Threads Intro

CPSC 1181 – O.O.

Jeremy Hilliker

Summer 2017

Langara.
THE COLLEGE OF HIGHER LEARNING.

Overview

- Process
- Thread

- Creating Threads
- Working with Threads
- Executors

Processes

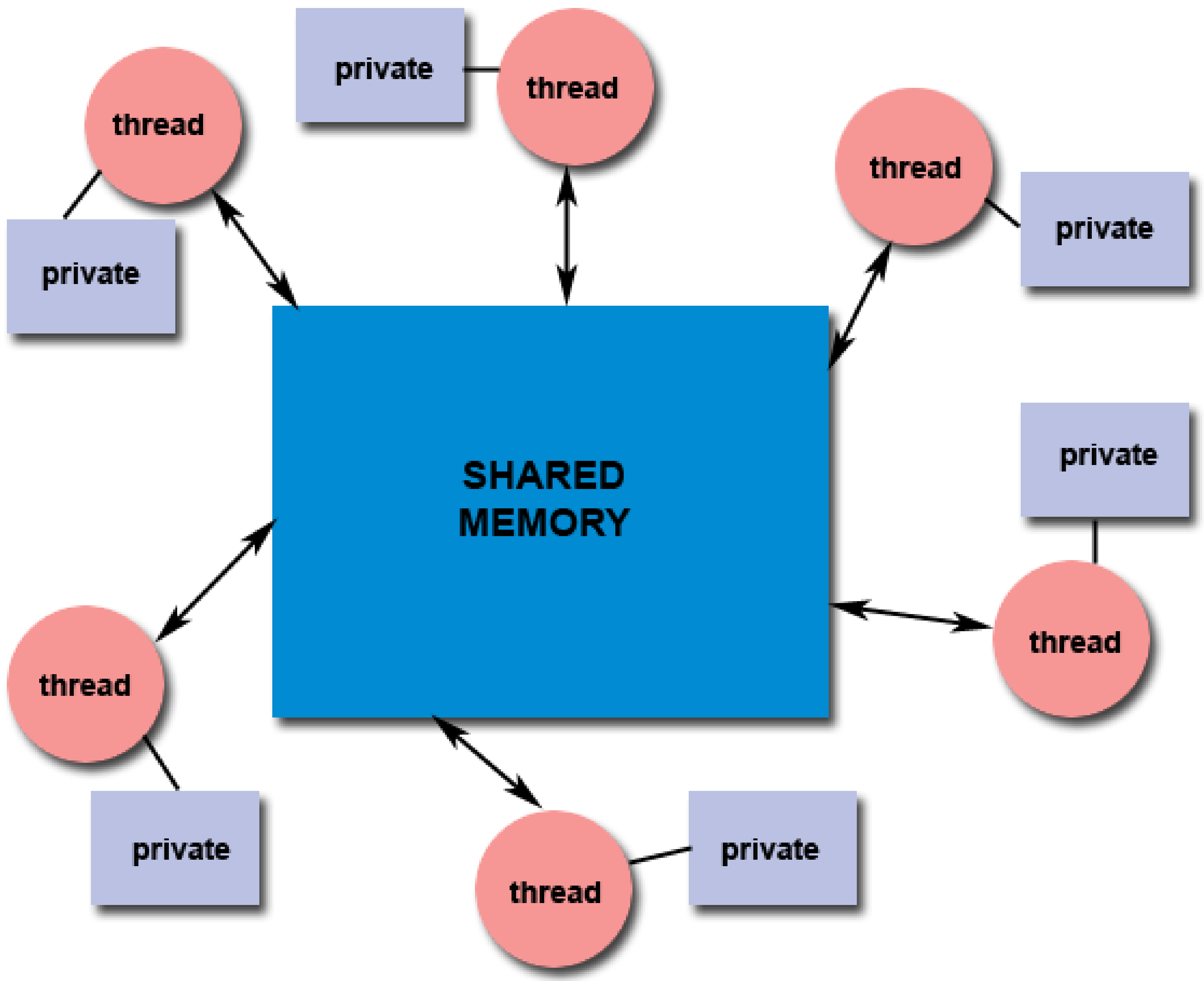
- An instance of a program
  - Like class & object
- Process:
  - Code
    - The program itself
  - Data (variables)
    - Stack
    - Heap
  - Resources
    - Files
    - Sockets
  - Threads

So Far…

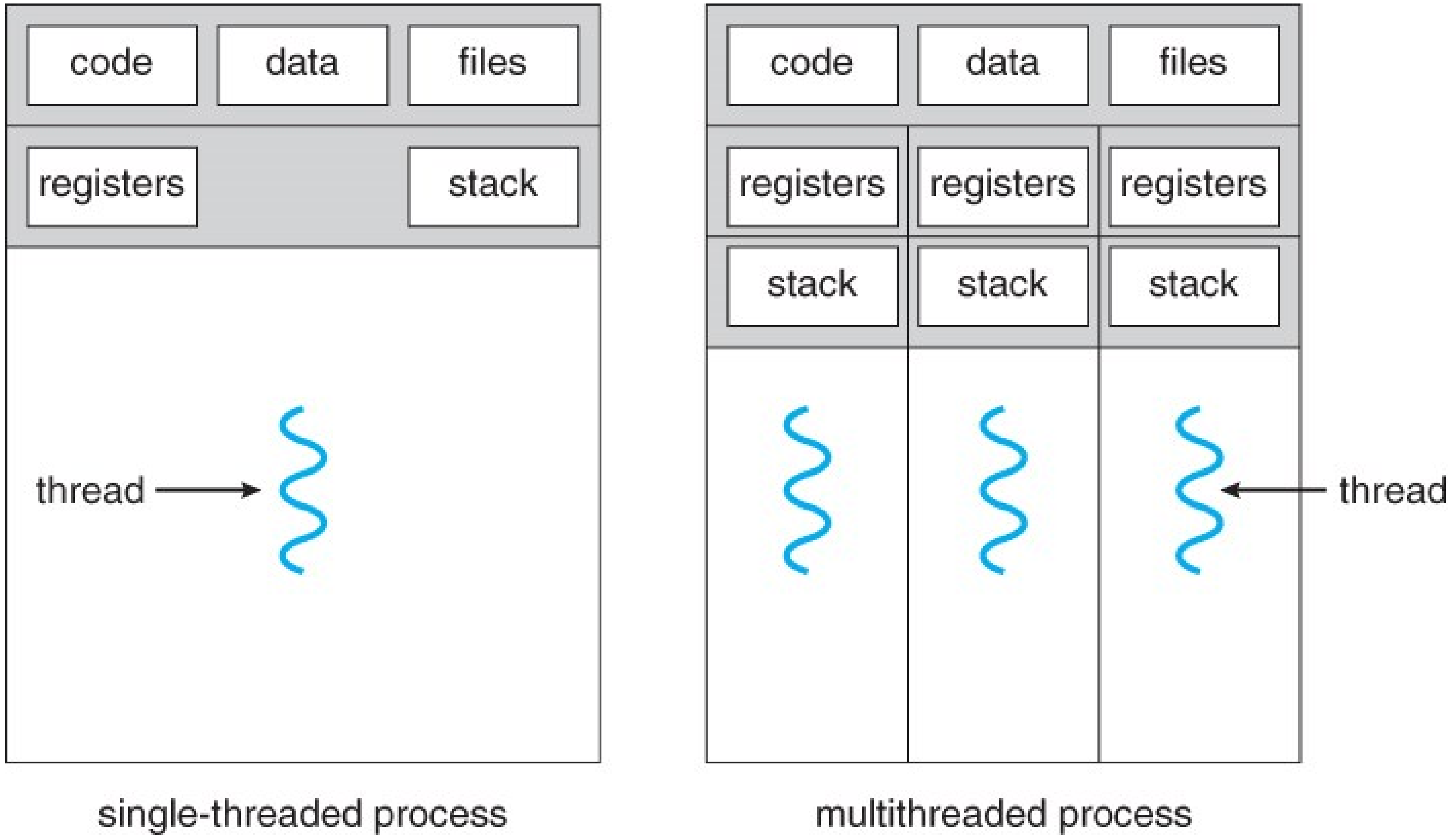
- Our processes have only had one thread of execution.
  - The main thread
- GUI programs:
  - Main thread
  - GUI thread: EDT (Event Dispatch Thread)
- Timers
  - Swing timers: execute on the EDT
  - Util timers: have their own thread

What is a thread

- A thread is:
  - **A schedulable unit**
  - **A “stream” of execution**
- Each thread has:
  - A PC (Program counter: what part of the code it is executing)
  - Registers*
  - A stack (local / temporary variables)
    - Some special resources (known as ThreadLocal)
- A thread shares:
  - Program code
  - The heap
    - In Java, all objects are on the heap
  - Process resources



SHARED MEMORY
thread
private
thread
private
thread
private
thread
private
thread
private
thread
private

