B.Sc. In Software Development. Year 3. Applications Programming. Multithreading and Concurrency.

LIMERICK INSTITUTE OF TECHNOLOGY
SCHOOL OF SCIENCE, ENGINEERING & I.T.
Department of Information Technology

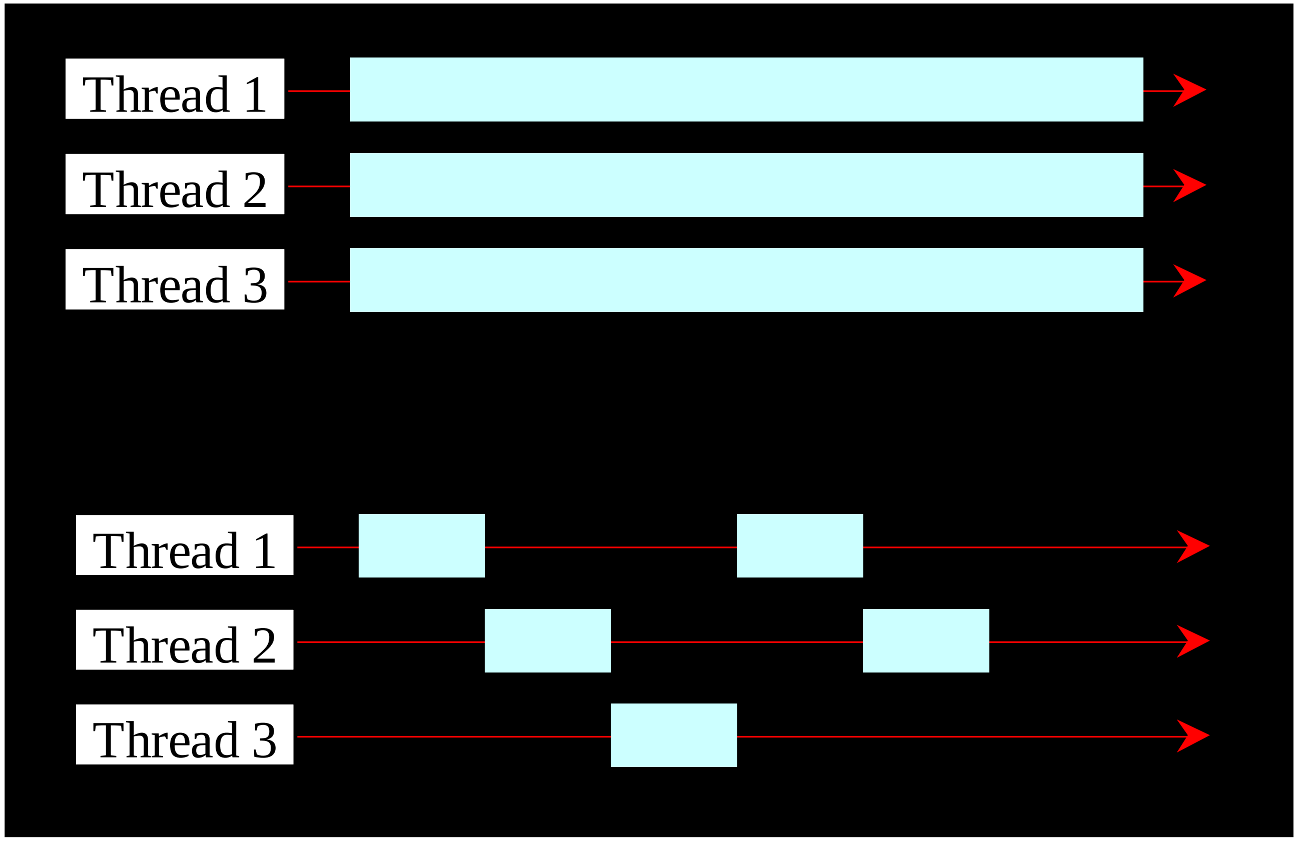
Introduction

- A thread is a flow of execution from beginning to end, of a task in a program.
- Java programs can launch multiple threads concurrently.
- These threads can be executed simultaneously in multiprocessor systems
- In single-processor systems, multiple threads share CPU time. The operating system is responsible for scheduling and allocating resources to them. This is practical because most of the time the CPU is idle waiting for example the user to enter data.

Introduction

Multiple threads on multiple CPUs

Multiple threads sharing a single CPU



Introduction

- Multithreading can make a program more responsive and interactive, as well as enhance performance e.g. a word processor.
- When a program runs as an application, the Java interpreter starts a thread for the main method.
- When a program creates an applet, the web browser starts a thread to run the applet.
- You can create additional threads to run concurrent tasks in your program.

Introduction

- Threads can be created in one of three ways–

1. By **extending** the Thread class.
2. **Implementing** the Runnable interface.
3. Using a **lambda** expression.

Creating Threads By Extending The Thread Class

- To create and run a thread:
  - Define a class that extends the Thread class
  - In the class, override the `run` method

```
// Custom thread class
public class CustomThread extends Thread {
  ...
  public CustomThread(...) {
    ...
  }

  // Override the run method in Thread
  public void run() {
    // Define what the thread does
    ...
  }

  ...
}
```

```
// Client class
public class Client {
  ...
  public someMethod() {
    ...
    // Create the thread
    CustomThread thread = new CustomThread(...);

    // Start the thread
    thread.start();
    ...
  }

  ...
}
```

Creating Threads By Implementing Runnable

- Implement the run method.

```
// Custom thread class
public class CustomThread
  implements Runnable {
  ...
  public CustomThread(...) {
    ...
  }

  // Implement the run method in Runnable
  public void run() {
    // Define what the thread does
    ...
  }

  ...
}
```

```
// Client class
public class Client {
  ...
  public static void main(String[] args) {
    ...
    // Create an instance of CustomThread
    CustomThread customThread
      = new CustomThread(...);

    // Create the thread
    Thread thread = new Thread(customThread);

    // Start the thread
    thread.start();
    ...
  }
  ...
}
```

- Runnable is a functional interface defining a single void no-args method called `run()`.

Creating Threads By Using Lambdas

- Using a lambda expression,

```
// Client class
public class Client {
  ...
  public static void main(String[] args) {

        //define the thread as a lambda
        Runnable customThread = () -> {
            // Define what the thread does
        };

        //create the thread
        Thread t1 = new Thread(customThread);

        //start the thread
        t1.start();

  }
```

Examples

- Consult the three examples in the *thread_basics* package.
- Each example creates two threads:

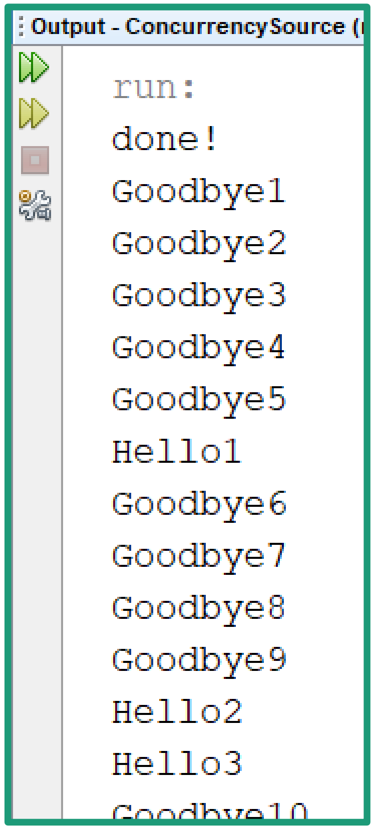
  - One thread to print the word “Hello” 100 times.

