practices (e.g., proper synchronization for shared resources).
- While using lambdas, ensure the task logic is concise and does not require a complex structure, as readability may suffer otherwise.

Kindly explain the below points in details: - Synchronization & Thread Safety
- Synchronized Methods
- Synchronized Blocks
- Volatile Keyword

Synchronization & Thread Safety

In a multithreaded environment, **thread safety** ensures that shared resources are accessed and updated by multiple threads in a consistent and predictable manner. **Synchronization** is a mechanism to control the access of multiple threads to shared resources to prevent race conditions and ensure thread safety.

---

1. Synchronized Methods

Synchronized methods ensure that only one thread can execute the method at a time for a given object. When a thread calls a synchronized method, it acquires a lock (or monitor) on the object. Other threads trying to execute any synchronized method on the same object must wait until the lock is released.

Example:

```java
class Counter {
    private int count = 0;

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

    public synchronized int getCount() {
        return count;
    }
}

public class Main {
    public static void main(String[] args) {
        Counter counter = new Counter();

        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        });

        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        });

        t1.start();
        t2.start();

        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Final Count: " + counter.getCount());
    }
}
```

Key Points:

- The `synchronized` keyword ensures that only one thread can access the `increment()` method at a time.

- The lock is associated with the object (`this`), so synchronization applies to methods invoked on the same object.

---

2. Synchronized Blocks

Synchronized blocks allow more granular control over the synchronization process by locking only specific critical sections of code instead of the entire method. This can improve performance by minimizing the amount of synchronized code.

Syntax:

```java
synchronized (lockObject) {
    // Critical section
}
```

Example:

```java
class Counter {
    private int count = 0;

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

    public int getCount() {
        synchronized (this) {
            return count;
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Counter counter = new Counter();

        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
```

```
                counter.increment();
            }
        });

        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        });

        t1.start();
        t2.start();

        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Final Count: " + counter.getCount());
    }
}
```

Key Points:

- You can lock any object, not just `` `this` ``. For example, you could use a custom lock object: `` `synchronized (lock)` ``.
- Synchronizing only critical sections can lead to better performance.

---

3. Volatile Keyword

The `` `volatile` `` keyword ensures that changes to a variable are always visible to all threads. It prevents threads from caching the value of a variable and forces them to read it directly from main memory.

Usage:

- It is used for variables that are shared between threads where the value might be updated by one thread and read by others.
- It does **not** provide atomicity for operations. For example, incrementing a `` `volatile` `` variable is not thread-safe.

Example:

```java
class Flag {
    private volatile boolean running = true;

    public void stop() {
        running = false;
    }

    public boolean isRunning() {
        return running;
    }
}

public class Main {
    public static void main(String[] args) {
        Flag flag = new Flag();

        Thread t = new Thread(() -> {
            while (flag.isRunning()) {
                System.out.println("Thread is running...");
                try {
                    Thread.sleep(500);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            System.out.println("Thread stopped.");
        });

        t.start();

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

        flag.stop(); // Stop the thread
    }
}
```

Key Points:

- Without `volatile`, the `running` variable might be cached by the thread, causing it to ignore updates made by other threads.
- Use `volatile` when you need visibility guarantees but not atomicity.

Comparison: Synchronized vs. Volatile

Aspect	Synchronized	Volatile
Purpose	Ensures both visibility and atomicity.	Ensures only visibility.
Scope	Can be applied to methods or blocks.	Can be applied to variables only.
Locking	Uses locks to synchronize access.	Does not use locks; lightweight.
Performance	May impact performance due to locking.	Faster due to lack of locking.
Use Case	When atomicity and visibility are required.	When only visibility is required.

Key Takeaways

1. Use **synchronized methods** when the entire method needs to be thread-safe.
2. Use **synchronized blocks** when only part of a method needs synchronization.
3. Use the **volatile keyword** when you need a variable to be visible across threads without needing atomicity.
4. For complex thread-safety requirements, consider using higher-level concurrency tools like `` `ReentrantLock` ``, `` `AtomicInteger` ``, or the `` `java.util.concurrent` `` package.

Explain the below points in details along with example and concise explanatipn:Inter Thread Communication and Synchronization
- Inter-Thread Communication
- wait(), notify(), and notifyAll() methods
- Producer-Consumer Problem
- Thread Joining

Inter-Thread Communication and Synchronization

Inter-thread communication is a mechanism that allows threads to communicate with each other to coordinate their execution and share resources safely. Synchronization ensures thread safety when multiple threads access shared resources.

1. Inter-Thread Communication

Java provides a way for threads to communicate and cooperate via `` `wait()` ``, `` `notify()` ``, and `` `notifyAll()` `` methods. These methods belong to the `` `Object` `` class because every object in Java can act as a monitor (lock) for thread synchronization.

Key Features:

- A thread can pause its execution and release the lock using `` `wait()` ``.
- Another thread can notify the waiting thread(s) using `` `notify()` `` or `` `notifyAll()` ``.

---

2. `` `wait()` ``, `` `notify()` ``, and `` `notifyAll()` `` Methods

`` `wait()` ``:

- Causes the current thread to wait until another thread invokes `` `notify()` `` or `` `notifyAll()` `` on the same object.
- Releases the lock on the object, allowing other threads to enter synchronized blocks.

`` `notify()` ``:

- Wakes up one thread waiting on the object's monitor.
- The thread resumes execution after reacquiring the lock.

`` `notifyAll()` ``:

- Wakes up all threads waiting on the object's monitor.
- Only one thread acquires the lock and proceeds, while others go back to waiting.

Example:

```java
class SharedResource {
    private int data;
    private boolean available = false;

    public synchronized void produce(int value) {
```

```
        while (available) {
            try {
                wait(); // Wait until the resource is consumed
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        data = value;
        available = true;
        System.out.println("Produced: " + value);
        notify(); // Notify the consumer
    }

    public synchronized void consume() {
        while (!available) {
            try {
                wait(); // Wait until the resource is produced
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("Consumed: " + data);
        available = false;
        notify(); // Notify the producer
    }
}

public class Main {
    public static void main(String[] args) {
        SharedResource resource = new SharedResource();

        Thread producer = new Thread(() -> {
            for (int i = 1; i <= 5; i++) {
                resource.produce(i);
            }
        });

        Thread consumer = new Thread(() -> {
            for (int i = 1; i <= 5; i++) {
                resource.consume();
            }
        });

        producer.start();
        consumer.start();
    }
}
```

Output:

```makefile
Produced: 1
Consumed: 1
Produced: 2
Consumed: 2
...
```

---

3. Producer-Consumer Problem

The **Producer-Consumer Problem** is a classic example of inter-thread communication where:

- **Producer** generates data and shares it with the **Consumer**.
- Synchronization ensures that the producer does not overwrite data before it is consumed, and the consumer does not read data before it is produced.

Solution:

The `SharedResource` example above demonstrates how the producer and consumer coordinate using `wait()` and `notify()`.

---

4. Thread Joining

The `join()` method allows one thread to wait for another thread to finish its execution. It is useful for ensuring that one thread completes its task before another starts.