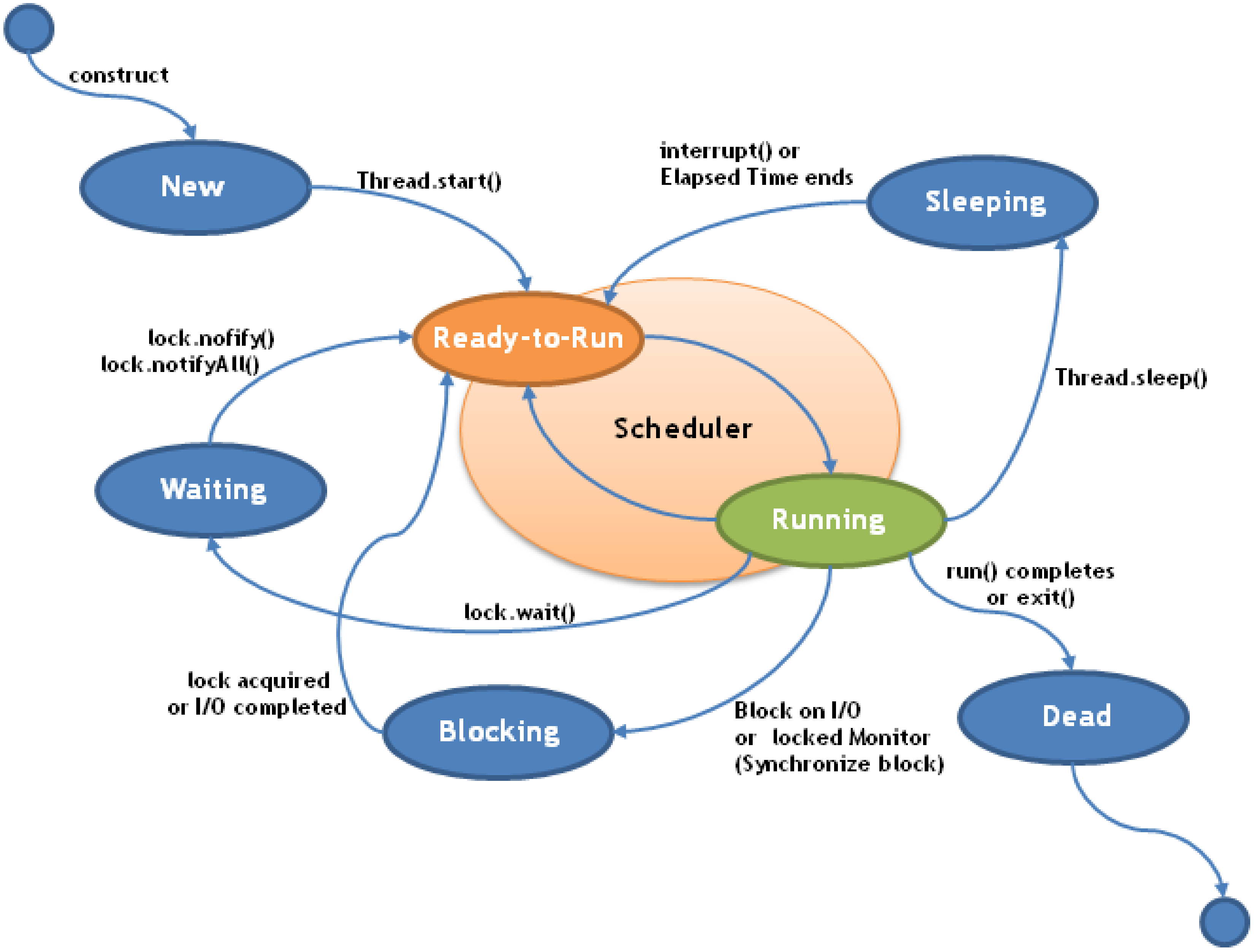

code
data
files
registers
stack
thread
single-threaded process
code
data
files
registers
registers
registers
stack
stack
stack
thread
multithreaded process

Why threads?

- Allows for parallel execution of code
- Resource utilization:
  - Take advantage of multiple CPUs
  - Take advantage of multiple cores
  - See: map-reduce
- Multitasking:
  - Do other things while another thread is blocked or waiting (timers, disk/network I/O, GUI)
  - Do other things while another thread is busy SwingWorker / GUI

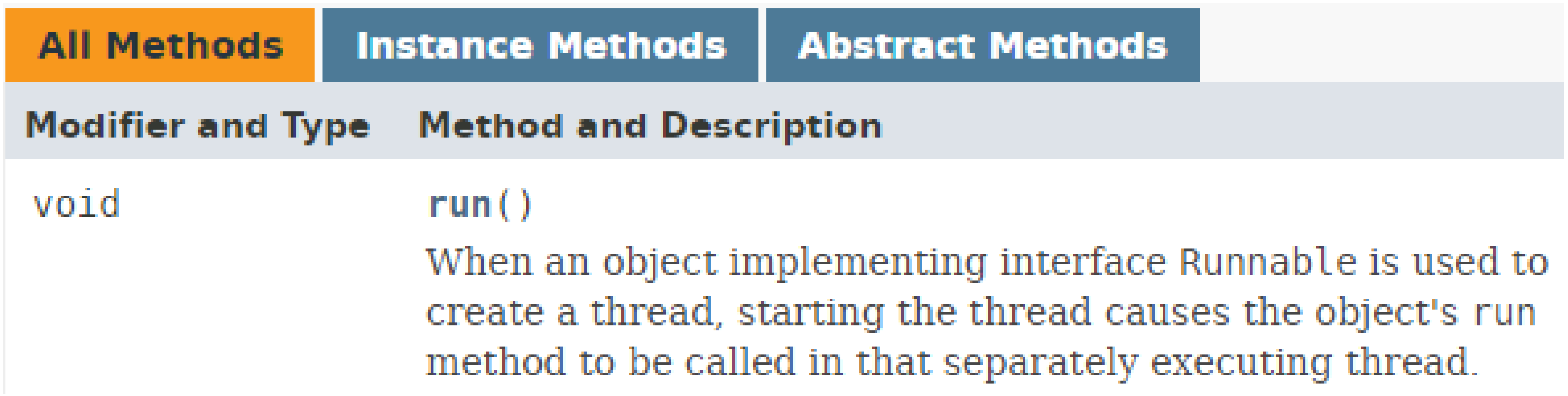
How it works

- The OS schedules each thread
  - Grants each ***ready*** thread a ***time-slice*** (or: ***slice***) on a core/CPU
  - The thread runs on that core until:
    - It is **pre-empted** by the OS/scheduler
    - It **yields** its slice
    - It **blocks** on I/O
    - It **sleeps** on a timer
    - It **waits** on a signal
  - In all of the above cases, the scheduler chooses another thread to run.
    - If none, idle until some event occurs


construct
New
Thread.start()
interrupt() or
Elapsed Time ends
Sleeping
lock.nofify()
lock.notifyAll()
Ready-to-Run
Scheduler
Thread.sleep()
Waiting
Running
lock.wait()
run() completes
or exit()
lock acquired
or I/O completed
Blocking
Block on I/O
or locked Monitor
(Synchronize block)
Dead

