Comprehensive Guide to Multithreading in Java

Trainer name

Shreyansh Kumar

Introduction to Multithreading

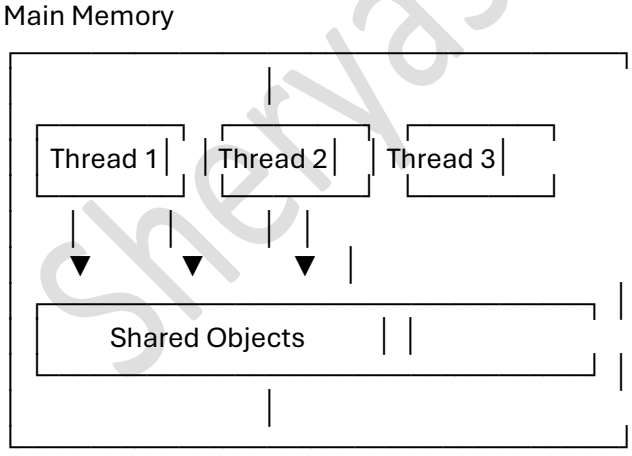
Multithreading is one of the most powerful features of Java, allowing developers to create applications that can perform multiple operations simultaneously. At its core, multithreading is about executing multiple threads concurrently within a single program. This capability is essential for developing efficient applications that can fully utilize modern multi-core processors and provide responsive user interfaces while performing background tasks.

In Java, multithreading is built into the language from the ground up. The Java Virtual Machine (JVM) manages threads, allowing developers to focus on the logic of concurrent execution rather than the low-level details of thread management. This comprehensive guide will explore all aspects of multithreading in Java, from the basics of creating threads to advanced concepts like thread synchronization, the producer-consumer problem, and daemon threads.

Understanding Threads in Java

A thread is the smallest unit of processing that can be scheduled by an operating system. In Java, threads are objects that encapsulate the execution of code. When a Java program starts, it automatically creates a main thread, which is responsible for executing the main method. From this main thread, developers can create additional threads to perform concurrent tasks.

Threads in Java share the same memory space, which means they can access the same objects and variables. This shared memory model is both powerful and dangerous—it enables efficient communication between threads but also introduces the potential for race conditions and other concurrency issues.