  - Another to create a separate thread to print the word “Goodbye” 100 times.

- Run the code examples and note the interleaved output.

Examples

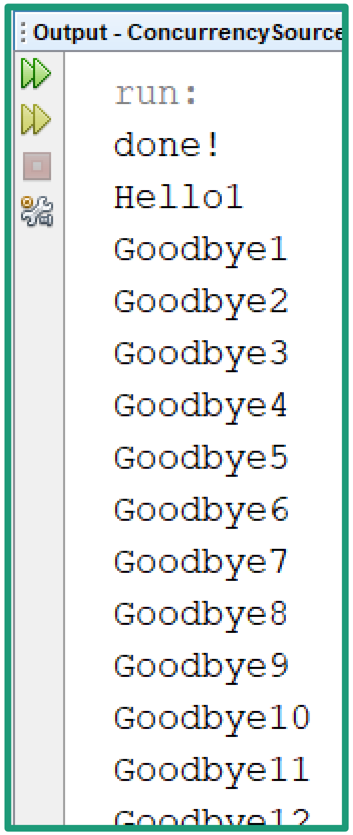
Run 1



Output - ConcurrencySource (

```
run:
done!
Goodbye1
Goodbye2
Goodbye3
Goodbye4
Goodbye5
Hello1
Goodbye6
Goodbye7
Goodbye8
Goodbye9
Hello2
Hello3
Goodbye10
```

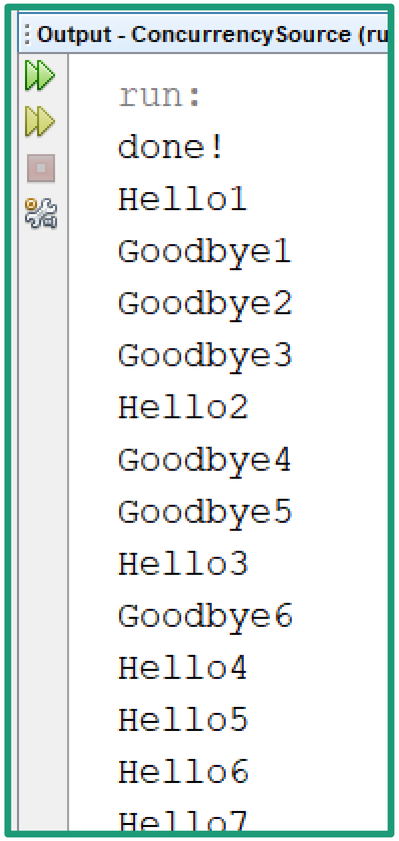
Run 2



Output - ConcurrencySource

```
run:
done!
Hello1
Goodbye1
Goodbye2
Goodbye3
Goodbye4
Goodbye5
Goodbye6
Goodbye7
Goodbye8
Goodbye9
Goodbye10
Goodbye11
Goodbye12
```

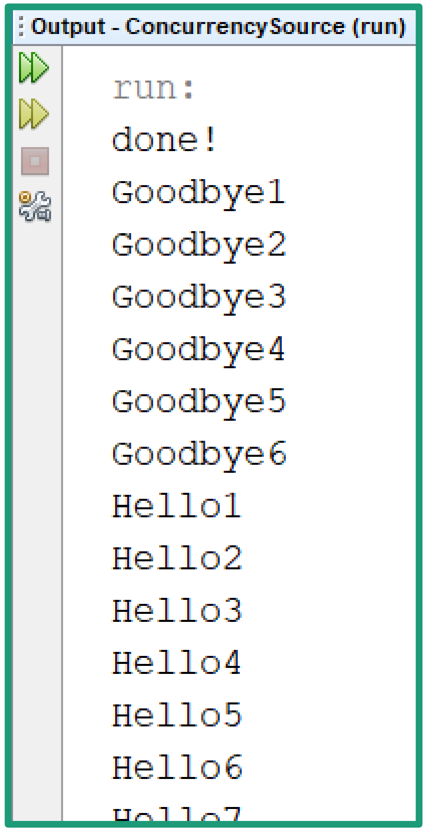
Run 3



Output - ConcurrencySource (ru

```
run:
done!
Hello1
Goodbye1
Goodbye2
Goodbye3
Hello2
Goodbye4
Goodbye5
Hello3
Goodbye6
Hello4
Hello5
Hello6
Hello7
```