How to Create a Thread: Thread Object

1. Implement Runnable

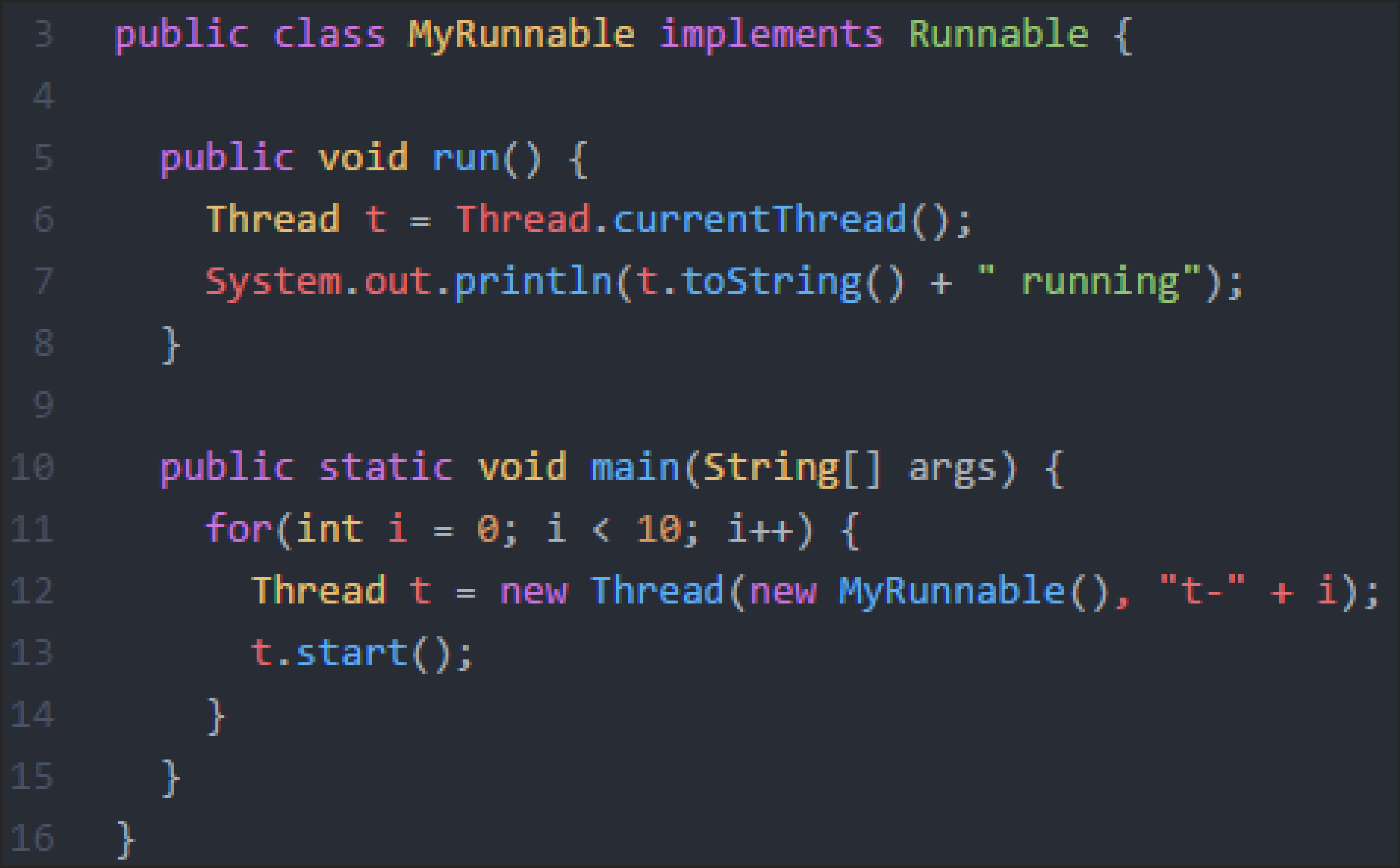


All Methods	Instance Methods	Abstract Methods

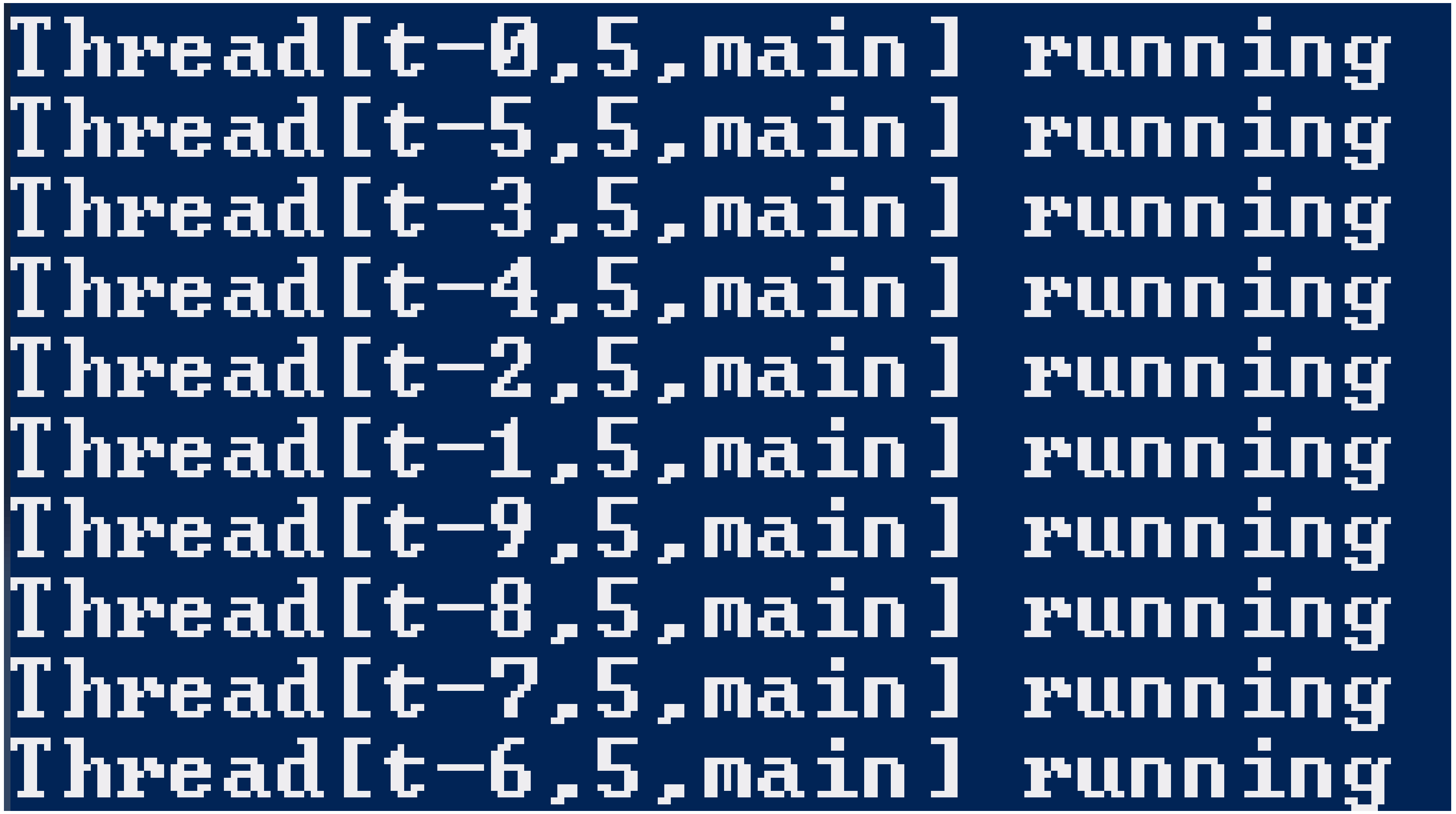
Modifier and Type	Method and Description
`void`	`run()` When an object implementing interface `Runnable` is used to create a thread, starting the thread causes the object's `run` method to be called in that separately executing thread.

How to Create a Thread: Thread Object

1. Implement Runnable
```
Class MyRunnable implements Runnable {
   public void run() {}
}
```
2. Create a Thread object with that runnable
```
   Thread t = new Thread(runnable);
```
3. Invoke Thread.start()
```
   t.start();
```
   - Do not invoke Thread.run() !

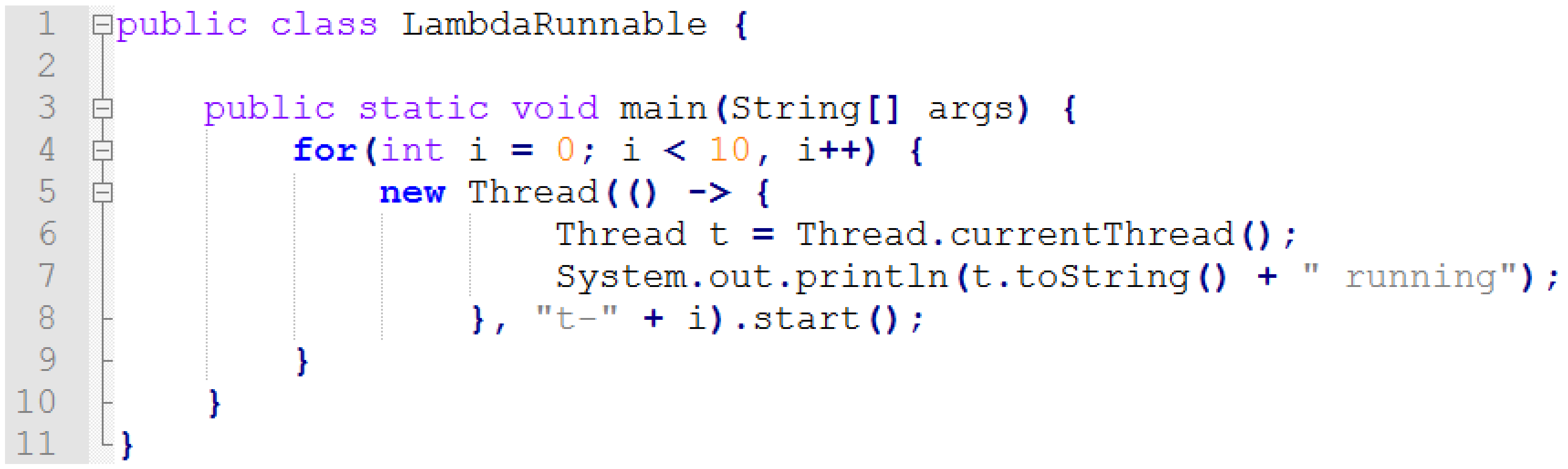


```java
public class MyRunnable implements Runnable {

  public void run() {
    Thread t = Thread.currentThread();
    System.out.println(t.toString() + " running");
  }

  public static void main(String[] args) {
    for(int i = 0; i < 10; i++) {
      Thread t = new Thread(new MyRunnable(), "t-" + i);
      t.start();
    }
  }
}
```



```
Thread[t-0,5,main] running
Thread[t-5,5,main] running
Thread[t-3,5,main] running
Thread[t-4,5,main] running
Thread[t-2,5,main] running
Thread[t-1,5,main] running
Thread[t-9,5,main] running
Thread[t-8,5,main] running
Thread[t-7,5,main] running
Thread[t-6,5,main] running
```

Or using lambda



```java
public class LambdaRunnable {

    public static void main(String[] args) {
        for(int i = 0; i < 10, i++) {
            new Thread(() -> {
                Thread t = Thread.currentThread();
                System.out.println(t.toString() + " running");
            }, "t-" + i).start();
        }
    }
}
```