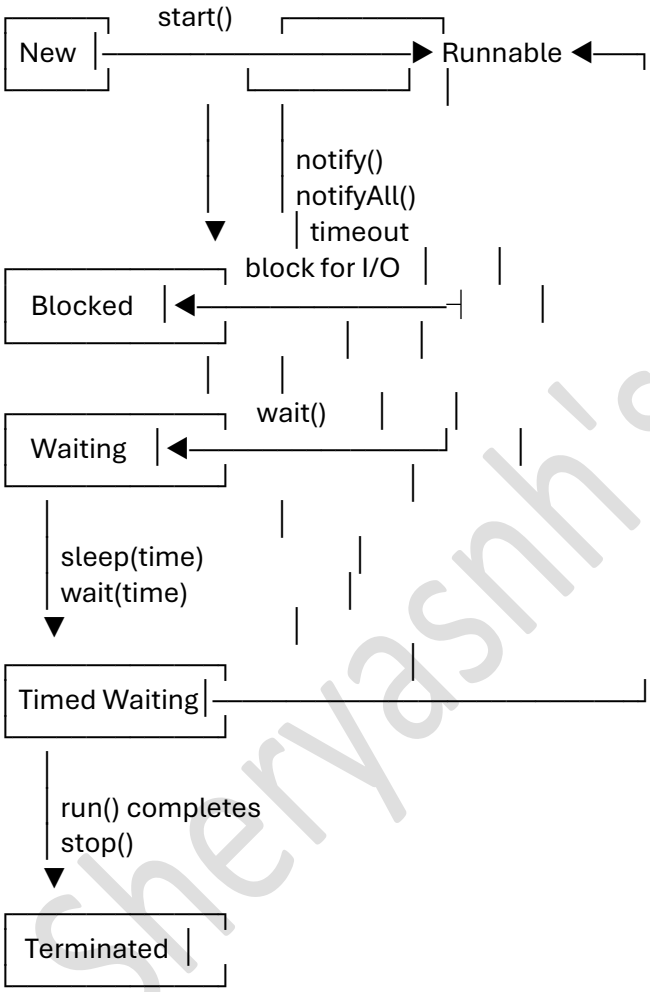


Thread Lifecycle in Java

Understanding the lifecycle of a thread is crucial for effective multithreading. A Java thread can exist in several states throughout its lifetime:

1. New: A thread that has been created but not yet started.
2. Runnable: A thread that is ready to run and waiting for CPU time.
3. Blocked: A thread that is waiting for a monitor lock to enter a synchronized block/method.
4. Waiting: A thread that is waiting indefinitely for another thread to perform a particular action.
5. Timed Waiting: A thread that is waiting for another thread to perform an action for a specified period.
6. Terminated: A thread that has completed execution or has been terminated.

Thread Lifecycle Diagram



Creating Threads in Java

Java provides multiple ways to create and start threads. Let's explore each approach in detail.

1. Extending the Thread Class

The most straightforward way to create a thread is by extending the Thread class and overriding its run() method. This approach is simple but has limitations because Java does not support multiple inheritance.

```
public class MyThread extends Thread {
  @Override
  public void run() {
    System.out.println("Thread is running: " + Thread.currentThread().getName());
    // Code to be executed in this thread
    for (int i = 0; i < 5; i++) {
      System.out.println(Thread.currentThread().getName() + ": " + i);
      try {
        Thread.sleep(1000); // Pause for 1 second
      } catch (InterruptedException e) {
        System.out.println("Thread interrupted");
      }
    }
  }

  public static void main(String[] args) {
    MyThread thread1 = new MyThread();
    thread1.setName("MyThread-1");
    thread1.start(); // Start the thread

    MyThread thread2 = new MyThread();
    thread2.setName("MyThread-2");
    thread2.start(); // Start another thread
  }
}
```

When you call the start() method, the JVM creates a new thread and calls the run() method of that thread. It's important to note that you should never call the run() method directly, as this would execute the code in the current thread rather than starting a new one.

2. Implementing the Runnable Interface

The preferred way to create a thread in Java is by implementing the Runnable interface. This approach separates the task (what to run) from the thread (how to run), following the principle of separation of concerns. It also allows your class to extend another class if needed.

```
public class MyRunnable implements Runnable {
  @Override
  public void run() {
    System.out.println("Thread is running: " + Thread.currentThread().getName());
    // Code to be executed in this thread
    for (int i = 0; i < 5; i++) {
```

```
            System.out.println(Thread.currentThread().getName() + ": " + i);
            try {
                Thread.sleep(1000); // Pause for 1 second
            } catch (InterruptedException e) {
                System.out.println("Thread interrupted");
            }
        }
    }

    public static void main(String[] args) {
        MyRunnable myRunnable = new MyRunnable();

        Thread thread1 = new Thread(myRunnable);
        thread1.setName("RunnableThread-1");
        thread1.start(); // Start the thread

        Thread thread2 = new Thread(myRunnable);
        thread2.setName("RunnableThread-2");
        thread2.start(); // Start another thread
    }
}
```

3. Using Anonymous Inner Classes

For simple thread tasks, you can use anonymous inner classes to create threads on the fly without defining a separate class.

```
public class AnonymousThreadExample {
    public static void main(String[] args) {
        // Using anonymous class extending Thread
        Thread thread1 = new Thread() {
            @Override
            public void run() {
                System.out.println("Anonymous Thread is running");
                // Thread code here
            }
        };
        thread1.start();

        // Using anonymous class implementing Runnable
        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                System.out.println("Anonymous Runnable is running");
                // Thread code here
            }
        };
        Thread thread2 = new Thread(runnable);
        thread2.start();
```

```
    }
}
```