Syntax:

```java
thread.join();        // Waits indefinitely for the thread to finish
thread.join(timeout); // Waits for the thread to finish or until the timeout expires
```

Example:

```java
public class Main {
    public static void main(String[] args) {
        Thread t1 = new Thread(() -> {
            System.out.println("Thread 1 is running...");
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("Thread 1 finished.");
        });

        Thread t2 = new Thread(() -> {
            System.out.println("Thread 2 is running...");
        });

        t1.start();
        try {
            t1.join(); // Main thread waits for t1 to complete
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        t2.start();
    }
}
```

Output:

```mathematica
Thread 1 is running...
Thread 1 finished.
Thread 2 is running...
```

Key Points:

- The main thread waits for `t1` to finish before starting `t2`.

Summary Table

Feature	Description
Inter-Thread Communication	Mechanism for threads to cooperate and share resources safely.
`wait()`	Pauses the current thread and releases the lock until notified.
`notify()`	Wakes up one waiting thread on the object's monitor.
`notifyAll()`	Wakes up all threads waiting on the object's monitor.
Producer-Consumer Problem	Demonstrates how producers and consumers coordinate using **`wait()`** and **`notify()`**.
Thread Joining	Makes one thread wait for another to complete before continuing execution.

This concise explanation with examples demonstrates how Java threads communicate and synchronize effectively.

Kindly give me the example of Producer-Consumer Problem in details along with expalnation and output

Producer-Consumer Problem Example

The **Producer-Consumer Problem** involves two types of threads:

1. **Producer**: Generates data and puts it into a shared buffer (or resource).
2. **Consumer**: Retrieves data from the shared buffer for processing.

The problem requires:

- Proper synchronization to ensure that:
  - The producer doesn't produce data when the buffer is full.
  - The consumer doesn't consume data when the buffer is empty.
- Use of **`wait()`** and **`notify()`** for communication between threads.

Detailed Implementation

Here, we'll implement a producer-consumer problem with a shared buffer that holds a single item.

Code:

```java
class SharedBuffer {
    private int data;
    private boolean isAvailable = false; // Flag to track if data is available

    // Producer method
    public synchronized void produce(int value) {
        // Wait if the buffer already contains data
        while (isAvailable) {
            try {
                wait(); // Release the lock and wait
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        // Produce data
        data = value;
        isAvailable = true;
        System.out.println("Produced: " + data);
        notify(); // Notify the consumer thread
    }

    // Consumer method
    public synchronized void consume() {
        // Wait if the buffer is empty
        while (!isAvailable) {
            try {
                wait(); // Release the lock and wait
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        // Consume data
        System.out.println("Consumed: " + data);
        isAvailable = false;
        notify(); // Notify the producer thread
    }
}

public class ProducerConsumerExample {
    public static void main(String[] args) {
        SharedBuffer buffer = new SharedBuffer();
```

```
        // Producer thread
        Thread producer = new Thread(() -> {
            for (int i = 1; i <= 5; i++) {
                buffer.produce(i);
                try {
                    Thread.sleep(500); // Simulate delay
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });

        // Consumer thread
        Thread consumer = new Thread(() -> {
            for (int i = 1; i <= 5; i++) {
                buffer.consume();
                try {
                    Thread.sleep(1000); // Simulate delay
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });

        producer.start();
        consumer.start();
    }
}
```

Explanation

1. **SharedBuffer Class**:
    - Contains a shared resource (`data`) and a flag (`isAvailable`) to indicate if the buffer is empty or full.
    - `produce(int value)`:
        - If the buffer already contains data (`isAvailable = true`), the producer thread waits using `wait()`.
        - Once the buffer is empty, the producer generates data, marks it as available, and notifies the consumer.
    - `consume()`:
        - If the buffer is empty (`isAvailable = false`), the consumer thread waits using `wait()`.
        - Once data is available, the consumer consumes it, marks the buffer as empty, and notifies the producer.
2. **Producer Thread**:
    - Generates data in a loop, calling `produce()` on the shared buffer.

- Simulates a delay using `` `Thread.sleep(500)` `` to mimic real-world production latency.

3. **Consumer Thread**:
    - Consumes data in a loop, calling `` `consume()` `` on the shared buffer.
    - Simulates a delay using `` `Thread.sleep(1000)` `` to mimic real-world consumption latency.

Output

```makefile
Produced: 1
Consumed: 1
Produced: 2
Consumed: 2
Produced: 3
Consumed: 3
Produced: 4
Consumed: 4
Produced: 5
Consumed: 5
```

How It Works

1. Initially, the producer generates the first item (`` `Produced: 1` ``).
2. The consumer consumes the item (`` `Consumed: 1` ``), notifying the producer to produce the next item.
3. This process continues:
    - The producer waits if the buffer is full.
    - The consumer waits if the buffer is empty.
4. Both threads alternate smoothly without conflict due to proper synchronization and inter-thread communication.

Key Concepts Demonstrated

1. **Synchronization**: Ensures only one thread accesses the shared resource at a time.
2. **Inter-Thread Communication**: `` `wait()` `` and `` `notify()` `` are used for communication between producer and consumer.
3. **Thread Cooperation**: Threads coordinate to ensure smooth execution and prevent deadlock.

This is a complete and classic example of how multithreading and synchronization work together to solve real-world problems.

Explain the concept of Thread Joining with example

Concept of Thread Joining

The `join()` method in Java is used to make one thread wait for another thread to finish its execution. This is particularly useful when you want to ensure that a thread has completed its task before allowing other threads or the main thread to proceed.

How `join()` Works

1. **Calling `join()`**: When a thread calls `join()` on another thread, it pauses execution of the calling thread until the target thread has completed execution.
2. **Overloaded Methods**:
   - `join()`: Waits indefinitely for the thread to finish.
   - `join(long millis)`: Waits for the thread to finish, but not longer than the specified time in milliseconds.
   - `join(long millis, int nanos)`: Provides finer control with nanosecond precision.

Example: Thread Joining

Code Example

```java
public class ThreadJoinExample {
    public static void main(String[] args) {
        // Thread 1
        Thread t1 = new Thread(() -> {
            System.out.println("Thread 1 is starting...");
            try {
                Thread.sleep(2000); // Simulating work
```

```
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("Thread 1 has finished.");
        });

        // Thread 2
        Thread t2 = new Thread(() -> {
            System.out.println("Thread 2 is starting...");
            try {
                Thread.sleep(1000); // Simulating work
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("Thread 2 has finished.");
        });

        // Start threads
        t1.start();
        t2.start();

        // Main thread waits for t1 and t2 to finish
        try {
            t1.join();
            System.out.println("Main thread waited for Thread 1 to finish.");
            t2.join();
            System.out.println("Main thread waited for Thread 2 to finish.");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("All threads have completed. Main thread is ending.");
    }
}
```

Explanation

1. **Thread Execution Order**:
    - `t1` and `t2` start executing concurrently.
    - `t1` simulates 2 seconds of work, while `t2` simulates 1 second.
2. **`join()` Behavior**:
    - The main thread waits for `t1` to complete using `t1.join()`.
    - After `t1` completes, the main thread waits for `t2` using `t2.join()`.

3. **Synchronized Completion**:
    - The main thread ensures both `t1` and `t2` finish their tasks before continuing.

