**Threading in Java**

1. Introduction to Threading:

Threading is a fundamental concept in Java that allows a program to execute multiple tasks concurrently. Threads are lightweight processes within a program, and they share the same memory space, which makes them suitable for multi-tasking and parallel processing.

2. Creating Threads:

In Java, there are two main ways to create threads:

a. Extending the Thread class:

You can create a new class that extends the `Thread` class and override the `run()` method to define the thread's behavior.

class MyThread extends Thread {

public void run() {

// Thread's behavior

}

}

public class ThreadCreationExample {

public static void main(String[] args) {

MyThread thread = new MyThread();

thread.start(); // Starts the thread's execution

}

}

```

b. Implementing the Runnable interface:

You can create a class that implements the `Runnable` interface and provide the thread's behavior in the `run()` method.

class MyRunnable implements Runnable {

public void run() {

// Thread's behavior

}

}

public class ThreadCreationExample {

public static void main(String[] args) {

MyRunnable runnable = new MyRunnable();

Thread thread = new Thread(runnable);

thread.start(); // Starts the thread's execution

}

}

```

3. Thread States:

Threads can be in various states:

- New: The thread is created but not yet started.

- Runnable: The thread is ready to run or is currently executing.

- Blocked/Waiting: The thread is waiting for a resource or condition to continue.

- Timed Waiting: The thread is waiting for a certain amount of time.

- Terminated: The thread has finished its execution.

4. Thread Synchronization:

Synchronization is important when multiple threads access shared resources to avoid data inconsistency and race conditions. The `synchronized` keyword and the `Lock` interface are used for synchronization.

Example using synchronized block:

```java

class Counter {

private int count = 0;

public synchronized void increment() {

count++;

}

}

```

5. Thread Communication:

Threads often need to communicate with each other. The `wait()`, `notify()`, and `notifyAll()` methods allow threads to coordinate their actions.

Example using wait and notify:

```java

class SharedResource {

private boolean isReady = false;

public synchronized void produce() throws InterruptedException {

while (isReady) {

wait();

}

// Produce data

isReady = true;

notify();

}

public synchronized void consume() throws InterruptedException {

while (!isReady) {

wait();

}

// Consume data

isReady = false;

notify();

}

}

```

6. Thread Pooling:

Creating threads can be resource-intensive. Thread pooling helps manage and reuse a pool of worker threads to improve performance and reduce overhead.

Example using ExecutorService:

```java

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class ThreadPoolExample {

public static void main(String[] args) {

ExecutorService executor = Executors.newFixedThreadPool(5);

for (int i = 0; i < 10; i++) {

Runnable task = new MyRunnable();

executor.execute(task);

}

executor.shutdown(); // Terminate the thread pool after tasks are done

}

}

```

7. Thread Safety and Immutable Objects:

Immutable objects are inherently thread-safe because their state cannot be changed once created.

Example of an immutable class:

```java

final class ImmutablePerson {

private final String name;

private final int age;

public ImmutablePerson(String name, int age) {

this.name = name;

this.age = age;

}

public String getName() {

return name;

}

public int getAge() {

return age;

}

}

```

8. Thread Local Variables:

Thread-local variables allow each thread to have its own copy of a variable.

Example using ThreadLocal:

```java

public class ThreadLocalExample {

private static ThreadLocal<Integer> threadLocal = ThreadLocal.withInitial(() -> 0);

public static void main(String[] args) {

threadLocal.set(42);

Runnable task = () -> {

int value = threadLocal.get();

System.out.println("Thread-local value: " + value);

};

Thread thread1 = new Thread(task);

Thread thread2 = new Thread(task);

thread1.start();

thread2.start();

}

}

```

These study notes provide an overview of threading concepts in Java, including creating threads, synchronization, communication, thread pooling, thread safety, immutable objects, and thread-local variables. Threading is a powerful tool, but it requires careful consideration and understanding to avoid potential issues like race conditions and deadlocks.