Run 4

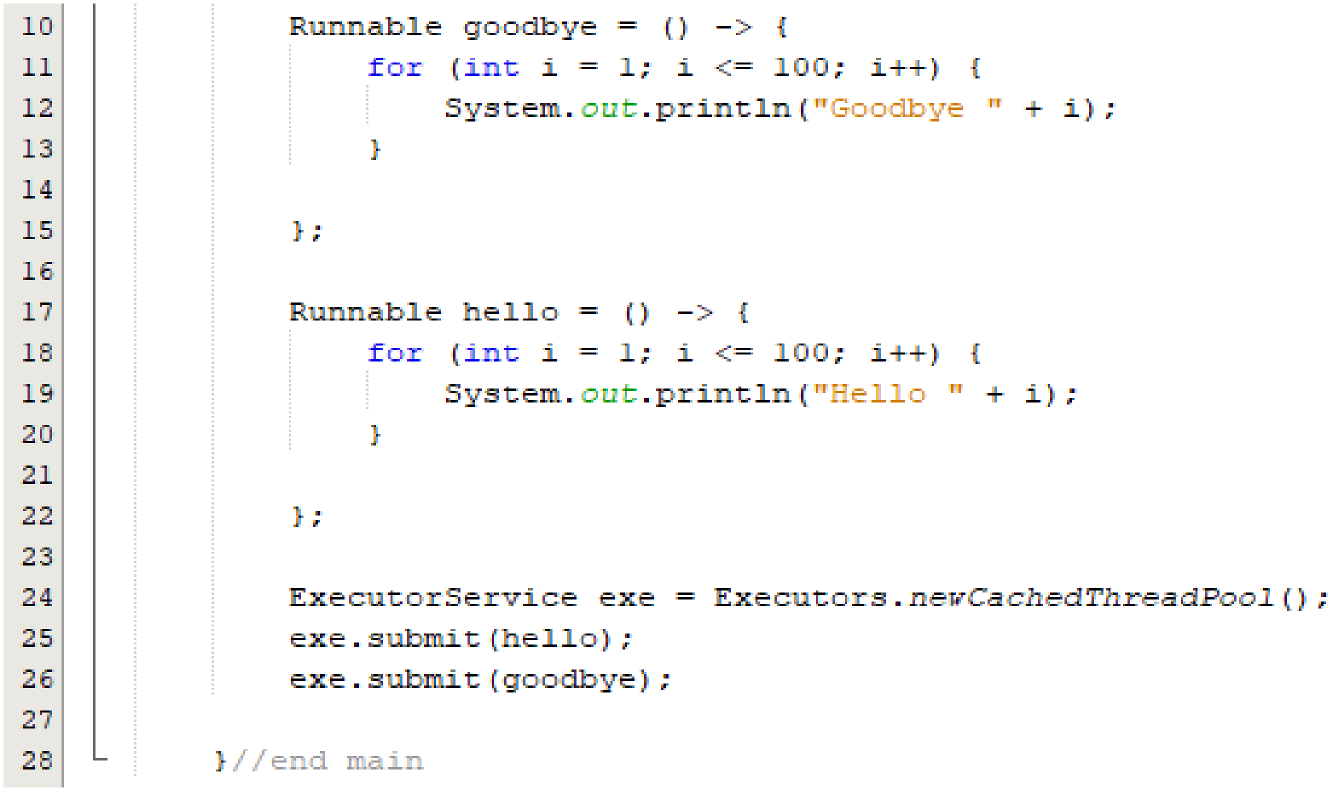


Output - ConcurrencySource (run)

```
run:
done!
Goodbye1
Goodbye2
Goodbye3
Goodbye4
Goodbye5
Goodbye6
Hello1
Hello2
Hello3
Hello4
Hello5
Hello6
Hello7
```

Executors: Example 1

- The ExecutorService is a higher level replacement for working with threads.
  - You don’t have to create threads manually.

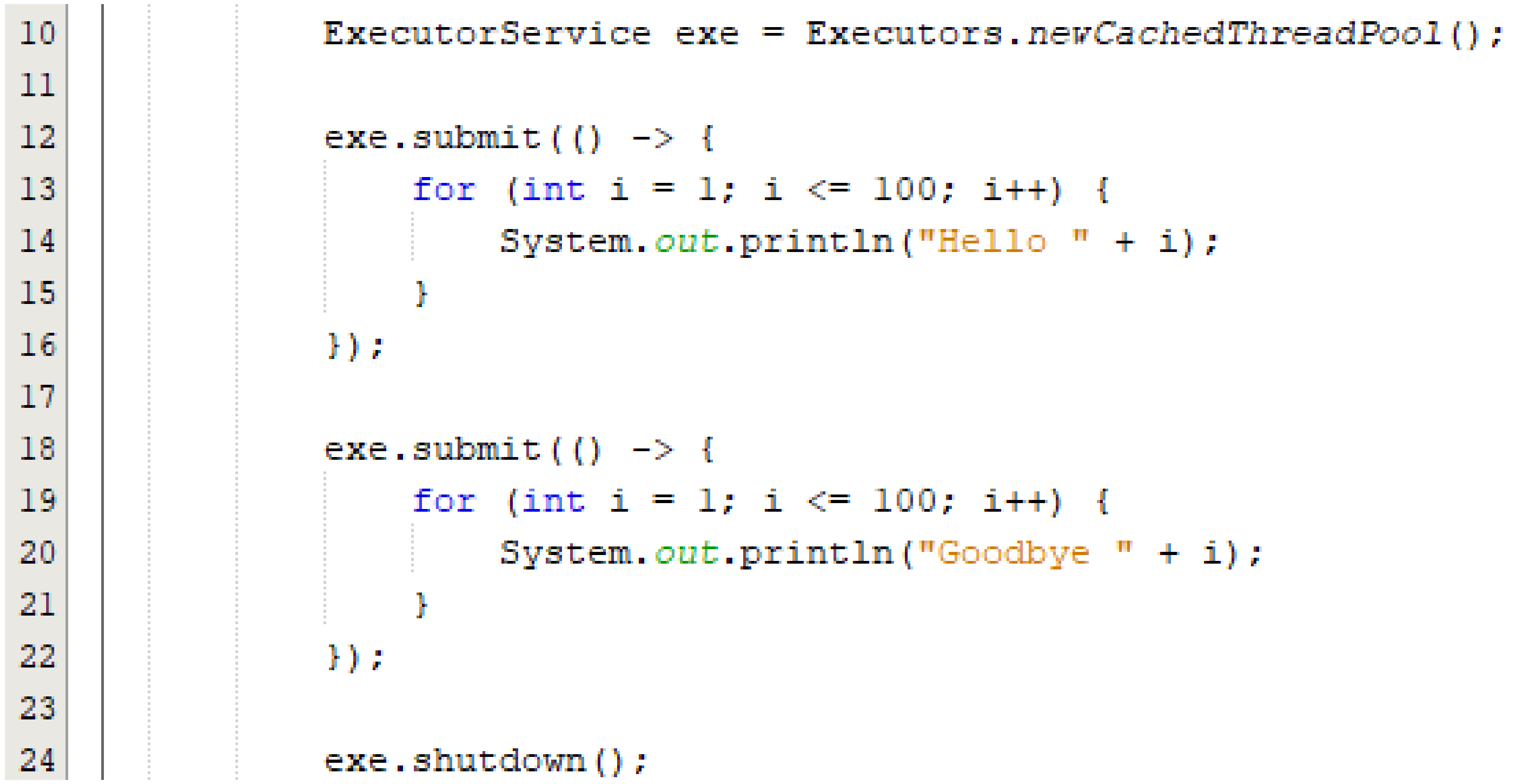


```java
        Runnable goodbye = () -> {
            for (int i = 1; i <= 100; i++) {
                System.out.println("Goodbye " + i);
            }

        };

        Runnable hello = () -> {
            for (int i = 1; i <= 100; i++) {
                System.out.println("Hello " + i);
            }

        };

        ExecutorService exe = Executors.newCachedThreadPool();
        exe.submit(hello);
        exe.submit(goodbye);

    }//end main
```