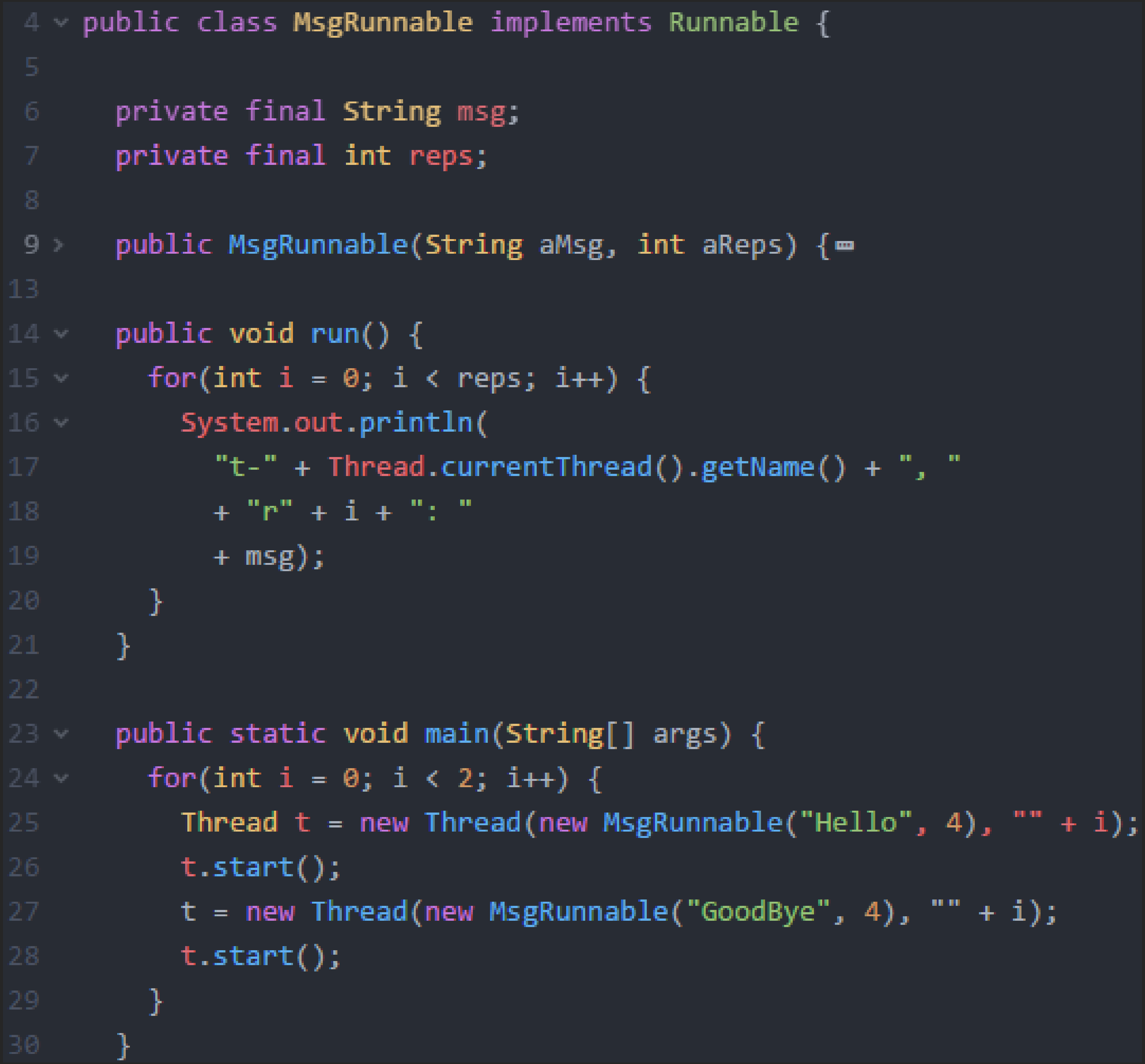
Passing Arguments

- Thread.run() takes no arguments
- How can we pass a thread arguments?

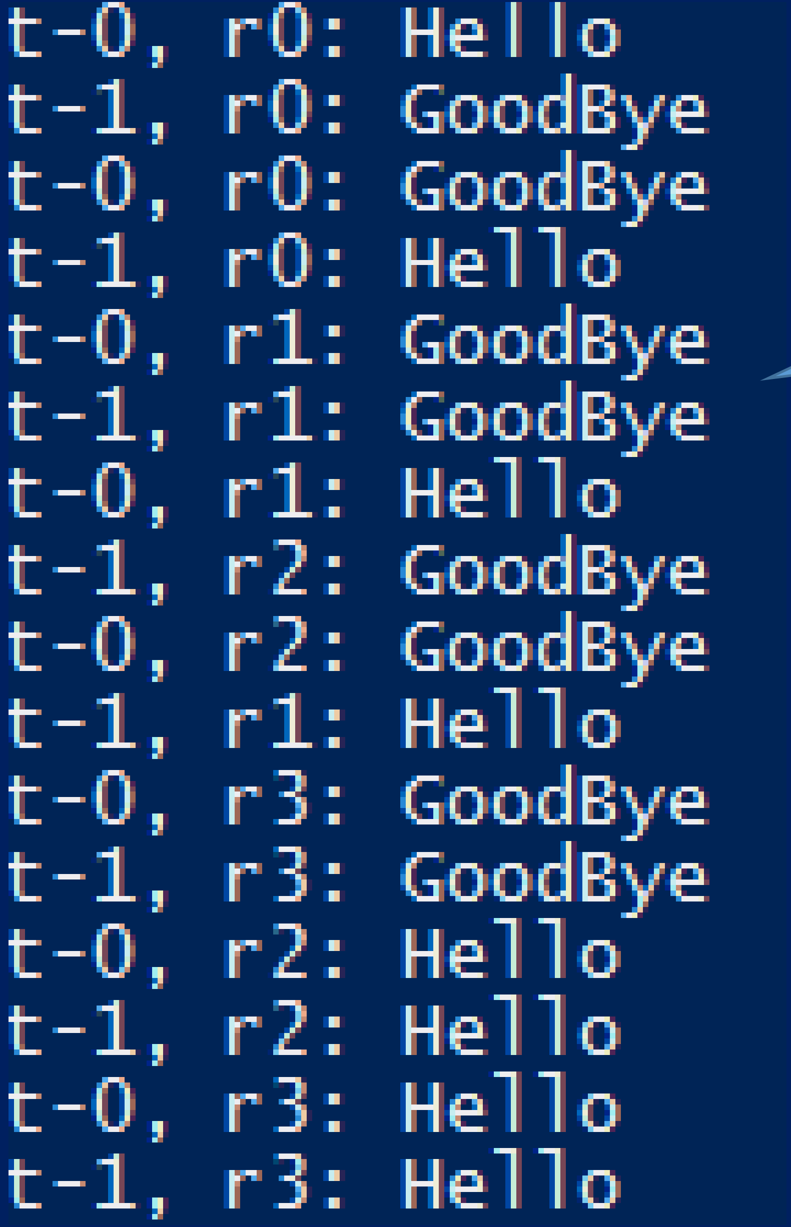
Passing Arguments

- Thread.run() takes no arguments
- How can we pass a thread arguments?

- Pass them through the constructor of the Runnable



```java
public class MsgRunnable implements Runnable {

  private final String msg;
  private final int reps;

  public MsgRunnable(String aMsg, int aReps) {

  public void run() {
    for(int i = 0; i < reps; i++) {
      System.out.println(
        "t-" + Thread.currentThread().getName() + ", "
        + "r" + i + ": "
        + msg);
    }
  }

  public static void main(String[] args) {
    for(int i = 0; i < 2; i++) {
      Thread t = new Thread(new MsgRunnable("Hello", 4), "" + i);
      t.start();
      t = new Thread(new MsgRunnable("GoodBye", 4), "" + i);
      t.start();
    }
  }
```



```
t-0, r0: Hello
t-1, r0: GoodBye
t-0, r0: GoodBye
t-1, r0: Hello
t-0, r1: GoodBye
t-1, r1: GoodBye
t-0, r1: Hello
t-1, r2: GoodBye
t-0, r2: GoodBye
t-1, r1: Hello
t-0, r3: GoodBye
t-1, r3: GoodBye
t-0, r2: Hello
t-1, r2: Hello
t-0, r3: Hello
t-1, r3: Hello
```

