Multi-Threading in Java

Java Interview Essentials

Prateek Maheshwari
@friskycodeur

1

What is Multithreading?

Multithreading lets a program do multiple things at the same time.
Each task runs in its own **thread**, and this can help make programs faster and more responsive.

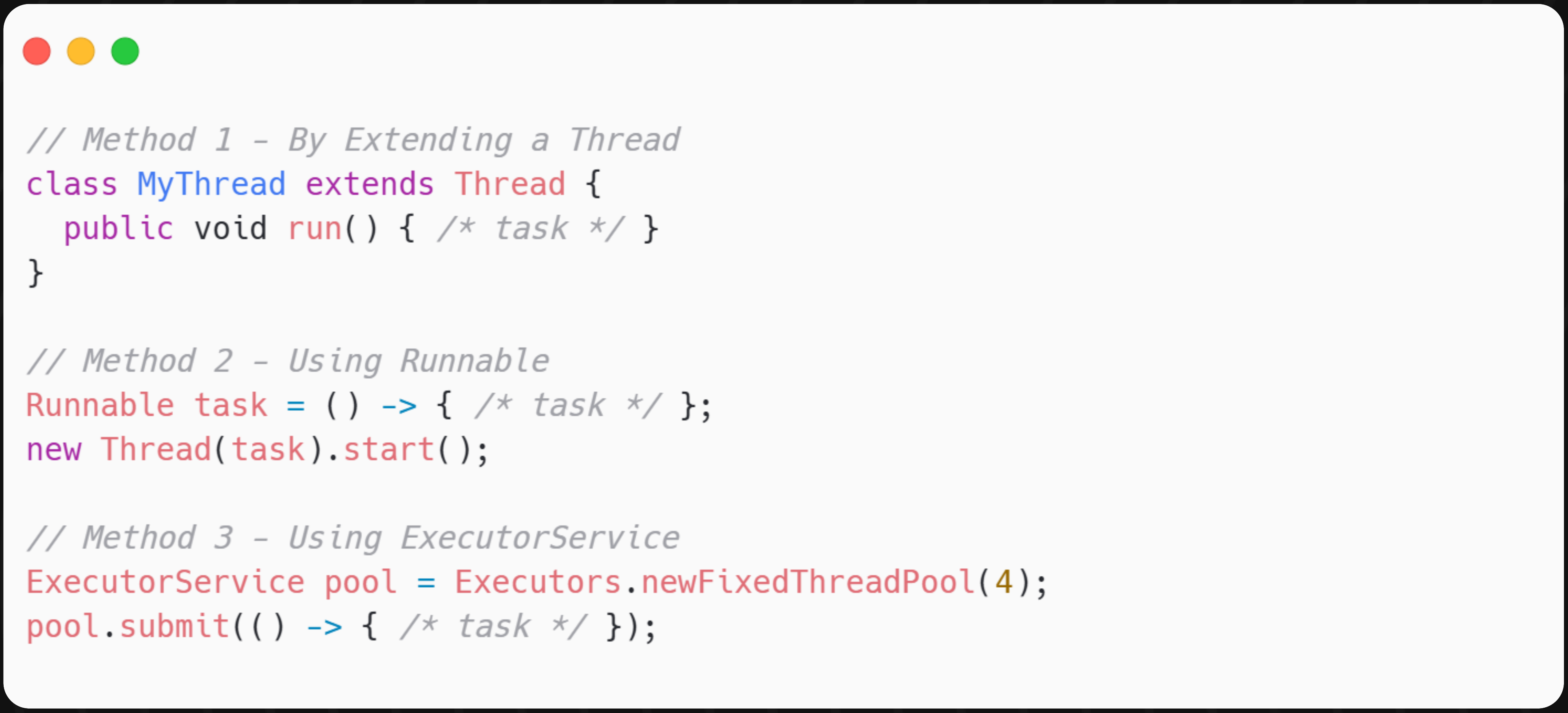
- Common use cases:
  - File processing
  - Background tasks
  - Handling multiple users on a web server

Prateek Maheshwari
@friskycodeur

2

How to Create Threads in Java

There are 3 common ways as descibed in the attached code snippet.
Note: Use thread pools (like ExecutorService) instead of starting threads manually. It’s cleaner and more efficient.



```java
// Method 1 - By Extending a Thread
class MyThread extends Thread {
  public void run() { /* task */ }
}

// Method 2 - Using Runnable
Runnable task = () -> { /* task */ };
new Thread(task).start();

// Method 3 - Using ExecutorService
ExecutorService pool = Executors.newFixedThreadPool(4);
pool.submit(() -> { /* task */ });
```