Output

```mathematica
Thread 1 is starting...
Thread 2 is starting...
Thread 2 has finished.
Thread 1 has finished.
Main thread waited for Thread 1 to finish.
Main thread waited for Thread 2 to finish.
All threads have completed. Main thread is ending.
```

Key Points

1. **Thread Coordination**:
    - `join()` is used to ensure proper thread coordination, especially when the main thread depends on the result of other threads.
2. **Impact on Main Thread**:
    - Without `join()`, the main thread might finish before `t1` or `t2` completes, leading to unexpected behavior.
3. **Timeout with `join(long millis)`**:
    - You can specify a timeout to avoid indefinite waiting if a thread takes too long.

Example with Timeout

```java
public class ThreadJoinWithTimeout {
    public static void main(String[] args) {
        Thread t = new Thread(() -> {
            System.out.println("Thread is running...");
            try {
                Thread.sleep(3000); // Simulate work
```

```
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("Thread finished.");
        });

        t.start();

        try {
            t.join(1000); // Wait only for 1 second
            System.out.println("Main thread waited for 1 second.");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Main thread proceeding without waiting further.");
    }
}
```

Output

```arduino
Thread is running...
Main thread waited for 1 second.
Main thread proceeding without waiting further.
Thread finished.
```

Use Cases for `join()`

1. **Data Dependency**:
    - Ensuring data produced by one thread is available before another thread starts processing.
2. **Thread Synchronization**:
    - Making sure all child threads complete before the main thread exits.
3. **Order of Execution**:
    - Ensuring sequential execution where needed in multithreaded applications.

`join()` is a powerful tool for controlling thread execution and ensuring orderly thread completion in multithreaded Java programs.

What are Threadpools in Java. Kindly explain it in details along with example

Thread Pools in Java

A **Thread Pool** in Java is a collection of pre-created threads that can be reused to execute multiple tasks. Instead of creating and destroying threads for each task, the thread pool efficiently manages the thread lifecycle, improving performance and reducing resource consumption.

Why Use Thread Pools?

1. **Improved Performance**:
    - Reduces the overhead of thread creation and destruction.
2. **Resource Management**:
    - Controls the number of concurrent threads, preventing excessive resource usage.
3. **Scalability**:
    - Allows applications to scale efficiently by managing thread reuse.
4. **Thread Safety**:
    - Ensures that the thread pool is synchronized and thread-safe.

How Thread Pools Work

1. Tasks are submitted to the pool as `Runnable` or `Callable` objects.
2. A thread from the pool picks up the task and executes it.
3. After completing the task, the thread returns to the pool, ready for reuse.
4. The number of threads in the pool can be fixed or dynamically managed.

Thread Pools in Java: Using `Executor` Framework

The `java.util.concurrent.Executor` framework provides a standard way to create and manage thread pools.

Commonly Used Thread Pool Methods:

1. `Executors.newFixedThreadPool(int nThreads)`:
   - Creates a thread pool with a fixed number of threads.
2. `Executors.newCachedThreadPool()`:
   - Creates a thread pool with dynamically adjustable threads.
3. `Executors.newSingleThreadExecutor()`:
   - Creates a single-threaded executor.
4. `Executors.newScheduledThreadPool(int nThreads)`:
   - Creates a thread pool for scheduled tasks.

Example: Fixed Thread Pool

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

public class ThreadPoolExample {
    public static void main(String[] args) {
        // Create a fixed thread pool with 3 threads
        ExecutorService executor = Executors.newFixedThreadPool(3);

        // Submit tasks to the thread pool
        for (int i = 1; i <= 5; i++) {
            int taskNumber = i; // Task identifier
            executor.submit(() -> {
                System.out.println("Task " + taskNumber + " is running on thread: " + Thread.currentThread().getName());
                try {
                    Thread.sleep(1000); // Simulate task execution time
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("Task " + taskNumber + " completed.");
            });
        }

        // Shutdown the thread pool
        executor.shutdown();
    }
}
```

Explanation:

1. **Fixed Thread Pool**:
    - The thread pool has 3 threads.
    - If more than 3 tasks are submitted, the extra tasks wait in a queue until a thread becomes available.
2. **Task Execution**:
    - Each task is assigned to a thread for execution.
3. **Shutdown**:
    - The `` `shutdown()` `` method prevents new tasks from being submitted but allows existing tasks to complete.

Output:

```arduino
Task 1 is running on thread: pool-1-thread-1
Task 2 is running on thread: pool-1-thread-2
Task 3 is running on thread: pool-1-thread-3
Task 1 completed.
Task 4 is running on thread: pool-1-thread-1
Task 2 completed.
Task 5 is running on thread: pool-1-thread-2
Task 3 completed.
Task 4 completed.
Task 5 completed.
```

Example: Cached Thread Pool

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

public class CachedThreadPoolExample {
    public static void main(String[] args) {
        // Create a cached thread pool
        ExecutorService executor = Executors.newCachedThreadPool();

        // Submit tasks to the thread pool
```

```
        for (int i = 1; i <= 5; i++) {
            int taskNumber = i;
            executor.submit(() -> {
                System.out.println("Task " + taskNumber + " is running on thread: " + Thread.currentThread().getName());
                try {
                    Thread.sleep(1000); // Simulate task execution
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            });
        }

        // Shutdown the thread pool
        executor.shutdown();
    }
}
```

Explanation:

- **Cached Thread Pool**:
  - Creates new threads as needed and reuses previously created threads if they are available.
  - Suitable for short-lived asynchronous tasks.
- Threads are dynamically created or reused based on the workload.

Output:

```arduino
Task 1 is running on thread: pool-1-thread-1
Task 2 is running on thread: pool-1-thread-2
Task 3 is running on thread: pool-1-thread-3
Task 4 is running on thread: pool-1-thread-4
Task 5 is running on thread: pool-1-thread-5
```

Example: Scheduled Thread Pool

```java
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

public class ScheduledThreadPoolExample {
    public static void main(String[] args) {
        // Create a scheduled thread pool with 2 threads
        ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(2);

        // Schedule a task to run after 3 seconds
        scheduler.schedule(() -> {
            System.out.println("Task executed after 3 seconds on thread: " + Thread.currentThread().getName());
        }, 3, TimeUnit.SECONDS);

        // Schedule a task to run repeatedly every 2 seconds
        scheduler.scheduleAtFixedRate(() -> {
            System.out.println("Repeating task running on thread: " + Thread.currentThread().getName());
        }, 1, 2, TimeUnit.SECONDS);

        // Shutdown the scheduler after 10 seconds
        scheduler.schedule(() -> {
            scheduler.shutdown();
            System.out.println("Scheduler shut down.");
        }, 10, TimeUnit.SECONDS);
    }
}
```

Explanation:

- **Scheduled Thread Pool**:
  - Allows tasks to be scheduled for one-time execution after a delay or periodic execution.
  - Useful for tasks like periodic polling or scheduled notifications.

Output:

```arduino
Task executed after 3 seconds on thread: pool-1-thread-1
Repeating task running on thread: pool-1-thread-2
Repeating task running on thread: pool-1-thread-1
```

```
Repeating task running on thread: pool-1-thread-2
...
Scheduler shut down.
```

Advantages of Thread Pools

1. **Efficient Resource Usage**:
   - Reuses threads, minimizing the overhead of creating and destroying threads.
2. **Task Queueing**:
   - Manages a queue of tasks to be executed by available threads.
3. **Scalability**:
   - Dynamically adjusts thread usage based on workload (in some pool types).
4. **Thread Safety**:
   - Ensures proper synchronization of thread resources.