using_executors. TestExecutorsEX1.java

Executors: Example 2

- Alternatively…

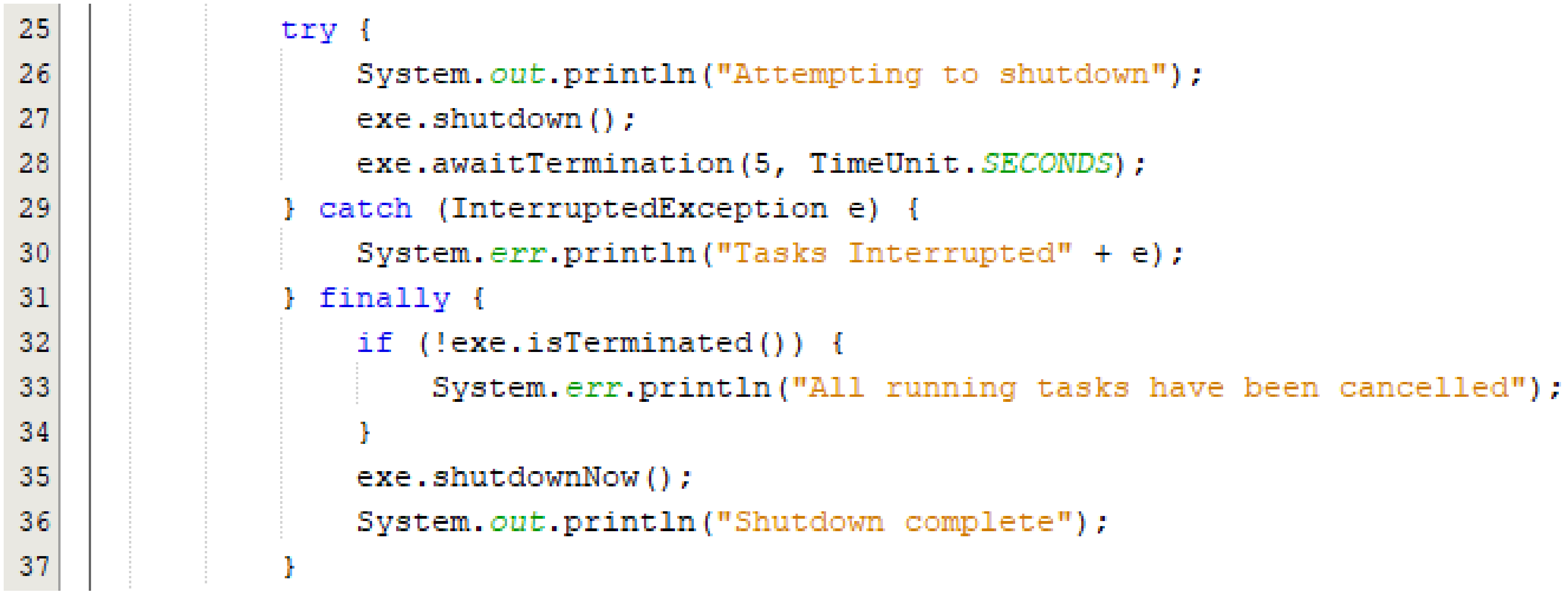


```java
ExecutorService exe = Executors.newCachedThreadPool();

exe.submit(() -> {
    for (int i = 1; i <= 100; i++) {
        System.out.println("Hello " + i);
    }
});

exe.submit(() -> {
    for (int i = 1; i <= 100; i++) {
        System.out.println("Goodbye " + i);
    }
});

exe.shutdown();
```

using_executors. TestExecutorsEX2.java

Executors: Example 2

- A better way to shutdown tasks – a “soft” shutdown.



```java
try {
    System.out.println("Attempting to shutdown");
    exe.shutdown();
    exe.awaitTermination(5, TimeUnit.SECONDS);
} catch (InterruptedException e) {
    System.err.println("Tasks Interrupted" + e);
} finally {
    if (!exe.isTerminated()) {
        System.err.println("All running tasks have been cancelled");
    }
    exe.shutdownNow();
    System.out.println("Shutdown complete");
}
```

using_executors. TestExecutorsEX2.java

Executors

- The Executors class has factory methods for executor services with different scheduling policies.

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

- Returns an executor optimised for programs with many short lived tasks or spend most of their time waiting.

Executors

```
ExecutorService exe =
     Executors.newFixedThreadPool(nthreads);
```

- Returns an executor with a fixed number of threads. When you submit a task it is queued until a thread becomes available.
- A good choice for intensive tasks or if you need to limit resource consumption.

Callables and Futures

- Recall that Runnable is a functional interface defining a single void no-args method called `run()`.
- Executors support another kind of task named Callable.
    - Callable is a functional interfaces just like Runnable but it returns a value.
    - Callable has one method - `call()`.
- When you submit the Callable task you get a future.
    - A future is an object that represents the computation that took place within the Callable.

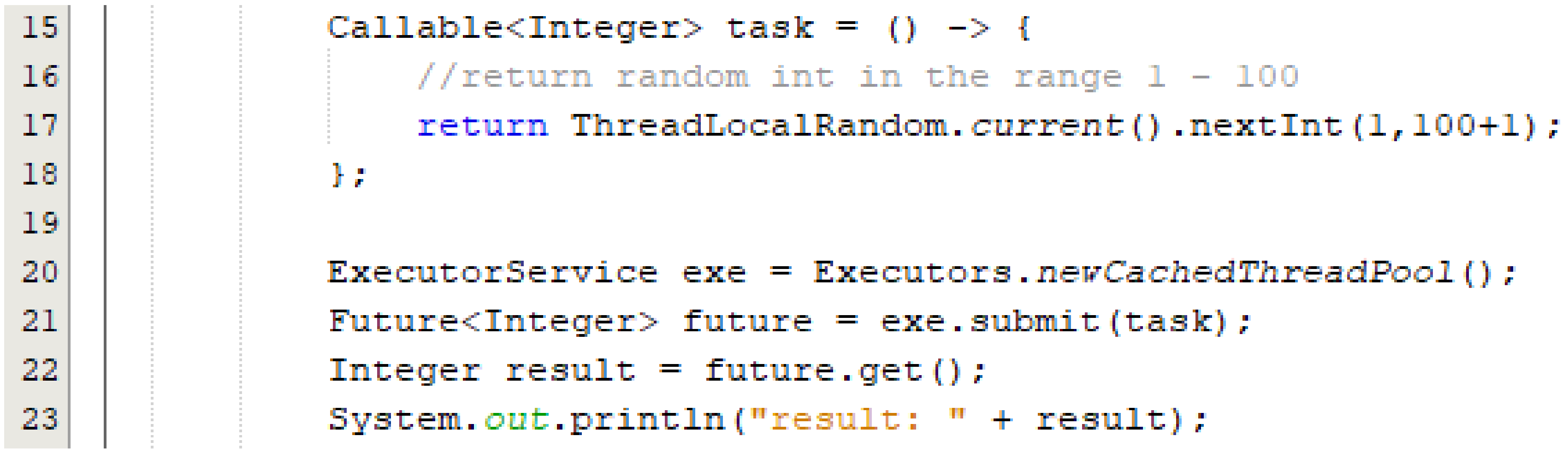
Callables and Futures

- The Callable interface is as follows:

```
public interface Callable<V> {
   V call() throws Exception;
}
```

Callables and Futures: Example 1

- An example of using a Callable and a Future.