4. Using Lambda Expressions (Java 8+)

With Java 8, you can use lambda expressions to create threads more concisely, as the Runnable interface is a functional interface with a single abstract method.

```
public class LambdaThreadExample {
    public static void main(String[] args) {
        // Using lambda expression
        Thread thread = new Thread(() -> {
            System.out.println("Lambda Thread is running");
            for (int i = 0; i < 5; i++) {
                System.out.println("Lambda Thread: " + i);
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });
        thread.start();
    }
}
```

5. Using the Executor Framework

For more advanced thread management, Java provides the Executor framework, which separates thread creation and management from the rest of your application. This approach is recommended for most applications as it provides better resource management.

```
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 3 threads
        ExecutorService executor = Executors.newFixedThreadPool(3);

        // Submit tasks to the executor
        for (int i = 0; i < 5; i++) {
            final int taskId = i;
            executor.submit(() -> {
                System.out.println("Task " + taskId + " is running on " +
                        Thread.currentThread().getName());
                // Task code here
                try {
```

```
            Thread.sleep(1000);
          } catch (InterruptedException e) {
            e.printStackTrace();
          }
        });
    }

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

Thread Methods and Properties

Java's Thread class provides numerous methods to control and query thread behavior. Here are some of the most important ones:

Starting a Thread

```
thread.start(); // Starts the thread by calling its run() method
```

Joining Threads

The join() method allows one thread to wait for the completion of another.

```
public class JoinExample {
  public static void main(String[] args) {
    Thread thread1 = new Thread(() -> {
      System.out.println("Thread 1 started");
      try {
        Thread.sleep(3000); // Simulate work
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
      System.out.println("Thread 1 completed");
    });

    Thread thread2 = new Thread(() -> {
      System.out.println("Thread 2 started");
      try {
        thread1.join(); // Wait for thread1 to complete
        System.out.println("Thread 1 joined, now Thread 2 continues");
        Thread.sleep(2000); // More work
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
      System.out.println("Thread 2 completed");
    });

    thread1.start();
```

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

Thread Sleep

The sleep() method pauses the current thread for a specified amount of time.

```
try {
    Thread.sleep(1000); // Sleep for 1 second
} catch (InterruptedException e) {
    // Handle interruption
}
```

Thread Interruption

Threads can be interrupted to signal that they should stop what they're doing.

```
public class InterruptExample {
    public static void main(String[] args) {
        Thread thread = new Thread(() -> {
            try {
                for (int i = 0; i < 10; i++) {
                    System.out.println("Working... " + i);
                    Thread.sleep(1000);
                }
            } catch (InterruptedException e) {
                System.out.println("Thread was interrupted!");
                return; // Exit the thread
            }
            System.out.println("Thread completed normally");
        });

        thread.start();

        // Let the thread run for a while
        try {
            Thread.sleep(3500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // Interrupt the thread
        thread.interrupt();
    }
}
```

Thread Priority

Java threads have priorities that can influence the thread scheduler's decisions.

```
thread.setPriority(Thread.MAX_PRIORITY); // 10
thread.setPriority(Thread.NORM_PRIORITY); // 5 (default)
thread.setPriority(Thread.MIN_PRIORITY); // 1
```

Thread Names

Giving threads meaningful names can help with debugging.

```
thread.setName("WorkerThread-1");
String threadName = thread.getName();
```

Thread State

You can query a thread's current state.