When to Use Thread Pools

- Applications with multiple independent tasks.
- High-performance server applications like web servers or database servers.
- Repetitive or scheduled tasks.

Thread pools are a powerful tool in Java's multithreading ecosystem, providing a flexible, efficient, and thread-safe way to manage concurrent tasks.

What is Executor Framework.Kindly explain it in details along with example
- What is ThreadPoolExecutor.Kindly explain it in details along with example

Executor Framework in Java

The **Executor Framework** in Java is a high-level API introduced in Java 5 (`java.util.concurrent` package) that provides a powerful mechanism for managing and controlling thread execution. It abstracts away the complexity of manually creating, managing, and coordinating threads by offering a flexible framework to execute tasks asynchronously.

Key Components of Executor Framework

1. **Executor Interface**:
    - The root interface for task execution.
    - Provides a single method:

      ```java
      void execute(Runnable command);
      ```

    - Example: `Executor executor = Executors.newFixedThreadPool(2);`
2. **ExecutorService Interface**:
    - An extension of `Executor` with additional methods for managing and controlling thread lifecycles:
        - `submit()`: Accepts `Runnable` or `Callable` and returns a `Future` object.
        - `shutdown()`: Initiates an orderly shutdown of the executor.
        - `shutdownNow()`: Attempts to stop all actively executing tasks.
3. **Executors Class**:
    - A utility class that provides factory methods to create preconfigured thread pools:
        - `newFixedThreadPool()`
        - `newCachedThreadPool()`
        - `newSingleThreadExecutor()`
        - `newScheduledThreadPool()`

Example: Basic Executor Framework

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

public class ExecutorExample {
    public static void main(String[] args) {
        // Create a fixed thread pool with 2 threads
        ExecutorService executor = Executors.newFixedThreadPool(2);

        // Submit tasks to the executor
        for (int i = 1; i <= 5; i++) {
            int taskNumber = i;
            executor.submit(() -> {
                System.out.println("Task " + taskNumber + " is running on thread: " + Thread.currentThread().getName());
                try {
                    Thread.sleep(1000); // Simulate task execution
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            });
        }

        // Shut down the executor
        executor.shutdown();
    }
}
```

Output:

```arduino
Task 1 is running on thread: pool-1-thread-1
Task 2 is running on thread: pool-1-thread-2
Task 3 is running on thread: pool-1-thread-1
Task 4 is running on thread: pool-1-thread-2
Task 5 is running on thread: pool-1-thread-1
```

Explanation:

- A fixed thread pool of 2 threads is created.
- Tasks are submitted to the executor and executed by the available threads in the pool.

- The `shutdown()` method ensures no new tasks can be submitted.

ThreadPoolExecutor

`ThreadPoolExecutor` is a concrete implementation of the `ExecutorService` interface. It is a highly configurable thread pool implementation that provides fine-grained control over thread behavior, task queuing, and thread lifecycle management.

Key Parameters of ThreadPoolExecutor Constructor

1. `corePoolSize`:
   - Minimum number of threads to keep in the pool, even if they are idle.
2. `maximumPoolSize`:
   - Maximum number of threads in the pool.
3. `keepAliveTime`:
   - Time to keep idle threads alive when the number of threads exceeds `corePoolSize`.
4. `unit`:
   - The time unit for `keepAliveTime`.
5. `workQueue`:
   - A queue to hold tasks before they are executed.
6. `threadFactory`:
   - Factory for creating new threads.
7. `handler`:
   - A rejection handler for tasks that cannot be executed (e.g., due to a full queue).

Example: Using ThreadPoolExecutor

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

public class ThreadPoolExecutorExample {
    public static void main(String[] args) {
        // Create a ThreadPoolExecutor
        ThreadPoolExecutor executor = new ThreadPoolExecutor(
            2, // corePoolSize
```

```
            4, // maximumPoolSize
            60, // keepAliveTime
            TimeUnit.SECONDS, // TimeUnit for keepAliveTime
            new ArrayBlockingQueue<>(2), // Work queue with a capacity of 2
            Executors.defaultThreadFactory(), // Default thread factory
            new ThreadPoolExecutor.AbortPolicy() // Rejection policy
        );

        // Submit tasks to the executor
        for (int i = 1; i <= 6; i++) {
            int taskNumber = i;
            executor.submit(() -> {
                System.out.println("Task " + taskNumber + " is running on thread: " + Thread.currentThread().getName());
                try {
                    Thread.sleep(1000); // Simulate task execution
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            });
        }

        // Shut down the executor
        executor.shutdown();
    }
}
```

Explanation

1. **ThreadPoolExecutor Setup**:
    - Core pool size: 2 threads.
    - Maximum pool size: 4 threads.
    - Task queue: Holds 2 tasks.
    - Rejection policy: **AbortPolicy** (throws `RejectedExecutionException` if a task cannot be accepted).
2. **Behavior**:
    - Initially, 2 threads are created to execute tasks.
    - If more tasks arrive and the queue is full, additional threads (up to `maximumPoolSize`) are created.
    - If the queue is full and no threads are available, tasks are rejected.

Output

```arduino
Task 1 is running on thread: pool-1-thread-1
Task 2 is running on thread: pool-1-thread-2
Task 3 is running on thread: pool-1-thread-3
Task 4 is running on thread: pool-1-thread-4
Task 5 is running on thread: pool-1-thread-1
Task 6 is running on thread: pool-1-thread-2
```

Rejection Policies in ThreadPoolExecutor

1. **AbortPolicy (Default)**:
   - Throws `RejectedExecutionException` for new tasks when the queue is full.
2. **CallerRunsPolicy**:
   - Runs the task in the calling thread if the queue is full.
3. **DiscardPolicy**:
   - Silently discards the rejected task.
4. **DiscardOldestPolicy**:
   - Discards the oldest task in the queue and adds the new task.

When to Use ThreadPoolExecutor

- Applications requiring **fine-grained control** over thread pool configuration.
- Scenarios where task rejection, custom thread factories, or dynamic thread pool resizing is necessary.

Summary

Feature	Executor Framework	ThreadPoolExecutor
Abstraction Level	High-level API for executing tasks.	Low-level implementation for configuring thread pools.
Customization	Limited customization via factory methods.	Full control over thread behavior, task queuing, and rejection policies.
Use Case	General-purpose task execution.	Advanced thread management and fine-tuned performance.

The **Executor Framework** and **ThreadPoolExecutor** are essential tools in Java for managing threads efficiently and handling concurrent tasks in a scalable and maintainable way.

- What is Callable and Future.Kindly explain it in details along with example
- What is Fork/Join Framework.Kindly explain it in details along with example

Callable and Future

`Callable` and `Future` are part of the `java.util.concurrent` package, introduced in Java 5, and are used for tasks that require a return value or may throw a checked exception.

---

1. Callable

- `Callable<T>` is a functional interface that represents a task that returns a result of type `T` and can throw exceptions.
- Unlike `Runnable`, which does not return a result or throw checked exceptions, `Callable` provides these capabilities.

Key Features:

- The single method `call()` must be implemented.
- The result of `call()` is returned as a `Future` object.