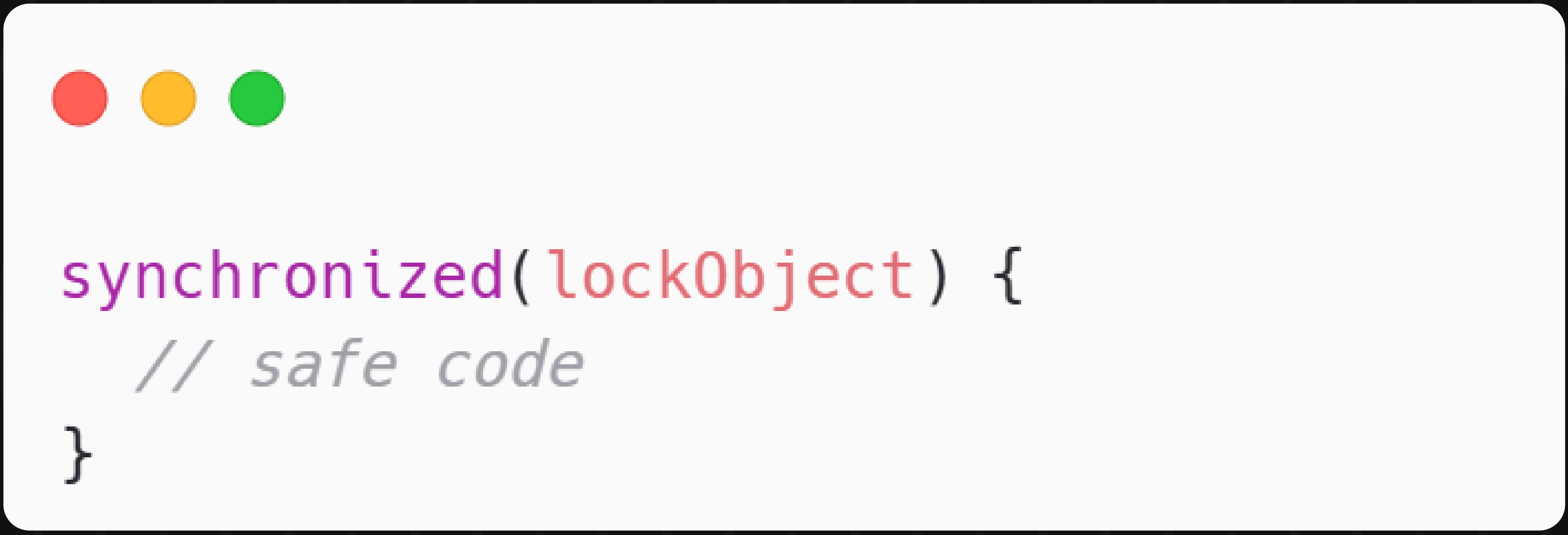
Prateek Maheshwari
@friskycodeur

3

How to Keep Your Code Safe

To keep things safe:

- Use the synchronized keyword to control access to shared data
- Use ConcurrentHashMap, CopyOnWriteArrayList, and other thread-safe classes
- Use AtomicInteger for counters and simple shared values
- Try to design your data to be immutable (unchanged after creation)



```
synchronized(lockObject) {
  // safe code
}
```

Prateek Maheshwari
@friskycodeur

4

Common Mistakes in Multithreading

Watch out for these issues:

- **Race condition** – two threads try to change the same data
- **Deadlock** – threads wait for each other and stop forever
- **Too much locking** – slows down the app
- **Unsynced memory** – changes made by one thread not visible to others

These bugs are not easy to find–so don't wait until production!

Prateek Maheshwari
@friskycodeur

5

Debugging Multithreading Issues

Thread issues can be tricky. Here’s how to make them easier to debug and solve:

- Use tools like VisualVM to inspect threads and detect deadlocks
- Avoid sharing data if you don’t have to
- Always lock shared resources in the same order
- Write simple code first—optimize later only when needed

Prateek Maheshwari
@friskycodeur

6

My Personal Go-To Best Practices

- Use ExecutorService instead of creating threads manually
- Keep shared data to a minimum
- Only lock what’s needed, and for a short time
- Use built-in thread-safe classes—don't try to build your own unless you need to
- Always log or handle exceptions in threads

Prateek Maheshwari
@friskycodeur

Share your experience or questions in comments!

Java interview topics every Tuesday & Thursday!

Follow @friskycodeur

Prateek Maheshwari

@friskycodeur