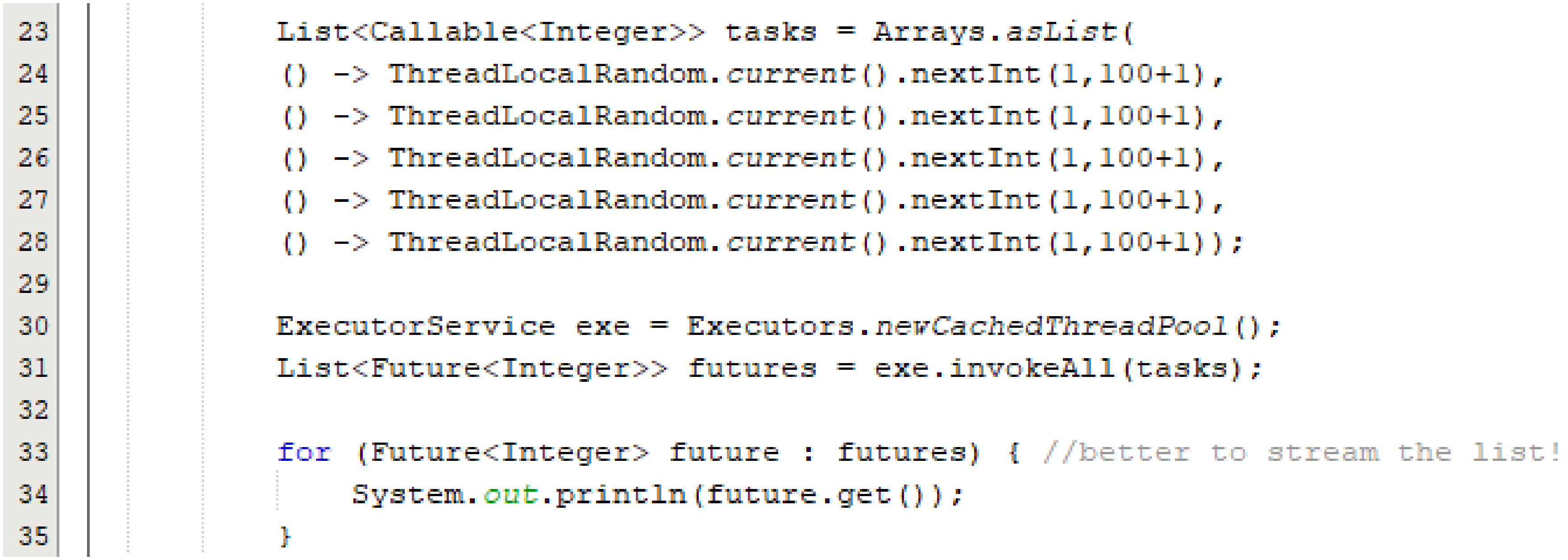


```
Callable<Integer> task = () -> {
    //return random int in the range 1 - 100
    return ThreadLocalRandom.current().nextInt(1,100+1);
};

ExecutorService exe = Executors.newCachedThreadPool();
Future<Integer> future = exe.submit(task);
Integer result = future.get();
System.out.println("result: " + result);
```

callables_and_futures.UsingCallablesAndFuturesEX1.java

Callables and Futures: Example 2

- A task may need to wait for the result of multiple subtasks.
- Instead of submitting each subtask separately use `invokeAll()`.

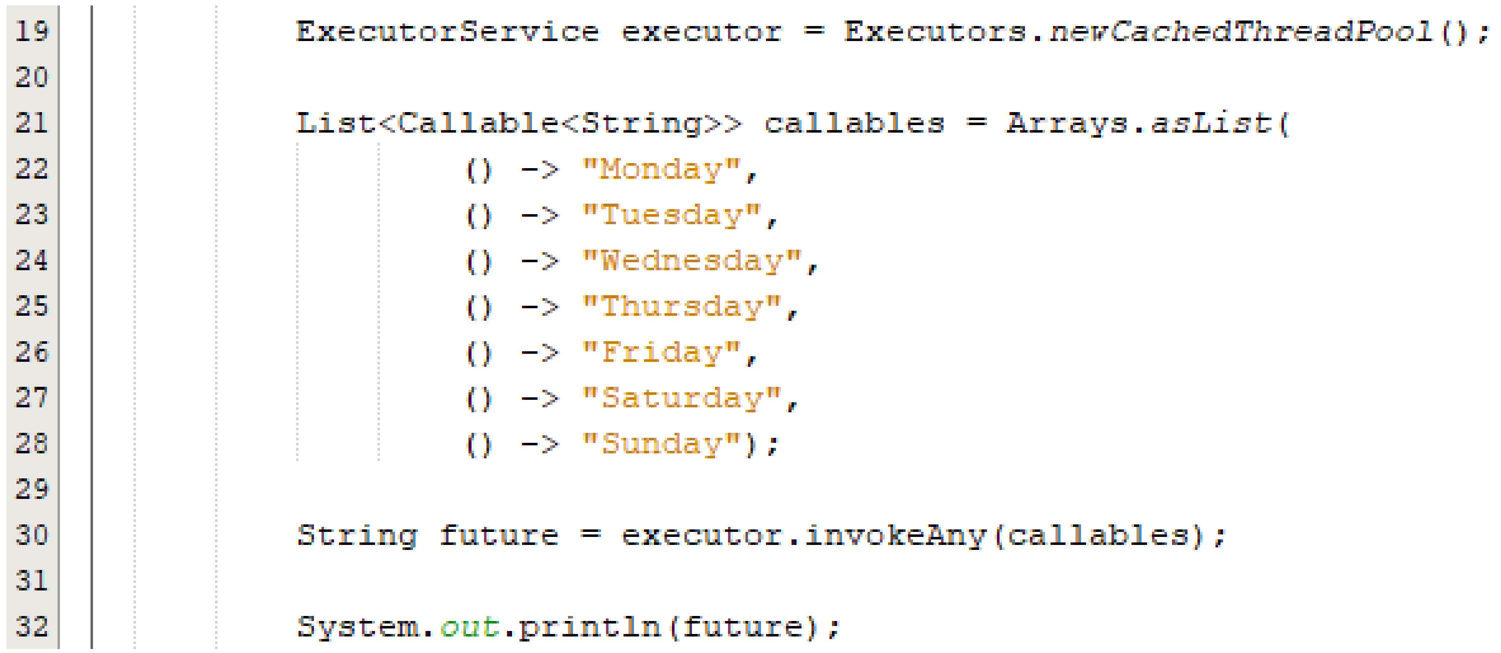


```java
List<Callable<Integer>> tasks = Arrays.asList(
() -> ThreadLocalRandom.current().nextInt(1,100+1),
() -> ThreadLocalRandom.current().nextInt(1,100+1),
() -> ThreadLocalRandom.current().nextInt(1,100+1),
() -> ThreadLocalRandom.current().nextInt(1,100+1),
() -> ThreadLocalRandom.current().nextInt(1,100+1));

ExecutorService exe = Executors.newCachedThreadPool();
List<Future<Integer>> futures = exe.invokeAll(tasks);

for (Future<Integer> future : futures) { //better to stream the list!
    System.out.println(future.get());
}
```

callables_and_futures.UsingCallablesAndFuturesEX2.java

Callables and Futures: Example 3

- `invokeAny()` is similar to `invokeAll()`.
  - It returns as soon as any one of the submitted tasks returns the value of its `Future` - the other tasks are cancelled.