Note: the “GoodBye”s finished before the “Hello”s

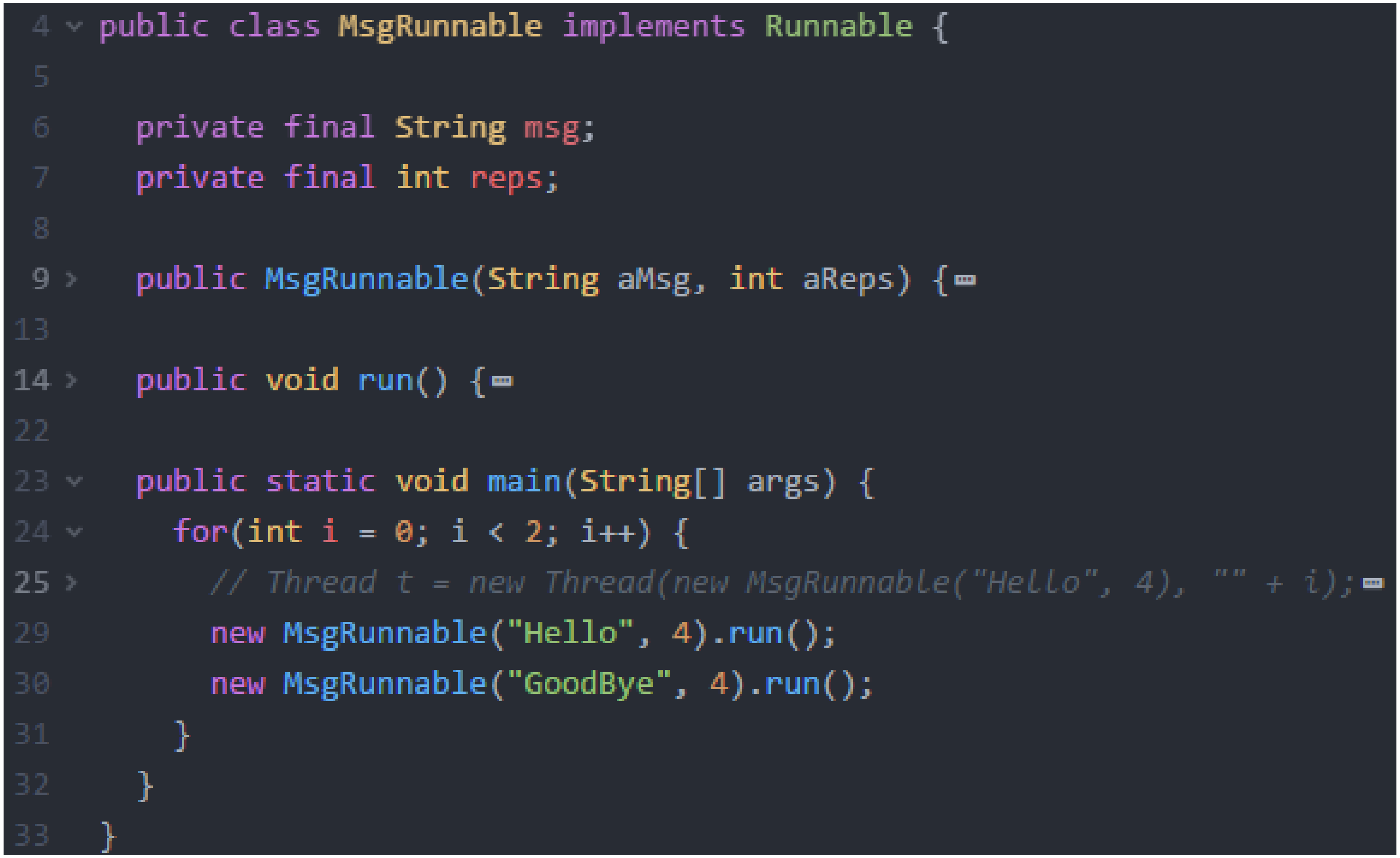
The scheduler can seem random

Scheduler

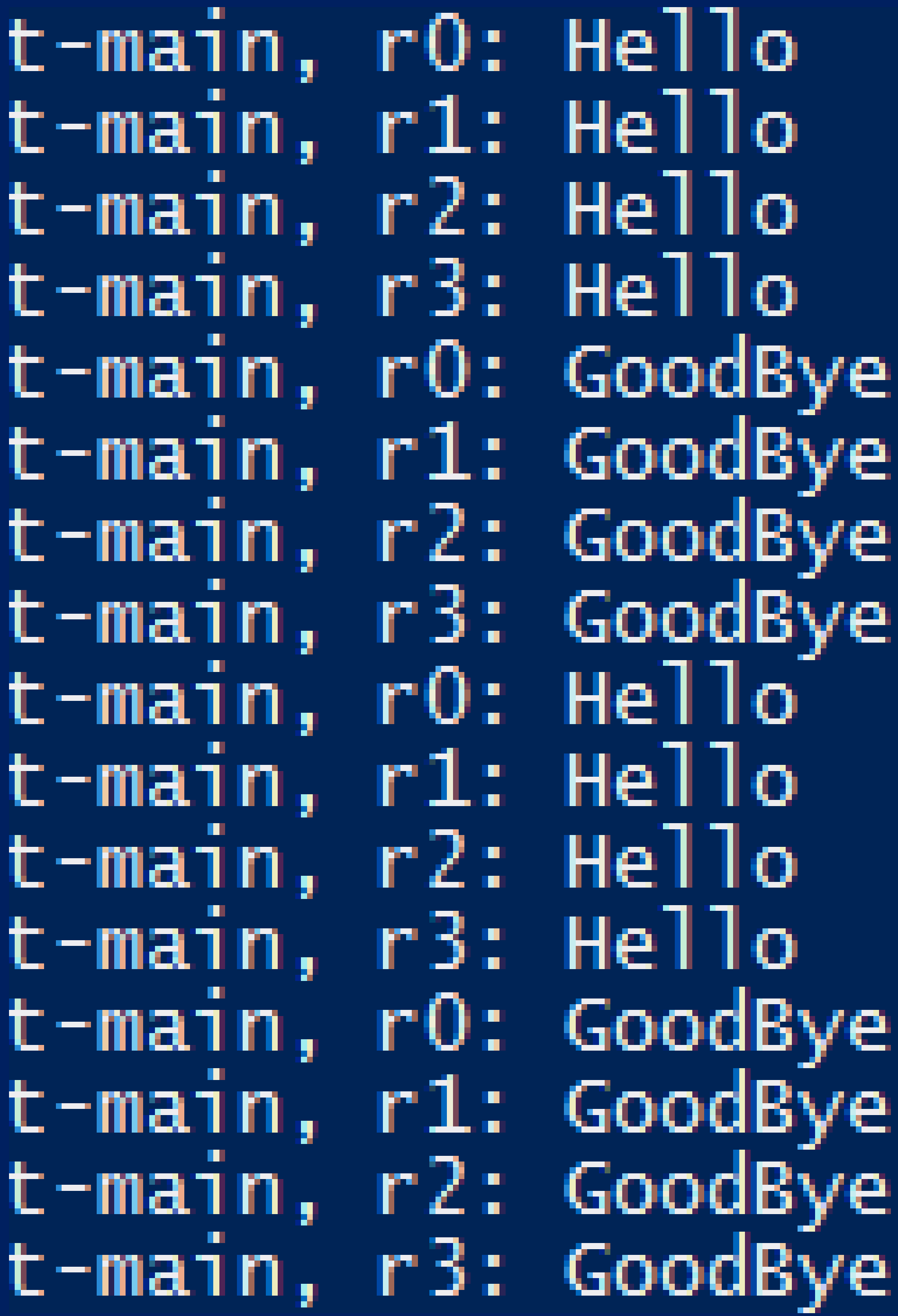
- For all intents and purposes:
- The scheduler is ***non-deterministic***

- There is ***no guarantee*** about the order in which threads are executed

Check:

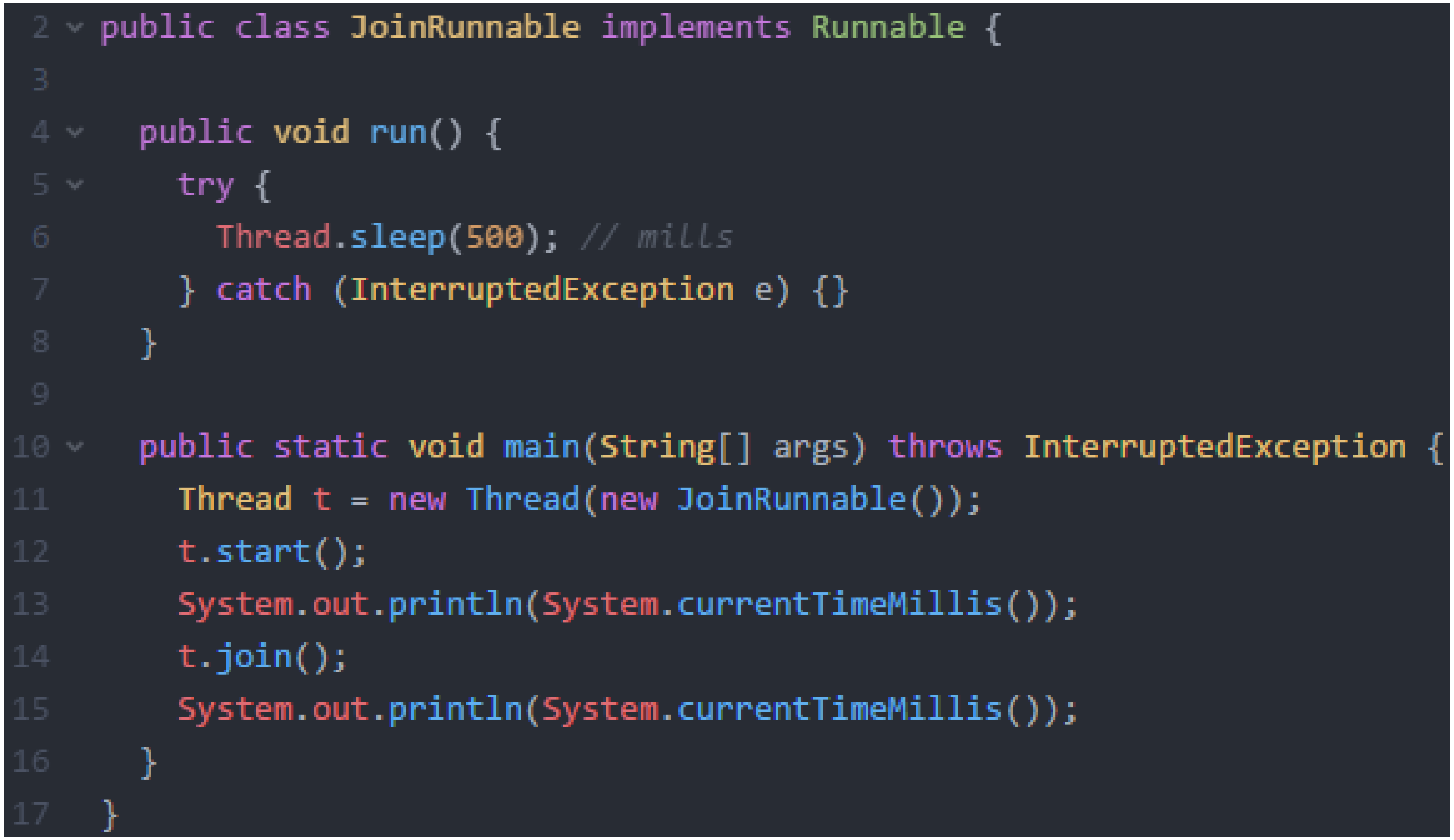


```java
public class MsgRunnable implements Runnable {

  private final String msg;
  private final int reps;

  public MsgRunnable(String aMsg, int aReps) {⋯

  public void run() {⋯

  public static void main(String[] args) {
    for(int i = 0; i < 2; i++) {
      // Thread t = new Thread(new MsgRunnable("Hello", 4), "" + i);⋯
      new MsgRunnable("Hello", 4).run();
      new MsgRunnable("GoodBye", 4).run();
    }
  }
}
```



```
t-main, r0: Hello
t-main, r1: Hello
t-main, r2: Hello
t-main, r3: Hello
t-main, r0: GoodBye
t-main, r1: GoodBye
t-main, r2: GoodBye
t-main, r3: GoodBye
t-main, r0: Hello
t-main, r1: Hello
t-main, r2: Hello
t-main, r3: Hello
t-main, r0: GoodBye
t-main, r1: GoodBye
t-main, r2: GoodBye
t-main, r3: GoodBye
```