```
Thread.State state = thread.getState(); // Returns one of the Thread.State enum values
```

Thread Synchronization

When multiple threads access shared resources, synchronization is necessary to prevent race conditions and ensure data consistency. Java provides several mechanisms for thread synchronization.

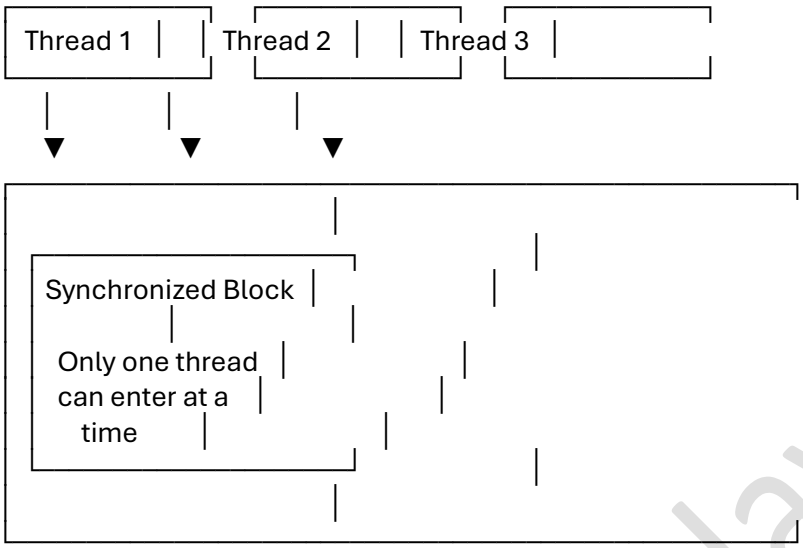
The Synchronized Keyword

The synchronized keyword can be applied to methods or blocks of code to ensure that only one thread can execute that code at a time.

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

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

  // Synchronized block
  public void incrementWithBlock() {
    synchronized(this) {
      count++;
    }
  }

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

When a thread enters a synchronized method or block, it acquires a lock (also called a monitor) on the specified object. Other threads attempting to enter any synchronized method or block on the same object will be blocked until the lock is released.

Thread Synchronization Diagram



Lock Interface

Java's java.util.concurrent.locks package provides more flexible locking mechanisms than the synchronized keyword.

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class LockExample {
  private final Lock lock = new ReentrantLock();
  private int count = 0;

  public void increment() {
    lock.lock(); // Acquire the lock
    try {
      count++;
    } finally {
      lock.unlock(); // Always release the lock in a finally block
    }
  }

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

Volatile Keyword

The volatile keyword ensures that a variable is always read from and written to main memory, rather than from thread-local caches. This guarantees visibility of changes to the variable across threads.

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

  public void setFlag() {
    flag = true; // This change is immediately visible to all threads
  }

  public boolean isFlag() {
    return flag;
  }
}
```

Atomic Classes

For simple operations that need to be atomic, Java provides atomic classes in the java.util.concurrent.atomic package.

```
import java.util.concurrent.atomic.AtomicInteger;

public class AtomicExample {
  private AtomicInteger count = new AtomicInteger(0);

  public void increment() {
    count.incrementAndGet(); // Atomic operation
  }

  public int getCount() {
    return count.get();
  }
}
```

Thread Communication

Threads often need to communicate with each other to coordinate their activities. Java provides several mechanisms for inter-thread communication.

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

These methods, inherited from the Object class, allow threads to communicate while synchronizing on the same object.

```
public class MessageQueue {
  private String message;
```

```
    private boolean empty = true;

    public synchronized String receive() {
        while (empty) {
            try {
                wait(); // Release lock and wait for a notification
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
        empty = true;
        notifyAll(); // Notify waiting threads
        return message;
    }

    public synchronized void send(String message) {
        while (!empty) {
            try {
                wait(); // Wait until the queue is empty
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
        empty = false;
        this.message = message;
        notifyAll(); // Notify waiting threads
    }
}
```

Condition Interface

The Condition interface, used with locks, provides more flexible waiting and signaling than the traditional wait() and notify() methods.

