

Multithreading Basics

Assignment 8 – Theoretical Questions

Advanced Programming (AP)

Faraz Ardeh

fa.ardeh@gmail.com

Shahid Beheshti University

June 2026

Contents

1	start() vs run()	2
1.1	Expected output	2
1.2	Why this output?	2
1.3	Key differences at a glance	2
2	Daemon Threads	3
2.1	Expected output (with <code>setDaemon(true)</code>)	3
2.2	What happens if you remove <code>thread.setDaemon(true)</code> ?	4
2.3	Real-life use cases of daemon threads	4
3	A Shorter Way to Create Threads – Lambda Expressions	4
3.1	Expected output	4
3.2	What is <code>() -> { ... }</code> called?	5
3.3	How does this differ from other approaches?	5

Question 1. `start()` vs `run()`

The program under analysis:

```
1 public class StartVsRun {
2     static class MyRunnable implements Runnable {
3         public void run() {
4             System.out.println("Running in: " + Thread.
5                 currentThread().getName());
6         }
7     }
8     public static void main(String[] args) throws
9         InterruptedException {
10         Thread t1 = new Thread(new MyRunnable(), "Thread-1");
11         System.out.println("Calling run()");
12         t1.run();
13         Thread.sleep(100);
14
15         Thread t2 = new Thread(new MyRunnable(), "Thread-2");
16         System.out.println("Calling start()");
17         t2.start();
18     }
19 }
```

Listing 1: StartVsRun.java

Expected output

```
Calling run()
Running in: main
Calling start()
Running in: Thread-2
```

Why this output?

The `t1.run()` call: Calling `run()` on a `Thread` object is just an ordinary method call, exactly like calling any other method on any other object. No new OS thread is created or scheduled. The body of `run()` executes *synchronously* inside the calling thread, which is `main`. Therefore `Thread.currentThread().getName()` returns `"main"`, not `"Thread-1"`. The thread object `t1` was constructed but its native thread was never started.

The `t2.start()` call: `start()` allocates a new OS-level thread, registers it with the JVM scheduler, and returns to the caller *immediately* (it does not wait for `run()` to finish). The new thread then invokes `run()` autonomously and concurrently. Inside that new thread, `Thread.currentThread().getName()` correctly returns `"Thread-2"`.

Key differences at a glance

Aspect	run()	start()
New thread created?	No	Yes (exactly one)
Execution thread	Current thread	New thread
Blocking behaviour	Synchronous (caller waits)	Asynchronous (returns at once)
Can be called twice?	Yes, legal	No – throws <code>IllegalThreadStateException</code>
Useful for?	Testing run() logic directly	True concurrent execution

Summary: `run()` is simply a method call; `start()` is what actually creates and launches a new thread of execution. Calling `run()` instead of `start()` is one of the most common multithreading bugs in Java – the code compiles and "works", but no concurrency ever happens.

Question 2. Daemon Threads

```

1 public class DaemonExample {
2     static class DaemonRunnable implements Runnable {
3         public void run() {
4             for (int i = 0; i < 20; i++) {
5                 System.out.println("Daemon thread running...");
6                 try {
7                     Thread.sleep(500);
8                 } catch (InterruptedException e) {
9                     // [Handling Exception...]
10                }
11            }
12        }
13    }
14    public static void main(String[] args) {
15        Thread thread = new Thread(new DaemonRunnable());
16        thread.setDaemon(true);
17        thread.start();
18        System.out.println("Main thread ends.");
19    }
20 }

```

Listing 2: DaemonExample.java

Expected output (with `setDaemon(true)`)

```

Main thread ends.
Daemon thread running...
(program terminates -- usually 0-2 more lines, non-deterministic)

```

Why? The JVM shuts down when there are no *non-daemon* threads still alive. After `main()` prints "Main thread ends." and returns, the only surviving thread is the daemon thread. Because it is marked as a daemon, the JVM does *not* wait for it to finish – it exits immediately (or after one scheduling quantum), abruptly terminating the daemon thread. The daemon may manage to print zero, one, or a few lines before termination, depending entirely on OS thread scheduling; this is non-deterministic.