Example: Callable Implementation

```java
import java.util.concurrent.Callable;

public class CallableExample {
    public static void main(String[] args) throws Exception {
        Callable<Integer> task = () -> {
            System.out.println("Calculating sum...");
            Thread.sleep(2000); // Simulate some work
            return 10 + 20;
```

```
        };

        System.out.println("Result: " + task.call());
    }
}
```

---

2. Future

- `Future<T>` represents the result of an asynchronous computation.
- A `Future` object is returned when a `Callable` task is submitted to an `ExecutorService`.
- Methods:
    - `get()`: Retrieves the result, blocking if the task is not yet completed.
    - `isDone()`: Checks if the task is completed.
    - `cancel()`: Attempts to cancel the task.

Callable and Future with ExecutorService

Example:

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

public class CallableFutureExample {
    public static void main(String[] args) {
        // Create an ExecutorService
        ExecutorService executor = Executors.newSingleThreadExecutor();

        // Submit a Callable task
        Callable<String> task = () -> {
            Thread.sleep(2000); // Simulate computation
            return "Task Completed";
        };

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

        // Perform other operations
        System.out.println("Task submitted. Doing other work...");

        try {
```

```
            // Get the result of the Callable task
            String result = future.get(); // Blocks until the result is available
            System.out.println("Result: " + result);
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        } finally {
            executor.shutdown();
        }
    }
}
```

Output:

```arduino
Task submitted. Doing other work...
Result: Task Completed
```

Fork/Join Framework

The **Fork/Join Framework**, introduced in Java 7, is designed for parallelism and is a part of the `java.util.concurrent` package. It splits a large task into smaller subtasks (forking) and then combines their results (joining) to complete the task. This framework is especially effective for **divide-and-conquer algorithms**.

Key Concepts

1. **Work-Stealing Algorithm**:
    - Threads that complete their tasks early can "steal" work from other threads, maximizing CPU utilization.
2. **ForkJoinPool**:
    - A special thread pool optimized for Fork/Join tasks.
    - Automatically manages worker threads.
3. **RecursiveTask**:
    - A subclass of `ForkJoinTask` that computes a result.
4. **RecursiveAction**:
    - A subclass of `ForkJoinTask` that performs an action without returning a result.

Example: RecursiveTask

Let's compute the sum of an array using the Fork/Join Framework.

```java
import java.util.concurrent.RecursiveTask;
import java.util.concurrent.ForkJoinPool;

class ArraySumTask extends RecursiveTask<Integer> {
    private final int[] arr;
    private final int start, end;
    private static final int THRESHOLD = 10;

    public ArraySumTask(int[] arr, int start, int end) {
        this.arr = arr;
        this.start = start;
        this.end = end;
    }

    @Override
    protected Integer compute() {
        // If the task is small enough, compute directly
        if (end - start <= THRESHOLD) {
            int sum = 0;
            for (int i = start; i < end; i++) {
                sum += arr[i];
            }
            return sum;
        }

        // Otherwise, split the task into subtasks
        int mid = (start + end) / 2;
        ArraySumTask leftTask = new ArraySumTask(arr, start, mid);
        ArraySumTask rightTask = new ArraySumTask(arr, mid, end);

        // Fork subtasks
        leftTask.fork();
        int rightResult = rightTask.compute();
        int leftResult = leftTask.join();

        // Combine results
        return leftResult + rightResult;
    }
}

public class ForkJoinExample {
```

```java
    public static void main(String[] args) {
        int[] arr = new int[100];
        for (int i = 0; i < arr.length; i++) {
            arr[i] = i + 1; // Fill array with values 1 to 100
        }

        ForkJoinPool pool = new ForkJoinPool();

        ArraySumTask task = new ArraySumTask(arr, 0, arr.length);

        int result = pool.invoke(task);

        System.out.println("Sum: " + result);
    }
}
```

Explanation

1. **RecursiveTask Implementation**:
    - The `ArraySumTask` class extends `RecursiveTask<Integer>` for returning an integer result.
    - The array is divided into smaller chunks based on a threshold (`THRESHOLD = 10`).
2. **Fork/Join Process**:
    - If the array chunk is small enough, it is processed directly.
    - Otherwise, the task is split into two subtasks, which are recursively processed in parallel.
3. **ForkJoinPool**:
    - Manages and optimizes thread usage.
    - The `invoke()` method starts the task.

Output:

```yaml
Sum: 5050
```

When to Use Callable/Future and Fork/Join

Feature	Callable/Future	Fork/Join Framework
Use Case	For tasks with a single return value.	For tasks that can be broken into smaller subtasks.
Concurrency Model	Executor-based thread management.	Divide-and-conquer model with work-stealing threads.
Complexity	Simpler to implement.	Requires deeper understanding of task splitting.
Example Scenario	Fetching data from a database.	Large computations like array processing, sorting.

Both **Callable/Future** and the **Fork/Join Framework** are essential tools in Java's concurrency API, catering to different needs in multithreaded programming.

What is Executors and ExecutorService.Kindly explain it in details along with example

Executors and ExecutorService in Java

The `Executors` class and `ExecutorService` interface are part of Java's `java.util.concurrent` package, introduced in Java 5. They simplify the process of creating and managing threads for asynchronous task execution. These abstractions provide better performance and resource management compared to manually creating threads.

What is Executors?

The `Executors` class is a utility class that provides factory methods for creating and managing thread pools. Instead of managing threads manually, you can use these factory methods to create an instance of `ExecutorService` or `ScheduledExecutorService`.

Common Factory Methods in `Executors`

1. `newFixedThreadPool(int nThreads)`:
    - Creates a thread pool with a fixed number of threads.
    - Excess tasks are queued until a thread becomes available.
2. `newCachedThreadPool()`:
    - Creates a thread pool with dynamically adjustable threads.

   - Ideal for short-lived asynchronous tasks.
3. `newSingleThreadExecutor()`:
   - Creates a single-threaded executor to execute tasks sequentially.
4. `newScheduledThreadPool(int nThreads)`:
   - Creates a thread pool for scheduling tasks to run after a delay or at fixed intervals.

What is ExecutorService?

The `ExecutorService` interface extends the `Executor` interface and provides a higher-level API for managing thread execution and lifecycles. It is the core interface that provides methods for:

1. **Submitting tasks**:
   - `submit()`: Accepts `Runnable` or `Callable` and returns a `Future` object.
2. **Shutting down the executor**:
   - `shutdown()`: Initiates an orderly shutdown.
   - `shutdownNow()`: Attempts to stop all actively executing tasks.
3. **Controlling thread behavior**:
   - `invokeAll()` and `invokeAny()` to execute multiple tasks.

Example: Using Executors and ExecutorService

1. Fixed Thread Pool

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

public class ExecutorsExample {
    public static void main(String[] args) {
        // Create a fixed thread pool with 3 threads
        ExecutorService executor = Executors.newFixedThreadPool(3);

        // Submit tasks to the executor
        for (int i = 1; i <= 5; i++) {
            int taskNumber = i; // Task identifier
```

```
            executor.submit(() -> {
                System.out.println("Task " + taskNumber + " is running on thread: " + Thread.currentThread().getName());
                try {
                    Thread.sleep(1000); // Simulate task execution
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            });
        }

        // Shutdown the executor
        executor.shutdown();
    }
}
```

Output:

```arduino
Task 1 is running on thread: pool-1-thread-1
Task 2 is running on thread: pool-1-thread-2
Task 3 is running on thread: pool-1-thread-3
Task 4 is running on thread: pool-1-thread-1
Task 5 is running on thread: pool-1-thread-2
```

Explanation:

- A fixed thread pool of size 3 is created.
- Tasks are submitted to the pool and executed by the available threads.
- Excess tasks are queued until threads become available.