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class ConditionExample {
    private final Lock lock = new ReentrantLock();
    private final Condition notFull = lock.newCondition();
    private final Condition notEmpty = lock.newCondition();
    private final String[] buffer = new String[10];
    private int count = 0, putIndex = 0, takeIndex = 0;

    public void put(String item) throws InterruptedException {
        lock.lock();
        try {
            while (count == buffer.length) {
```

```
            notFull.await(); // Wait until buffer is not full
        }
        buffer[putIndex] = item;
        putIndex = (putIndex + 1) % buffer.length;
        count++;
        notEmpty.signal(); // Signal that buffer is not empty
    } finally {
        lock.unlock();
    }
  }

  public String take() throws InterruptedException {
    lock.lock();
    try {
        while (count == 0) {
            notEmpty.await(); // Wait until buffer is not empty
        }
        String item = buffer[takeIndex];
        takeIndex = (takeIndex + 1) % buffer.length;
        count--;
        notFull.signal(); // Signal that buffer is not full
        return item;
    } finally {
        lock.unlock();
    }
  }
}
```

The Producer-Consumer Problem

The producer-consumer problem is a classic example of multi-process synchronization. It describes two processes, the producer and the consumer, who share a common, fixed-size buffer. The producer's job is to generate data and put it into the buffer, while the consumer's job is to consume the data from the buffer.

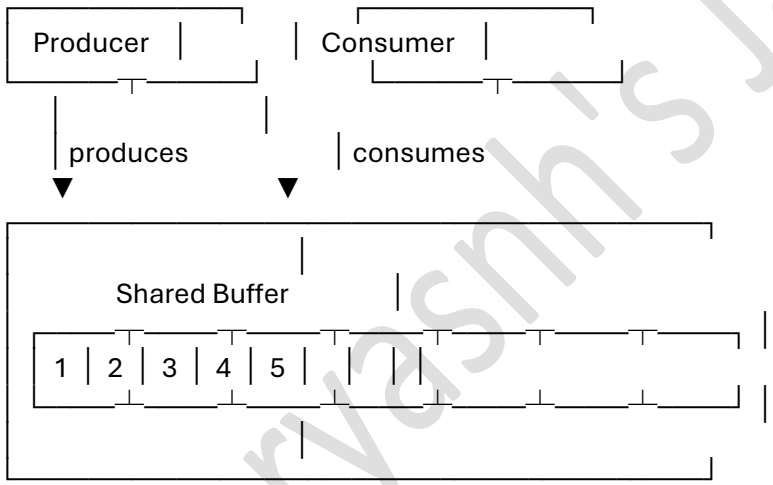
Using wait() and notify()

```
public class ProducerConsumerExample {
  private static final int BUFFER_SIZE = 5;
  private final Queue<Integer> buffer = new LinkedList<>();
  private final Object lock = new Object();

  class Producer implements Runnable {
    @Override
    public void run() {
        int value = 0;
        while (true) {
            synchronized (lock) {
                while (buffer.size() == BUFFER_SIZE) {
```

```
                try {
                    lock.wait(); // Buffer is full, wait
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                    return;
                }
            }

            System.out.println("Producing: " + value);
            buffer.add(value++);
            lock.notifyAll(); // Notify consumers
        }

        // Simulate some work
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            return;
        }
    }
  }
}

class Consumer implements Runnable {
  @Override
  public void run() {
    while (true) {
      synchronized (lock) {
        while (buffer.isEmpty()) {
          try {
            lock.wait(); // Buffer is empty, wait
          } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            return;
          }
        }

        int value = buffer.poll();
        System.out.println("Consuming: " + value);
        lock.notifyAll(); // Notify producers
      }

      // Simulate some work
      try {
        Thread.sleep(2000);
      } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
```

```
            return;
          }
        }
      }
    }

    public void start() {
      Thread producerThread = new Thread(new Producer());
      Thread consumerThread = new Thread(new Consumer());

      producerThread.start();
      consumerThread.start();
    }

    public static void main(String[] args) {
      new ProducerConsumerExample().start();
    }
}
```

Producer-Consumer Diagram



Using BlockingQueue

Java's BlockingQueue interface provides a thread-safe queue implementation that blocks when necessary.