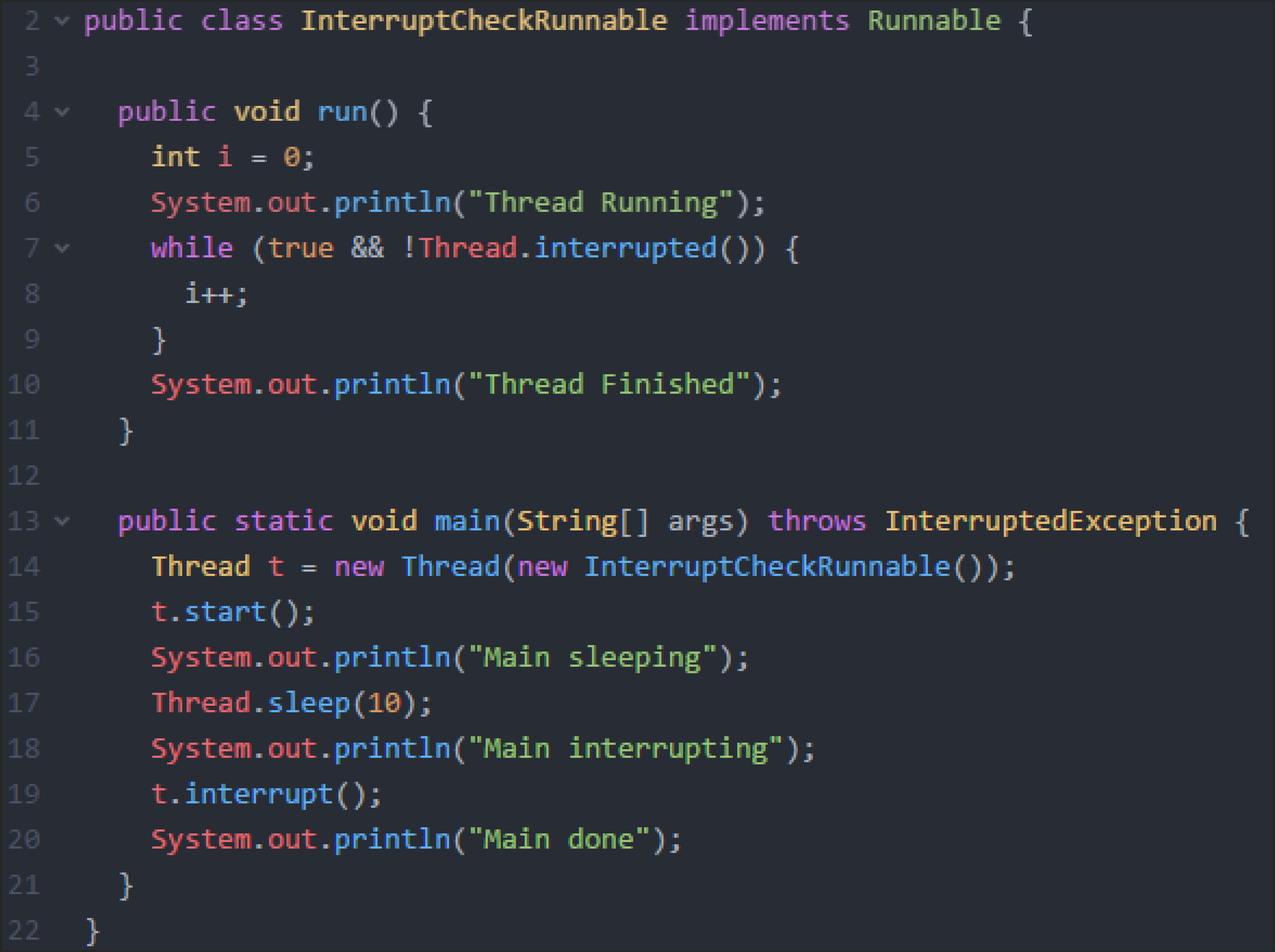
Waiting for a Thread



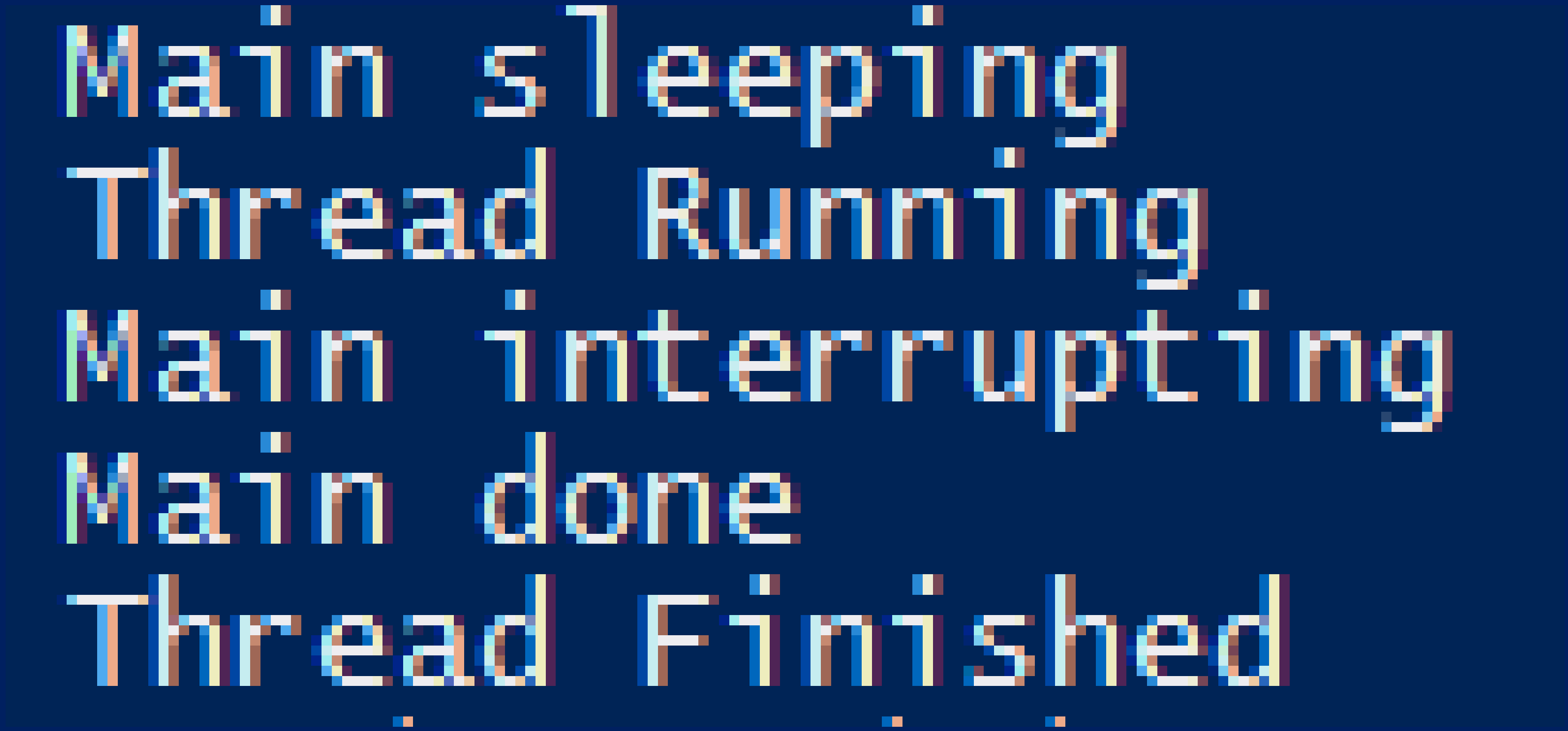
```
public class JoinRunnable implements Runnable {

  public void run() {
    try {
      Thread.sleep(500); // mills
    } catch (InterruptedException e) {}
  }

  public static void main(String[] args) throws InterruptedException {
    Thread t = new Thread(new JoinRunnable());
    t.start();
    System.out.println(System.currentTimeMillis());
    t.join();
    System.out.println(System.currentTimeMillis());
  }
}
```