---

2. Cached Thread Pool

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

```java
public class CachedThreadPoolExample {
    public static void main(String[] args) {
        // Create a cached thread pool
        ExecutorService executor = Executors.newCachedThreadPool();

        // Submit tasks to the executor
        for (int i = 1; i <= 5; i++) {
            int taskNumber = i;
            executor.submit(() -> {
                System.out.println("Task " + taskNumber + " is running on thread: " + Thread.currentThread().getName());
                try {
                    Thread.sleep(1000); // Simulate task execution
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            });
        }

        // Shutdown the executor
        executor.shutdown();
    }
}
```

Output:

```arduino
Task 1 is running on thread: pool-1-thread-1
Task 2 is running on thread: pool-1-thread-2
Task 3 is running on thread: pool-1-thread-3
Task 4 is running on thread: pool-1-thread-4
Task 5 is running on thread: pool-1-thread-5
```

Explanation:

- Threads are created dynamically as needed.
- Ideal for handling a large number of short-lived asynchronous tasks.

---

3. Scheduled Thread Pool

```java
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

public class ScheduledExecutorExample {
    public static void main(String[] args) {
        // Create a scheduled thread pool with 2 threads
        ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(2);

        // Schedule a task to run after 3 seconds
        scheduler.schedule(() -> {
            System.out.println("Task executed after 3 seconds on thread: " + Thread.currentThread().getName());
        }, 3, TimeUnit.SECONDS);

        // Schedule a repeating task
        scheduler.scheduleAtFixedRate(() -> {
            System.out.println("Repeating task running on thread: " + Thread.currentThread().getName());
        }, 1, 2, TimeUnit.SECONDS);

        // Shut down the scheduler after 10 seconds
        scheduler.schedule(() -> {
            scheduler.shutdown();
            System.out.println("Scheduler shut down.");
        }, 10, TimeUnit.SECONDS);
    }
}
```

Output:

```arduino
Task executed after 3 seconds on thread: pool-1-thread-1
Repeating task running on thread: pool-1-thread-2
Repeating task running on thread: pool-1-thread-1
...
Scheduler shut down.
```

Advantages of Executors and ExecutorService

1. **Thread Management**:
   - Handles thread creation and termination automatically.
2. **Resource Efficiency**:
   - Reuses threads to minimize resource consumption.
3. **Scalability**:
   - Adapts to workload demands.
4. **Simplified API**:
   - Factory methods in `Executors` make thread pool creation easy.

Comparison Between Executors and ExecutorService

Feature	Executors	ExecutorService
Purpose	Utility class for creating thread pools.	Interface for managing thread execution.
Methods	Factory methods for thread pool creation.	Methods for task submission and lifecycle control.
Example Usage	`Executors.newFixedThreadPool(3);`	`executor.submit(task);`

Key Takeaways

- `Executors`: Provides factory methods to create thread pools.
- `ExecutorService`: Manages the lifecycle of thread pools, allowing task submission and execution control.

By combining `Executors` and `ExecutorService`, you can efficiently manage multithreaded tasks in Java.

What is CompletableFuture.Kindly explain it in details along with example

CompletableFuture in Java

`CompletableFuture`, introduced in **Java 8**, is part of the `java.util.concurrent` package. It represents a future result of an asynchronous computation and provides a powerful and flexible API for handling asynchronous tasks.

Unlike the traditional `Future` interface, `CompletableFuture` allows you to:

- Combine multiple asynchronous tasks.
- Apply callbacks when a task completes.
- Handle errors in a more functional and streamlined way.
- Avoid blocking calls where possible.

Key Features of CompletableFuture

1. **Non-blocking Operations**:
   - Supports asynchronous, non-blocking computations with methods like `supplyAsync()` and `runAsync()`.
2. **Chaining**:
   - You can chain multiple computations using methods like `thenApply()`, `thenAccept()`, and `thenRun()`.
3. **Combining Futures**:
   - Combine results from multiple futures using `thenCombine()` or `allOf()`.
4. **Exception Handling**:
   - Handle errors gracefully with `exceptionally()` or `handle()`.
5. **Custom Executors**:
   - You can provide a custom executor for executing tasks.

Key Methods of CompletableFuture

Method	Description
`supplyAsync()`	Runs a `Supplier` function asynchronously and returns a result.
`runAsync()`	Runs a `Runnable` asynchronously but does not return a result.
`thenApply()`	Processes the result of the previous stage and returns a new result.
`thenAccept()`	Consumes the result of the previous stage without returning a value.

Method	Description
`thenRun()`	Executes a `Runnable` after the previous stage completes but does not return or accept a value.
`thenCombine()`	Combines the results of two `CompletableFuture` instances.
`allOf()`	Waits for all given `CompletableFuture` instances to complete.
`exceptionally()`	Handles exceptions from a computation and provides a fallback value.
`handle()`	Processes both the result and any exceptions from the previous stage.

Example 1: Basic Asynchronous Task

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

public class CompletableFutureExample {
    public static void main(String[] args) {
        // Run an asynchronous task
        CompletableFuture<Void> future = CompletableFuture.runAsync(() -> {
            System.out.println("Task is running asynchronously on thread: " + Thread.currentThread().getName());
        });

        // Wait for the task to complete
        future.join();
        System.out.println("Main thread completed.");
    }
}
```

Output:

```arduino
Task is running asynchronously on thread: ForkJoinPool.commonPool-worker-1
Main thread completed.
```

Example 2: Returning a Result

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

public class CompletableFutureResultExample {
    public static void main(String[] args) {
        // Run an asynchronous task with a result
        CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
            System.out.println("Calculating result...");
            return "Result from async task";
        });

        // Process the result
        future.thenAccept(result -> System.out.println("Received: " + result));

        // Wait for completion
        future.join();
    }
}
```

Output:

```vbnet
Calculating result...
Received: Result from async task
```

Example 3: Chaining Tasks

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

public class CompletableFutureChainingExample {
    public static void main(String[] args) {
        CompletableFuture.supplyAsync(() -> {
            System.out.println("Step 1: Fetching data...");
            return 100;
        }).thenApply(data -> {
```

```
            System.out.println("Step 2: Processing data...");
            return data * 2;
        }).thenAccept(result -> {
            System.out.println("Step 3: Result is " + result);
        }).join();
    }
}
```

Output:

```vbnet
Step 1: Fetching data...
Step 2: Processing data...
Step 3: Result is 200
```

Example 4: Combining Futures

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

public class CompletableFutureCombineExample {
    public static void main(String[] args) {
        CompletableFuture<Integer> future1 = CompletableFuture.supplyAsync(() -> {
            System.out.println("Task 1: Fetching number...");
            return 10;
        });

        CompletableFuture<Integer> future2 = CompletableFuture.supplyAsync(() -> {
            System.out.println("Task 2: Fetching multiplier...");
            return 5;
        });

        CompletableFuture<Integer> combined = future1.thenCombine(future2, (number, multiplier) -> {
            System.out.println("Combining results...");
            return number * multiplier;
        });

        System.out.println("Final Result: " + combined.join());
```

```
    }
}
```