```
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;

public class BlockingQueueExample {
    private static final int BUFFER_SIZE = 5;
    private final BlockingQueue<Integer> queue = new LinkedBlockingQueue<>(BUFFER_SIZE);

    class Producer implements Runnable {
      @Override
```

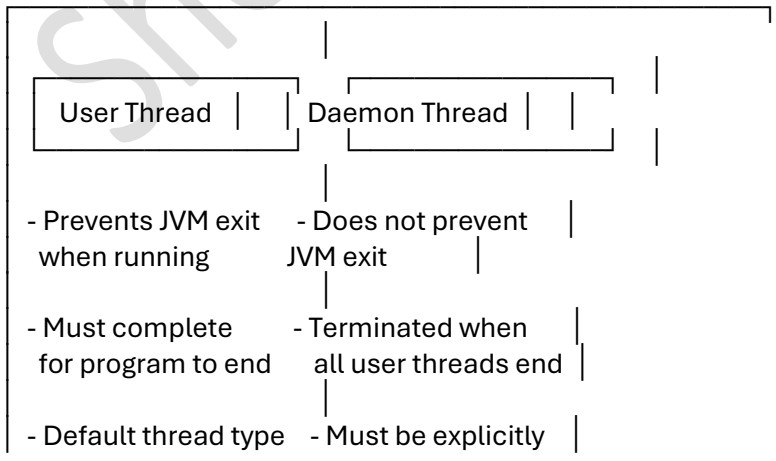
```
        public void run() {
            int value = 0;
            while (true) {
                try {
                    System.out.println("Producing: " + value);
                    queue.put(value++); // Blocks if queue is full
                    Thread.sleep(1000); // Simulate work
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                    return;
                }
            }
        }
    }

    class Consumer implements Runnable {
        @Override
        public void run() {
            while (true) {
                try {
                    int value = queue.take(); // Blocks if queue is empty
                    System.out.println("Consuming: " + value);
                    Thread.sleep(2000); // Simulate work
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                    return;
                }
            }
        }
    }

    public void start() {
        Thread producerThread = new Thread(new Producer());
        Thread consumerThread = new Thread(new Consumer());

        producerThread.start();
        consumerThread.start();
    }

    public static void main(String[] args) {
        new BlockingQueueExample().start();
    }
}
```

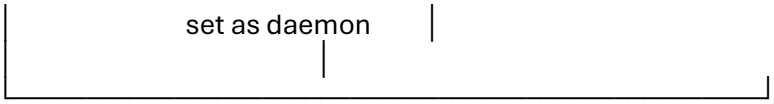
Daemon Threads

Daemon threads are background threads that do not prevent the JVM from exiting when the program finishes. They are typically used for background tasks like garbage collection or service threads that should run for the lifetime of the application.

```
public class DaemonThreadExample {
  public static void main(String[] args) {
    Thread daemonThread = new Thread(() -> {
      while (true) {
        System.out.println("Daemon thread is running...");
        try {
          Thread.sleep(1000);
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
      }
    });

    // Set as daemon thread
    daemonThread.setDaemon(true);
    daemonThread.start();

    // Main thread sleeps for 5 seconds
    try {
      Thread.sleep(5000);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }

    System.out.println("Main thread exiting, daemon thread will be terminated");
    // When main thread exits, the JVM will terminate, and the daemon thread will be terminated
  }
}
```

Daemon Thread vs User Thread





Thread Pools and the Executor Framework

For most applications, creating threads directly is not the best approach. The Executor framework provides a higher-level abstraction for thread management through thread pools.

Types of Thread Pools

Java's Executors class provides factory methods for creating different types of thread pools:

Fixed Thread Pool

```
ExecutorService executor = Executors.newFixedThreadPool(5);
```

A fixed thread pool creates a specified number of threads and reuses them for task execution. If all threads are busy, new tasks wait in a queue.

Cached Thread Pool