How to Interrupt

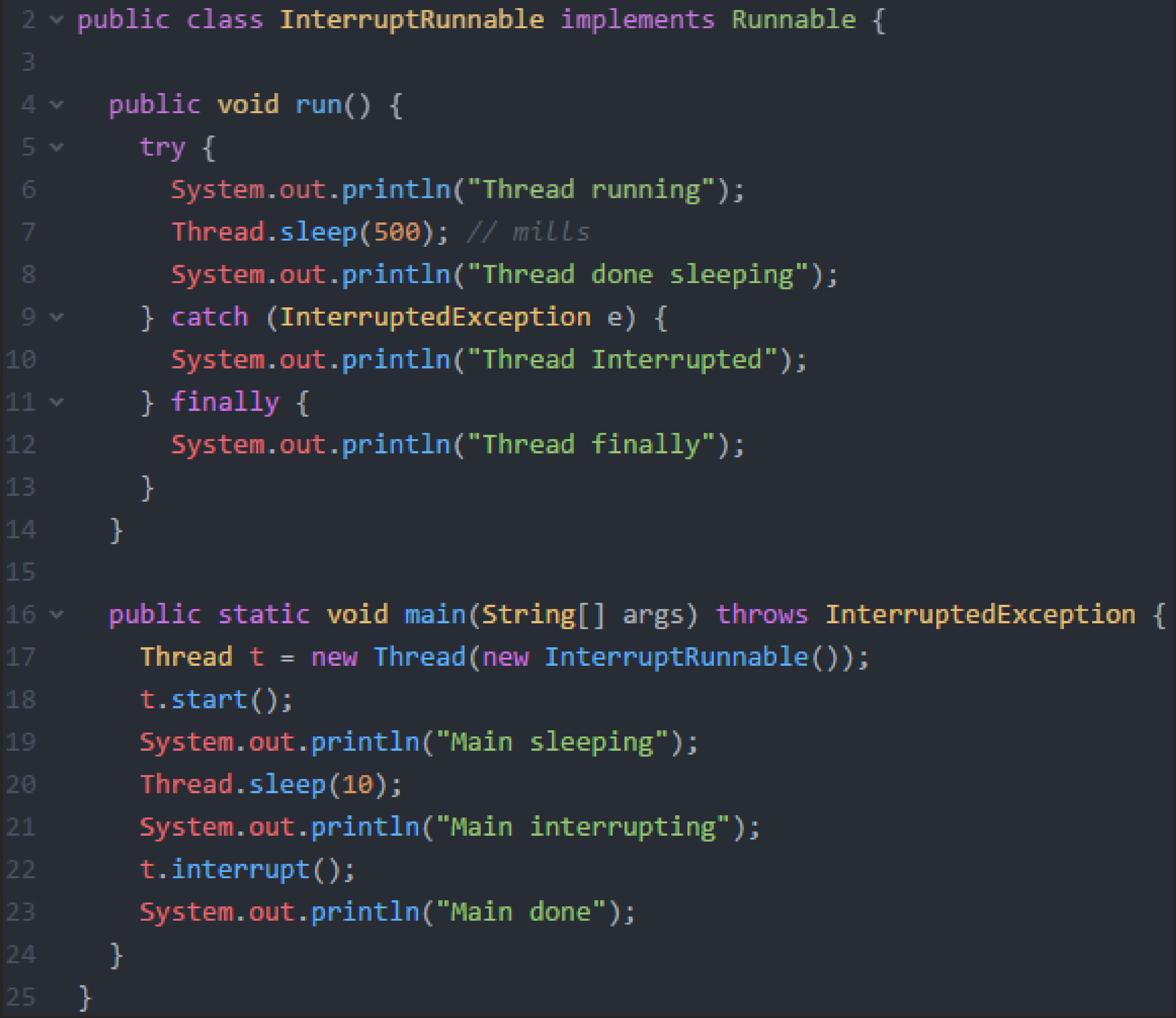
- Don’t call Thread.stop() !
  - Abruptly ends the thread, may leave program in an inconsistent state
  - https://docs.oracle.com/javase/8/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html
- Call Thread.interrupt()
  - Does not kill thread
  - Notifies thread that it should terminate
  - Thread is responsible for response
    - Can ignore
- .interrupt() is a general mechanism to get a thread’s attention



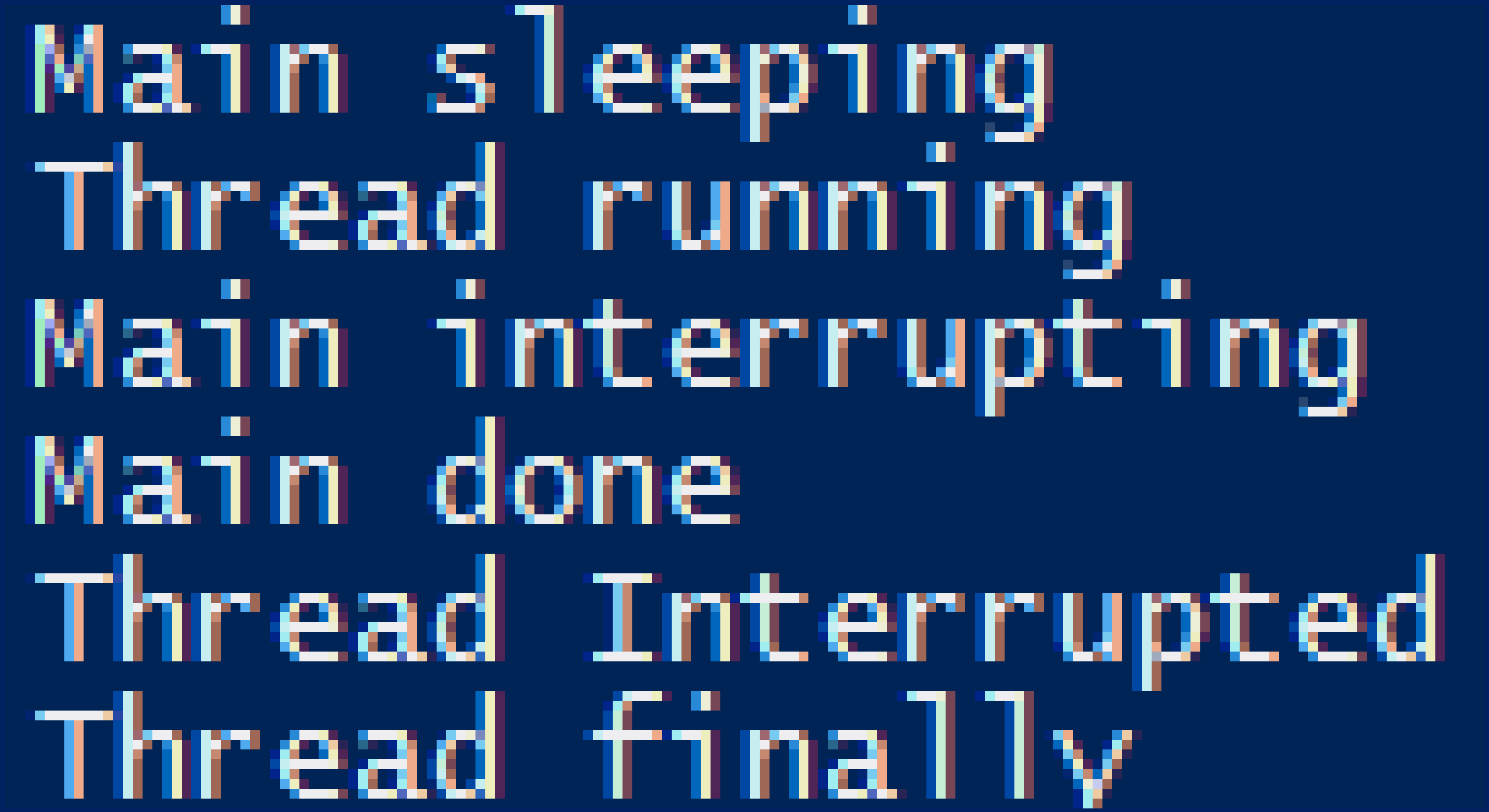
```java
public class InterruptCheckRunnable implements Runnable {

  public void run() {
    int i = 0;
    System.out.println("Thread Running");
    while (true && !Thread.interrupted()) {
      i++;
    }
    System.out.println("Thread Finished");
  }

  public static void main(String[] args) throws InterruptedException {
    Thread t = new Thread(new InterruptCheckRunnable());
    t.start();
    System.out.println("Main sleeping");
    Thread.sleep(10);
    System.out.println("Main interrupting");
    t.interrupt();
    System.out.println("Main done");
  }
}
```



```
Main sleeping
Thread Running
Main interrupting
Main done
Thread Finished
```



```java
public class InterruptRunnable implements Runnable {

  public void run() {
    try {
      System.out.println("Thread running");
      Thread.sleep(500); // mills
      System.out.println("Thread done sleeping");
    } catch (InterruptedException e) {
      System.out.println("Thread Interrupted");
    } finally {
      System.out.println("Thread finally");
    }
  }

  public static void main(String[] args) throws InterruptedException {
    Thread t = new Thread(new InterruptRunnable());
    t.start();
    System.out.println("Main sleeping");
    Thread.sleep(10);
    System.out.println("Main interrupting");
    t.interrupt();
    System.out.println("Main done");
  }
}
```



```
Main sleeping
Thread running
Main interrupting
Main done
Thread Interrupted
Thread finally
```