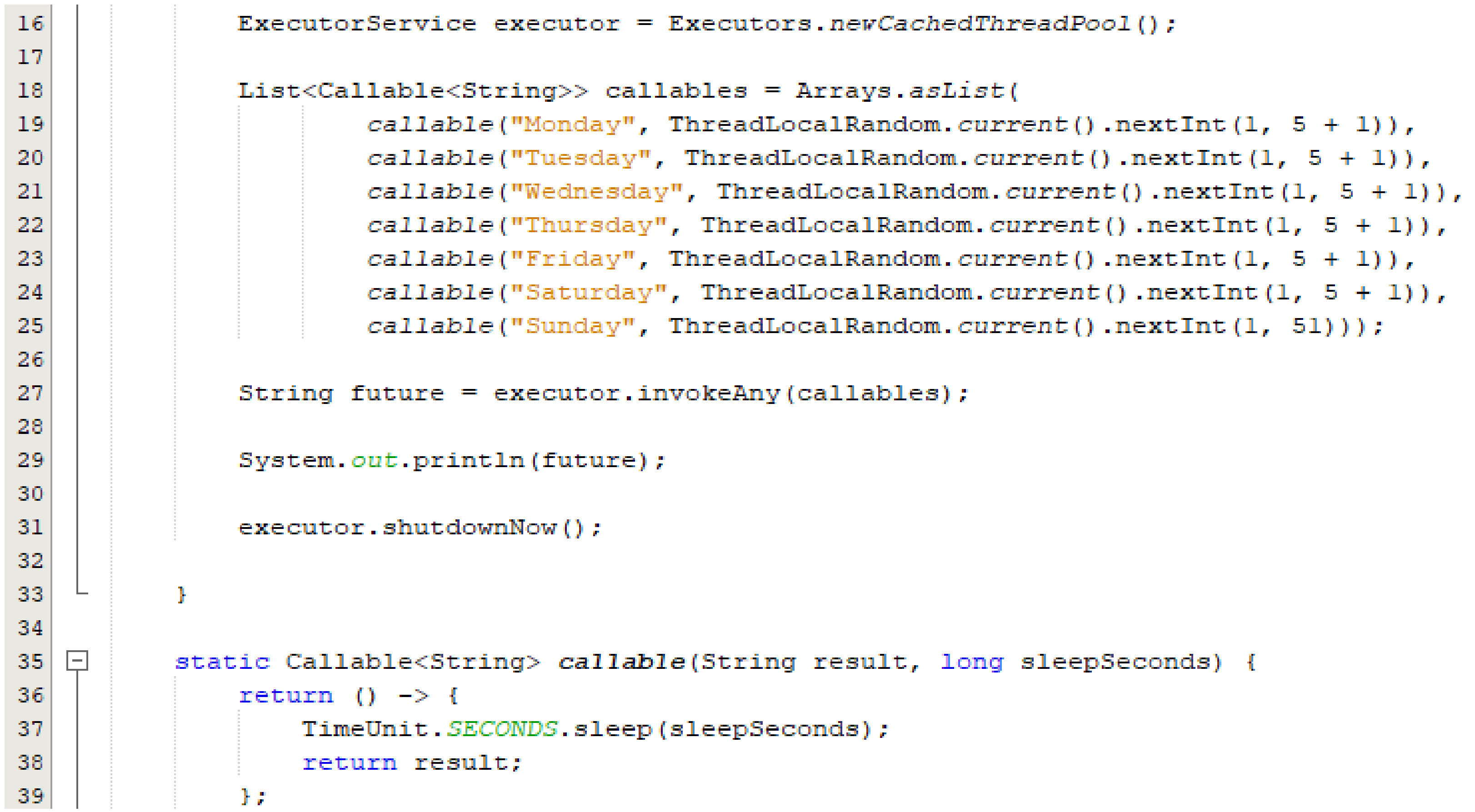


```java
ExecutorService executor = Executors.newCachedThreadPool();

List<Callable<String>> callables = Arrays.asList(
        () -> "Monday",
        () -> "Tuesday",
        () -> "Wednesday",
        () -> "Thursday",
        () -> "Friday",
        () -> "Saturday",
        () -> "Sunday");

String future = executor.invokeAny(callables);

System.out.println(future);
```

callables_and_futures.UsingCallablesAndFuturesEX3.java

Callables and Futures: Example 4

- Simulate tasks taking unpredictable lengths to compete.



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

        List<Callable<String>> callables = Arrays.asList(
                callable("Monday", ThreadLocalRandom.current().nextInt(1, 5 + 1)),
                callable("Tuesday", ThreadLocalRandom.current().nextInt(1, 5 + 1)),
                callable("Wednesday", ThreadLocalRandom.current().nextInt(1, 5 + 1)),
                callable("Thursday", ThreadLocalRandom.current().nextInt(1, 5 + 1)),
                callable("Friday", ThreadLocalRandom.current().nextInt(1, 5 + 1)),
                callable("Saturday", ThreadLocalRandom.current().nextInt(1, 5 + 1)),
                callable("Sunday", ThreadLocalRandom.current().nextInt(1, 51)));

        String future = executor.invokeAny(callables);

        System.out.println(future);

        executor.shutdownNow();

    }

    static Callable<String> callable(String result, long sleepSeconds) {
        return () -> {
            TimeUnit.SECONDS.sleep(sleepSeconds);
            return result;
        };
```

callables_and_futures.UsingCallablesAndFuturesEX4.java

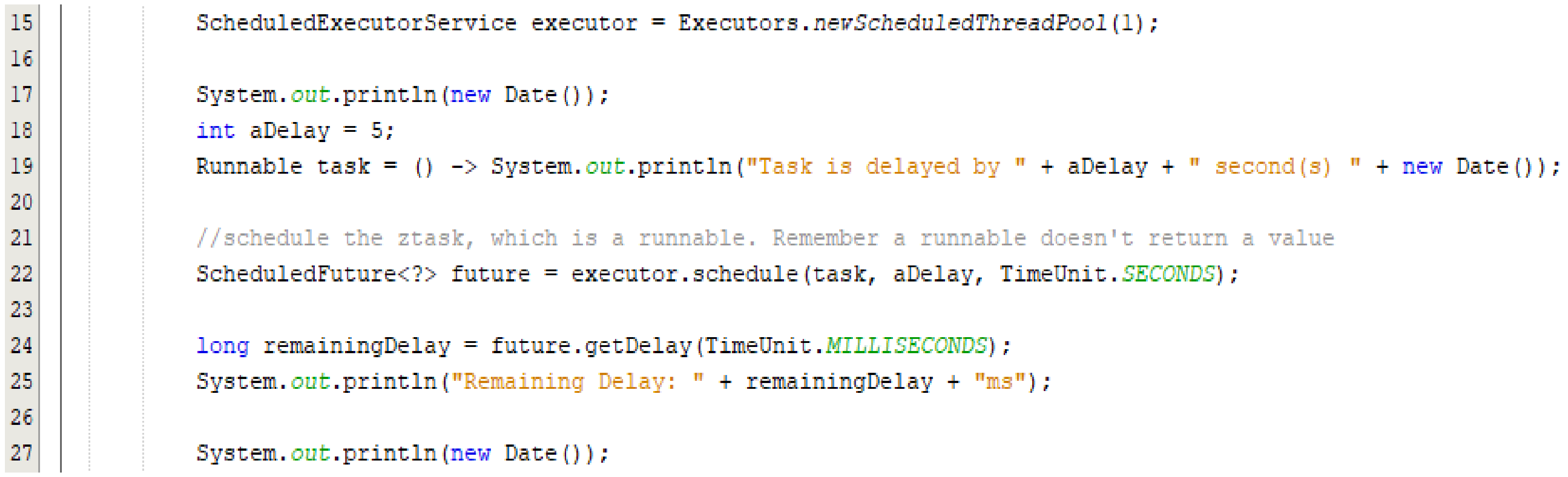
Callables and Futures

- Useful methods of the Future interface:

Return Type	Method Name	Description
V	get()	Waits if necessary for the computation to complete, and then retrieves its result.
V	get(long timeout, TimeUnit unit)	Waits if necessary for at most the given time for the computation to complete, and then retrieves its result, if availabl
boolean	cancel(boolean mayInterruptIfRunning)	Attempts to cancel execution of this task.
boolean	isCancelled()	Returns true if this task was cancelled before it completed normally
boolean	isDone()	Returns true if this task completed.

Scheduling Tasks: Example 1

- A `ScheduledExecutorService` schedules tasks to run either periodically or after a period of time has passed.
- Here is an example of a scheduled task, delayed by 5 seconds.