```
ExecutorService executor = Executors.newCachedThreadPool();
```

A cached thread pool creates new threads as needed and reuses existing idle threads. Threads that remain idle for 60 seconds are terminated.

Scheduled Thread Pool

```
ScheduledExecutorService executor = Executors.newScheduledThreadPool(3);
```

A scheduled thread pool can execute tasks after a delay or periodically.

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

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

    // Execute a task after 2 seconds delay
    executor.schedule(() -> {
      System.out.println("Delayed task executed");
    }, 2, TimeUnit.SECONDS);

    // Execute a task every 3 seconds, starting after 0 seconds
    executor.scheduleAtFixedRate(() -> {
      System.out.println("Periodic task executed at fixed rate");
    }, 0, 3, TimeUnit.SECONDS);
```

```
        // Execute a task every 3 seconds after the previous task completes
        executor.scheduleWithFixedDelay(() -> {
            System.out.println("Periodic task executed with fixed delay");
            try {
                Thread.sleep(1000); // Simulate work
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }, 0, 3, TimeUnit.SECONDS);

        // Let the tasks run for a while
        try {
            Thread.sleep(15000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

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

Single Thread Executor

```
ExecutorService executor = Executors.newSingleThreadExecutor();
```

A single thread executor uses a single worker thread to execute tasks sequentially.

Submitting Tasks to an Executor

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

public class ExecutorSubmitExample {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(3);

        // Submit a Runnable task
        executor.submit(() -> {
            System.out.println("Runnable task executed by " + Thread.currentThread().getName());
        });

        // Submit a Callable task that returns a result
        Future<String> future = executor.submit(() -> {
            System.out.println("Callable task executed by " + Thread.currentThread().getName());
            return "Task Result";
        });
```

```
        try {
            // Get the result of the Callable task
            String result = future.get();
            System.out.println("Task result: " + result);
        } catch (Exception e) {
            e.printStackTrace();
        }

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

Shutting Down an Executor

```
executor.shutdown(); // Allows previously submitted tasks to execute before terminating
// or
executor.shutdownNow(); // Attempts to stop all actively executing tasks and returns a list of tasks that were awaiting execution
```

Thread-Local Variables

ThreadLocal provides thread-local variables, which are variables that are local to each thread. Each thread has its own, independently initialized copy of the variable.

```
public class ThreadLocalExample {
    // ThreadLocal variable
    private static final ThreadLocal<Integer> threadLocalValue = ThreadLocal.withInitial(() -> 0);

    public static void main(String[] args) {
        // Create two threads
        Thread thread1 = new Thread(() -> {
            // Set thread-local value for this thread
            threadLocalValue.set(1);
            System.out.println("Thread 1: " + threadLocalValue.get());

            // Sleep to demonstrate that the value persists
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            // Value is still 1 for this thread
            System.out.println("Thread 1 (after sleep): " + threadLocalValue.get());
        });

        Thread thread2 = new Thread(() -> {
            // Sleep to ensure thread1 sets its value first
```

```
        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // This thread has its own copy of the variable, initialized to 0
        System.out.println("Thread 2 (initial): " + threadLocalValue.get());

        // Set a different value for this thread
        threadLocalValue.set(2);
        System.out.println("Thread 2 (after set): " + threadLocalValue.get());
    });

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

Concurrent Collections

Java provides thread-safe collection classes in the java.util.concurrent package.

ConcurrentHashMap

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

public class ConcurrentHashMapExample {
  public static void main(String[] args) {
    ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>();

    // Multiple threads can safely modify the map
    Thread thread1 = new Thread(() -> {
      for (int i = 0; i < 1000; i++) {
        map.put("Key-" + i, i);
      }
    });

    Thread thread2 = new Thread(() -> {
      for (int i = 1000; i < 2000; i++) {
        map.put("Key-" + i, i);
      }
    });

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