Output:

```yaml
Task 1: Fetching number...
Task 2: Fetching multiplier...
Combining results...
Final Result: 50
```

Example 5: Exception Handling

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

public class CompletableFutureExceptionExample {
    public static void main(String[] args) {
        CompletableFuture<Integer> future = CompletableFuture.supplyAsync(() -> {
            System.out.println("Performing computation...");
            if (true) throw new RuntimeException("Error during computation");
            return 100;
        }).exceptionally(ex -> {
            System.out.println("Handling exception: " + ex.getMessage());
            return -1;
        });

        System.out.println("Result: " + future.join());
    }
}
```

Output:

```vbnet
Performing computation...
Handling exception: Error during computation
Result: -1
```

Key Advantages of CompletableFuture

1. **Non-blocking API**:
   - No need for blocking `get()` calls, making it more efficient for asynchronous programming.
2. **Chaining and Pipelining**:
   - Easy to chain multiple tasks to process data in a sequence.
3. **Parallel Processing**:
   - Combine multiple tasks running in parallel using `allOf()` or `anyOf()`.
4. **Error Handling**:
   - Provides robust error handling mechanisms for asynchronous tasks.

Comparison Between Future and CompletableFuture

Feature	Future	CompletableFuture
Result Handling	Blocks the calling thread to get the result.	Allows non-blocking result processing.
Task Chaining	Not supported.	Supports task chaining with methods like `thenApply()`.
Exception Handling	Not directly supported.	Built-in methods like `exceptionally()` for handling errors.
Combining Results	Not supported.	Combines multiple futures with `thenCombine()` or `allOf()`.

When to Use CompletableFuture

1. **Asynchronous API Calls**:
   - Fetch data from multiple APIs in parallel and process results.

2. **Background Processing**:
   - Perform computations or I/O operations in the background without blocking the main thread.
3. **Complex Task Pipelines**:
   - Chain dependent tasks with data processing and error handling.
4. **High-Performance Applications**:
   - Use for scalable, non-blocking programming in modern Java applications.

`CompletableFuture` is a powerful and flexible tool for handling asynchronous programming in Java, offering advanced features for efficient and clean multithreaded applications.

What is Scheduled ExecutorService.Kindly explain it in details along with example

ScheduledExecutorService in Java

The `ScheduledExecutorService`, part of the `java.util.concurrent` package, is an interface that extends `ExecutorService`. It is specifically designed to schedule tasks to run after a delay or repeatedly at fixed intervals. This makes it ideal for tasks like periodic updates, scheduled background tasks, or recurring maintenance operations.

Key Features of ScheduledExecutorService

1. **Delay Execution**:
   - Schedule tasks to run after a specified delay.
2. **Periodic Execution**:
   - Schedule tasks to run repeatedly at fixed intervals.
3. **Flexible Scheduling**:
   - Tasks can be `Runnable` or `Callable`.
4. **Thread Pool Management**:

- Backed by a thread pool to efficiently manage threads.

Key Methods

Method	Description
`schedule(Runnable task, long delay, TimeUnit unit)`	Schedules a `Runnable` to execute after the specified delay.
`schedule(Callable task, long delay, TimeUnit unit)`	Schedules a `Callable` to execute after the specified delay and returns a `Future`.
`scheduleAtFixedRate(Runnable task, long initialDelay, long period, TimeUnit unit)`	Schedules a task to run periodically at a fixed rate.
`scheduleWithFixedDelay(Runnable task, long initialDelay, long delay, TimeUnit unit)`	Schedules a task to run periodically with a fixed delay between the end of one execution and the start of the next.

Difference Between `scheduleAtFixedRate` and `scheduleWithFixedDelay`

Feature	scheduleAtFixedRate	scheduleWithFixedDelay
Start Timing	Starts the next execution at a fixed interval.	Starts the next execution after the previous task completes.
Usage	For tasks that need to execute at a regular interval.	For tasks with varying execution times.

Example 1: Scheduling a Task After a Delay

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

public class ScheduledExecutorExample {
    public static void main(String[] args) {
        ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(1);

        // Schedule a task to run after a 3-second delay
        scheduler.schedule(() -> {
            System.out.println("Task executed after 3 seconds on thread: " + Thread.currentThread().getName());
        }, 3, TimeUnit.SECONDS);

        // Shutdown the scheduler after the task is executed
```

```
        scheduler.schedule(() -> {
            scheduler.shutdown();
            System.out.println("Scheduler shut down.");
        }, 4, TimeUnit.SECONDS);
    }
}
```

Output:

```arduino
Task executed after 3 seconds on thread: pool-1-thread-1
Scheduler shut down.
```

Example 2: Scheduling Periodic Tasks with Fixed Rate

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

public class FixedRateExample {
    public static void main(String[] args) {
        ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(2);

        // Schedule a task to run every 2 seconds, starting after an initial delay of 1 second
        scheduler.scheduleAtFixedRate(() -> {
            System.out.println("Fixed-rate task running at: " + System.currentTimeMillis());
        }, 1, 2, TimeUnit.SECONDS);

        // Shut down the scheduler after 10 seconds
        scheduler.schedule(() -> {
            scheduler.shutdown();
            System.out.println("Scheduler shut down.");
        }, 10, TimeUnit.SECONDS);
    }
}
```

Output:

```arduino
Fixed-rate task running at: 1670000001000
Fixed-rate task running at: 1670000003000
Fixed-rate task running at: 1670000005000
...
Scheduler shut down.
```

Example 3: Scheduling Periodic Tasks with Fixed Delay

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

public class FixedDelayExample {
    public static void main(String[] args) {
        ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(2);

        // Schedule a task to run every 2 seconds, starting after an initial delay of 1 second
        scheduler.scheduleWithFixedDelay(() -> {
            System.out.println("Fixed-delay task running at: " + System.currentTimeMillis());
            try {
                Thread.sleep(1000); // Simulate task execution taking 1 second
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }, 1, 2, TimeUnit.SECONDS);

        // Shut down the scheduler after 10 seconds
        scheduler.schedule(() -> {
            scheduler.shutdown();
            System.out.println("Scheduler shut down.");
        }, 10, TimeUnit.SECONDS);
    }
}
```

Output:

```arduino
Fixed-delay task running at: 1670000001000
Fixed-delay task running at: 1670000004000
Fixed-delay task running at: 1670000007000
...
Scheduler shut down.
```

Explanation of Examples

1. **Example 1**:
    - The `schedule()` method is used to delay the execution of a task by 3 seconds.
    - Another task is scheduled to shut down the scheduler after 4 seconds.
2. **Example 2**:
    - The `scheduleAtFixedRate()` method ensures the task starts at a regular interval (every 2 seconds) regardless of the task's execution time.
3. **Example 3**:
    - The `scheduleWithFixedDelay()` method ensures the next task starts after the current task finishes, with a fixed delay of 2 seconds.

Advantages of ScheduledExecutorService

1. **Efficient Resource Utilization**:
    - Manages threads efficiently by reusing threads from a pool.
2. **Precision**:
    - Provides precise scheduling with options for fixed-rate or fixed-delay execution.
3. **Scalability**:
    - Allows you to specify the number of threads in the pool, enabling better handling of multiple tasks.
4. **Ease of Use**:
    - Simplifies periodic and delayed task execution compared to using `Thread` or `Timer`.