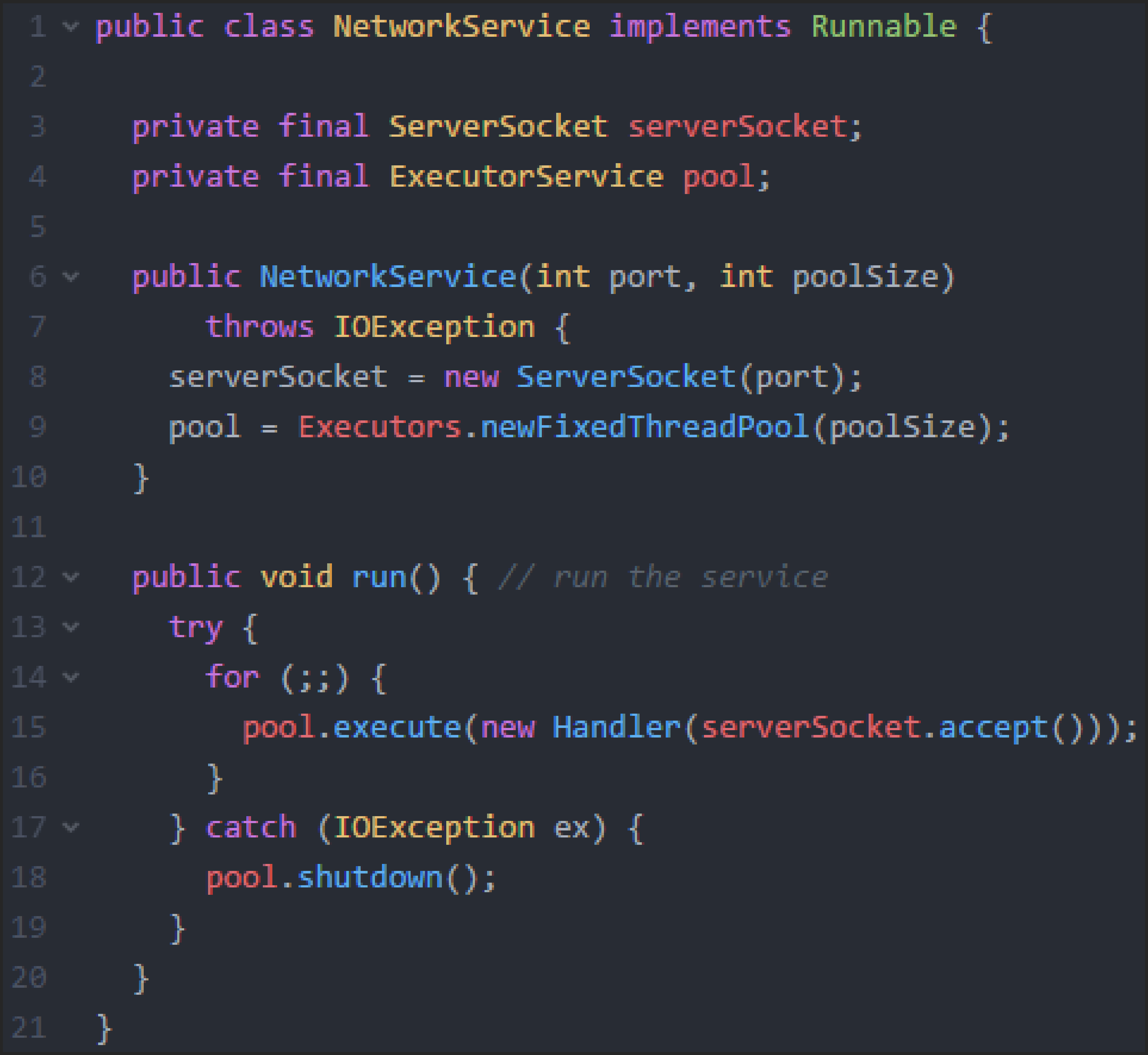
ExecutorService

- Executors manage threads for you
- Can achieve efficiencies
    - Thread creation is fairly expensive
    - Executor can manage a pool of threads
    - ExecutorService exec = Executors.
        - newCachedThreadPool()
        - newFixedThreadPool(int nThreads)
        - newSingleThreadExecutor()
        - newWorkStealingPool() // for many short tasks
        - newWorkStealingPool(int parallelism)
    - Also ExecutorService s for Scheduled Execution

ExecutorService

- void execute(Runnable command)
- <T> Future<T> submit(Callable<T> task)
- <T> Future<T> submit(Runnable task, T result)
- Future<?> submit(Runnable task)

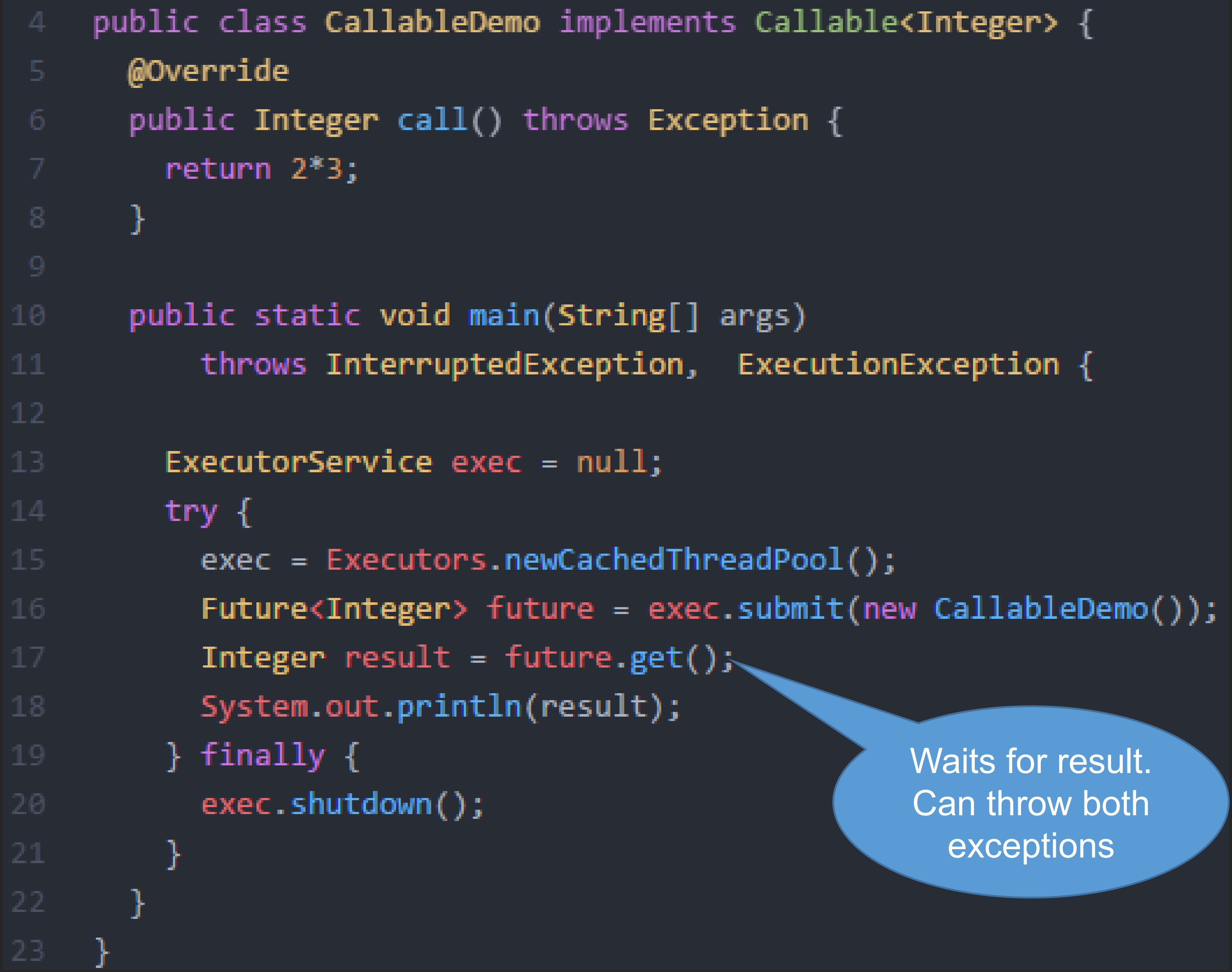
- awaitTermination, invokeAll, invokeAny, shutdown



```java
public class NetworkService implements Runnable {

    private final ServerSocket serverSocket;
    private final ExecutorService pool;

    public NetworkService(int port, int poolSize)
        throws IOException {
      serverSocket = new ServerSocket(port);
      pool = Executors.newFixedThreadPool(poolSize);
    }

    public void run() { // run the service
      try {
        for (;;) {
          pool.execute(new Handler(serverSocket.accept()));
        }
      } catch (IOException ex) {
        pool.shutdown();
      }
    }
}
```

Callable<V>

- V call() throws Exception



```java
public class CallableDemo implements Callable<Integer> {
  @Override
  public Integer call() throws Exception {
    return 2*3;
  }

  public static void main(String[] args)
      throws InterruptedException,  ExecutionException {

    ExecutorService exec = null;
    try {
      exec = Executors.newCachedThreadPool();
      Future<Integer> future = exec.submit(new CallableDemo());
      Integer result = future.get();
      System.out.println(result);
    } finally {
      exec.shutdown();
    }
  }
}
```

Waits for result. Can throw both exceptions

Streams

- Streams can run in parallel by default.
- Stream.parallel() to suggest it…

Advanced:

- ForkJoinPool & ForkJoinTask
  - For work that can be split and merged
  - Works sort of like a parallel recursive function

Recap

- Process
- Thread

- Creating Threads
  - Runnable
  - Thread.start()

- Working with Threads
  - Scheduler
  - Thread.join()
  - Thread.interrupt()

- Executors
  - execute()
  - submit()
- Callable<V>
  - Future<V>