```
        ScheduledExecutorService executor = Executors.newScheduledThreadPool(1);

        System.out.println(new Date());
        int aDelay = 5;
        Runnable task = () -> System.out.println("Task is delayed by " + aDelay + " second(s) " + new Date());

        //schedule the ztask, which is a runnable. Remember a runnable doesn't return a value
        ScheduledFuture<?> future = executor.schedule(task, aDelay, TimeUnit.SECONDS);

        long remainingDelay = future.getDelay(TimeUnit.MILLISECONDS);
        System.out.println("Remaining Delay: " + remainingDelay + "ms");

        System.out.println(new Date());
```

schedulers.UsingAScheulerEX1.java

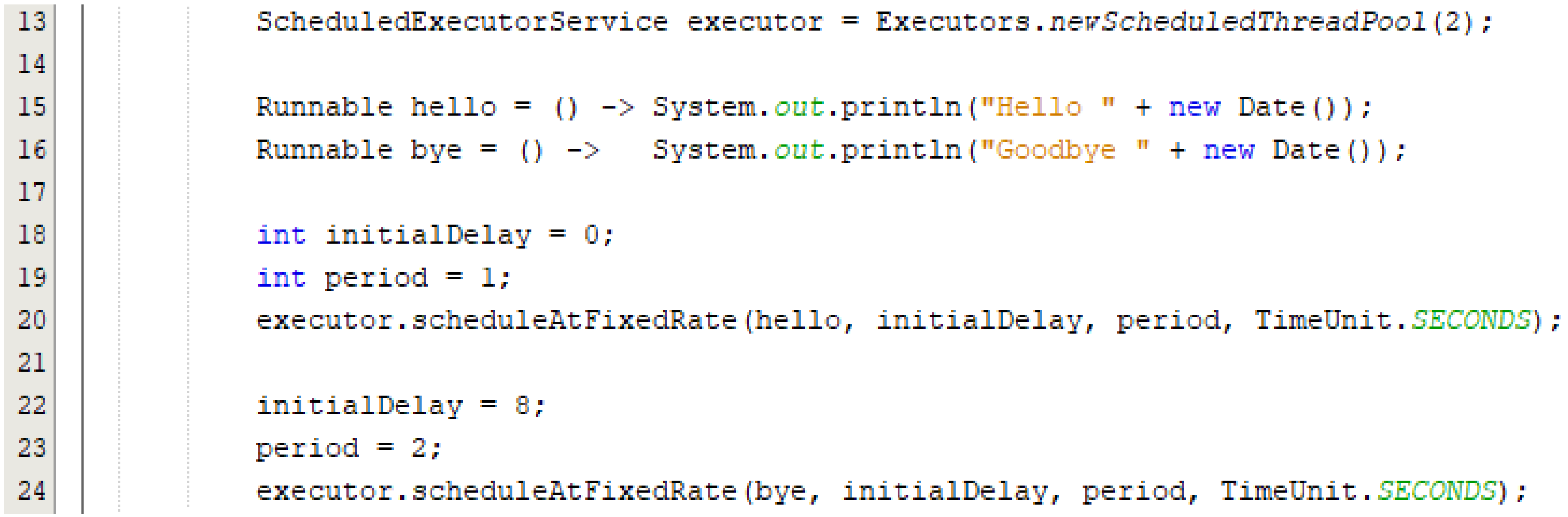
Scheduling Tasks: Example 1

- In order to schedule tasks to be executed periodically, executors provide two methods:

`scheduleAtFixedRate()`

  - Capable of executing tasks with a fixed time rate.
  - Additionally this method accepts an initial delay which describes the leading wait time before the task will be executed for the first time.

Scheduling Tasks: Example 2



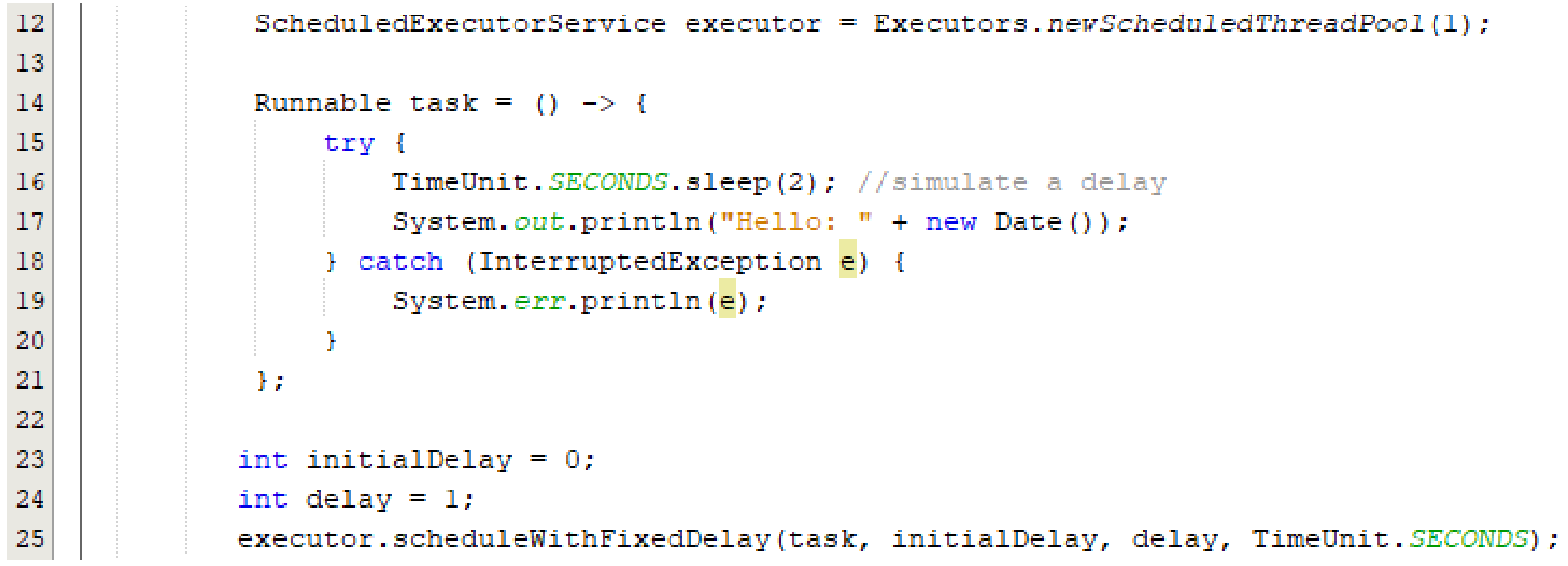
```
        ScheduledExecutorService executor = Executors.newScheduledThreadPool(2);

        Runnable hello = () -> System.out.println("Hello " + new Date());
        Runnable bye = () ->   System.out.println("Goodbye " + new Date());

        int initialDelay = 0;
        int period = 1;
        executor.scheduleAtFixedRate(hello, initialDelay, period, TimeUnit.SECONDS);

        initialDelay = 8;
        period = 2;
        executor.scheduleAtFixedRate(bye, initialDelay, period, TimeUnit.SECONDS);
```

schedulers.UsingAScheulerEX2.java

Scheduling Tasks: Example 3

`scheduleWithFixedDelay()`

- This method works just like the first. The difference is that the wait time period applies between the end of a task and the start of the next task.