Comparison: ScheduledExecutorService vs Timer

Feature	ScheduledExecutorService	Timer
Thread Management	Uses a thread pool.	Uses a single thread.
Task Execution	Handles multiple tasks concurrently.	Tasks are serialized, can block each other.
Exception Handling	Robust error handling, doesn't terminate other tasks.	Unhandled exceptions terminate the `Timer`.
Precision	More precise and scalable.	Limited precision and scalability.

When to Use ScheduledExecutorService

- Periodic tasks like polling, logging, or cleaning temporary files.
- Delayed tasks like scheduling reminders or sending notifications.
- Concurrently executing multiple timed tasks in a scalable and robust way.

The `ScheduledExecutorService` provides a powerful, flexible, and thread-safe solution for scheduling and managing asynchronous tasks in Java applications.

What is CountDownLatch, CyclicBarrier, Phaser, and Exchanger.Kindly explain it in details along with example

CountDownLatch, CyclicBarrier, Phaser, and Exchanger in Java

These classes are part of the `java.util.concurrent` package and are designed for coordinating threads in multithreaded applications. Each serves a unique purpose in managing thread interactions.

---

1. CountDownLatch

A `CountDownLatch` is used to block one or more threads until a set of operations in other threads completes. It uses a counter that is decremented by calls to `countDown()`. When the counter reaches zero, the threads waiting on the latch are released.

Key Points

- It cannot be reused once the count reaches zero.
- Useful for scenarios like waiting for multiple threads to complete before proceeding.

Example:

```java
import java.util.concurrent.CountDownLatch;

public class CountDownLatchExample {
    public static void main(String[] args) {
        CountDownLatch latch = new CountDownLatch(3);

        // Create worker threads
        for (int i = 1; i <= 3; i++) {
            final int workerId = i;
            new Thread(() -> {
                System.out.println("Worker " + workerId + " is doing work...");
                try {
                    Thread.sleep(workerId * 1000); // Simulate work
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("Worker " + workerId + " finished work.");
                latch.countDown(); // Decrement latch count
            }).start();
        }

        System.out.println("Main thread waiting for workers to finish...");
        try {
            latch.await(); // Wait until the count reaches zero
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("All workers finished. Main thread proceeds.");
    }
}
```

Output:

```css
Main thread waiting for workers to finish...
Worker 1 is doing work...
Worker 2 is doing work...
Worker 3 is doing work...
Worker 1 finished work.
Worker 2 finished work.
Worker 3 finished work.
All workers finished. Main thread proceeds.
```

---

2. CyclicBarrier

A `CyclicBarrier` is used to synchronize a group of threads to a common barrier point. All threads must reach the barrier before any can proceed. Once all threads reach the barrier, the barrier is broken, and threads can continue execution.

Key Points

- It can be reused after the barrier is broken.
- Useful for scenarios like waiting for all threads to reach a checkpoint.

Example:

```java
import java.util.concurrent.CyclicBarrier;

public class CyclicBarrierExample {
    public static void main(String[] args) {
        CyclicBarrier barrier = new CyclicBarrier(3, () -> {
            System.out.println("All workers reached the barrier. Proceeding...");
        });

        // Create worker threads
        for (int i = 1; i <= 3; i++) {
            final int workerId = i;
            new Thread(() -> {
                System.out.println("Worker " + workerId + " is doing work...");
                try {
```

```
                        Thread.sleep(workerId * 1000); // Simulate work
                        System.out.println("Worker " + workerId + " reached the barrier.");
                        barrier.await(); // Wait at the barrier
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }).start();
            }
        }
    }
```

Output:

```arduino
Worker 1 is doing work...
Worker 2 is doing work...
Worker 3 is doing work...
Worker 1 reached the barrier.
Worker 2 reached the barrier.
Worker 3 reached the barrier.
All workers reached the barrier. Proceeding...
```

---

3. Phaser

A `Phaser` is a more flexible synchronization mechanism than `CyclicBarrier`. It allows threads to register dynamically and works in phases. Threads must wait for all others in the same phase before moving to the next phase.

Key Points

- Supports dynamic thread registration/deregistration.
- Useful for scenarios where the number of participating threads can change over time.

Example:

```java
import java.util.concurrent.Phaser;

public class PhaserExample {
    public static void main(String[] args) {
        Phaser phaser = new Phaser(1); // Main thread registered

        // Create worker threads
        for (int i = 1; i <= 3; i++) {
            final int workerId = i;
            phaser.register(); // Register a new party
            new Thread(() -> {
                System.out.println("Worker " + workerId + " is doing work...");
                try {
                    Thread.sleep(workerId * 1000); // Simulate work
                    System.out.println("Worker " + workerId + " completed phase 1.");
                    phaser.arriveAndAwaitAdvance(); // Wait for others in phase 1

                    System.out.println("Worker " + workerId + " completed phase 2.");
                    phaser.arriveAndDeregister(); // Deregister from further phases
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }).start();
        }

        phaser.arriveAndAwaitAdvance(); // Main thread waits for phase 1
        System.out.println("Phase 1 completed. Main thread proceeds.");

        phaser.arriveAndDeregister(); // Main thread deregisters
    }
}
```

Output:

```csharp
Worker 1 is doing work...
Worker 2 is doing work...
Worker 3 is doing work...
Worker 1 completed phase 1.
Worker 2 completed phase 1.
Worker 3 completed phase 1.
Phase 1 completed. Main thread proceeds.
```

```
Worker 1 completed phase 2.
Worker 2 completed phase 2.
Worker 3 completed phase 2.
```

---

4. Exchanger

An `Exchanger` allows two threads to exchange data. Each thread calls the `exchange()` method, which waits until the other thread also calls `exchange()` to swap data.

Key Points

- Designed for thread-pair communication.
- Useful for scenarios like producer-consumer or data transformation.

Example:

```java
import java.util.concurrent.Exchanger;

public class ExchangerExample {
    public static void main(String[] args) {
        Exchanger<String> exchanger = new Exchanger<>();

        // Thread 1
        new Thread(() -> {
            try {
                String data = "Data from Thread 1";
                System.out.println("Thread 1 exchanging: " + data);
                String receivedData = exchanger.exchange(data);
                System.out.println("Thread 1 received: " + receivedData);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }).start();

        // Thread 2
        new Thread(() -> {
            try {
                String data = "Data from Thread 2";
                System.out.println("Thread 2 exchanging: " + data);
```

```
                String receivedData = exchanger.exchange(data);
                System.out.println("Thread 2 received: " + receivedData);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }).start();
    }
}
```

Output:

```mathematica
Thread 1 exchanging: Data from Thread 1
Thread 2 exchanging: Data from Thread 2
Thread 1 received: Data from Thread 2
Thread 2 received: Data from Thread 1
```

Comparison of CountDownLatch, CyclicBarrier, Phaser, and Exchanger

Feature	CountDownLatch	CyclicBarrier	Phaser	Exchanger
Purpose	Wait for multiple threads to finish.	Wait for threads to reach a barrier.	Synchronize phases of threads.	Exchange data between two threads.
Reusability	No	Yes	Yes	Not applicable.
Dynamic Parties	No	No	Yes	No
Use Case	One-time events like initialization.	Checkpoints in concurrent tasks.	Dynamic tasks with multiple phases.	Data exchange between two threads.

Each of these tools offers unique capabilities and is suited for specific use cases in multithreaded programming.