What happens if you remove `thread.setDaemon(true)`?

Without that call the thread becomes a regular *user thread* (the default). The JVM rule is: **wait for all user threads to finish before exiting**. The daemon thread (now a user thread) will sleep 500 ms per iteration and loop 20 times – the program will take approximately $20 \times 0.5 = 10$ seconds to terminate, and all 20 lines of "Daemon thread running..." will be printed in full before the JVM exits.

Real-life use cases of daemon threads

Daemon threads are ideal for background housekeeping work that should not prevent the application from exiting cleanly.

1. **Garbage Collection.** The JVM's garbage collector runs as a daemon thread; it must not keep the JVM alive on its own.
2. **Log flushing.** A background thread that periodically flushes buffered log entries to disk. If the application finishes, unsaved logs are an acceptable loss compared with hanging the process.
3. **Cache invalidation / expiry.** A thread that sweeps an in-memory cache and removes stale entries does not need to outlive the application.
4. **Heartbeat / health-check threads.** Service-mesh sidecars and microservices often send periodic pings to a service-discovery system. These should die with the service, not keep it alive.
5. **IDE background indexing.** IntelliJ IDEA's indexer and VS Code's language-server workers are typically daemon-like: if you close the IDE, you do not want to wait for them to finish indexing.

Question 3. A Shorter Way to Create Threads – Lambda Expressions

```
1 public class ThreadDemo {
2     public static void main(String[] args) {
3         Thread thread = new Thread(() -> {
4             System.out.println("Thread is running using a lambda!")
5             ;
6         });
7         thread.start();
8     }
9 }
```

Listing 3: ThreadDemo.java

Expected output

```
Thread is running using a lambda!
```

What is `() -> { ... }` called?

The `() -> { ... }` syntax is called a **lambda expression** (also known as an *anonymous function* or *arrow function* in other languages).

A lambda expression is a concise way to provide an implementation of a *functional interface* – any interface that declares exactly one abstract method. `Runnable` is a functional interface: its single abstract method is `void run()`. The lambda `() -> { ... }` supplies that implementation inline, without naming it.

How does this differ from other approaches?

There are three common ways to supply a `Runnable` to a `Thread`:

```

1 // 1. Class that extends Thread
2 class MyThread extends Thread {
3     public void run() { System.out.println("extends Thread"); }
4 }
5 new MyThread().start();
6
7 // 2. Named class that implements Runnable
8 class MyRunnable implements Runnable {
9     public void run() { System.out.println("implements Runnable"); }
10 }
11 new Thread(new MyRunnable()).start();
12
13 // 3. Lambda expression (shortest)
14 new Thread(() -> System.out.println("lambda")).start();

```

Listing 4: Three equivalent approaches

Approach	Boilerplate	Separate class?	Captures outer vars?	Extensible?
<code>extends Thread</code>	High	Yes (named)	No	Inherits <code>Thread</code>
<code>implements Runnable</code>	Medium	Yes (named)	No	Yes
Anonymous class	Medium	Yes (anon.)	Yes (eff. final)	Yes
Lambda	Minimal	No	Yes (eff. final)	No

Key advantages of lambdas:

- **Conciseness.** No need to declare a class, override a method, or write a constructor. A single line can create and start a thread.
- **Readability.** The intent (“run this block of code in a new thread”) is immediately obvious to the reader.
- **Variable capture.** A lambda can reference local variables from the enclosing scope as long as they are effectively final (i.e., never reassigned after declaration). This is often more convenient than passing data through a constructor.
- **Functional programming patterns.** Lambdas work seamlessly with the Java Streams API, `CompletableFuture`, `ExecutorService`, and other modern concurrency utilities that accept `Runnable` or `Callable` arguments.

Limitation: Because a lambda is not a subclass of `Thread`, it cannot override thread life-cycle methods such as `interrupt()` or `setUncaughtExceptionHandler()` directly inside the lambda body. When that level of control is needed, a named class is more appropriate.