```
 ScheduledExecutorService executor = Executors.newScheduledThreadPool(1);

 Runnable task = () -> {
     try {
         TimeUnit.SECONDS.sleep(2); //simulate a delay
         System.out.println("Hello: " + new Date());
     } catch (InterruptedException e) {
         System.err.println(e);
     }
 };

int initialDelay = 0;
int delay = 1;
executor.scheduleWithFixedDelay(task, initialDelay, delay, TimeUnit.SECONDS);
```

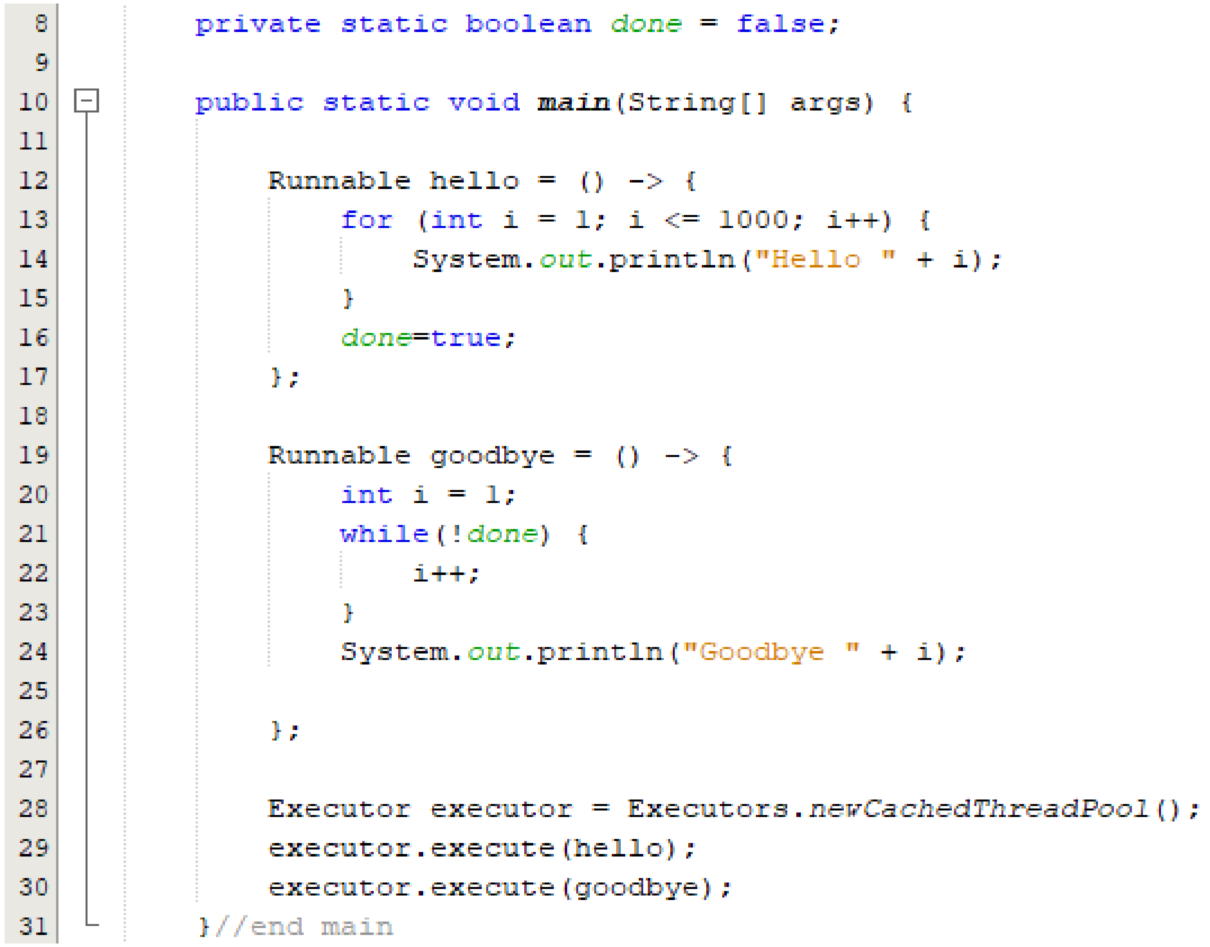
schedulers.UsingAScheulerEX3.java

Thread Safety

- Many think concurrent programming is easy.
  - Divide up your work into tasks.
  - Start the threads.
  - Many any results (if any).
- However, much can go wrong.
- Even reading/writing a variable can be problematic.

Thread Safety

- Consider this example.

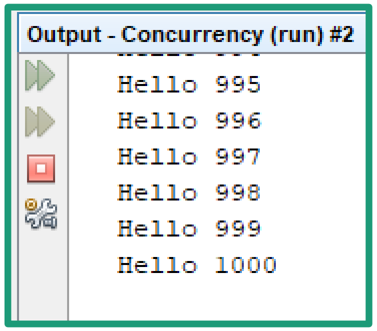


```java
private static boolean done = false;

public static void main(String[] args) {

    Runnable hello = () -> {
        for (int i = 1; i <= 1000; i++) {
            System.out.println("Hello " + i);
        }
        done=true;
    };

    Runnable goodbye = () -> {
        int i = 1;
        while(!done) {
            i++;
        }
        System.out.println("Goodbye " + i);

    };

    Executor executor = Executors.newCachedThreadPool();
    executor.execute(hello);
    executor.execute(goodbye);
}//end main
```

thread_safety.ThreadSafetyProblemEX1.java

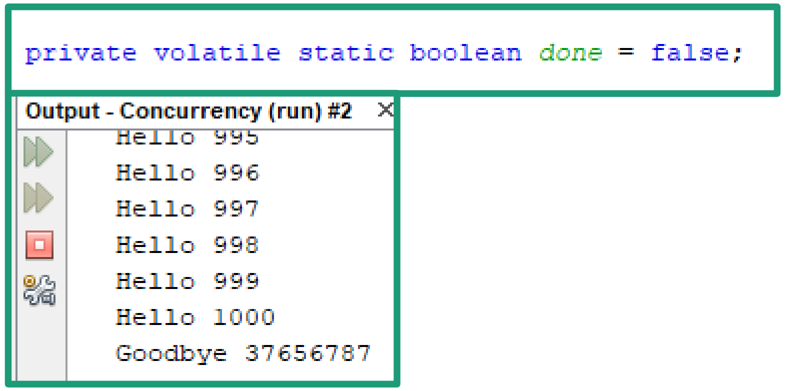
Thread Safety

- The first task prints "Hello" 1000 times and sets `done` to `true`.
- The second task waits for `done` to become `true` and prints "Goodbye" once, incrementing a counter while it is waiting for `done` (to become `true`).
- However, when the code is run, "Goodbye" is never printed.



Output - Concurrency (run) #2

```
Hello 995
Hello 996
Hello 997
Hello 998
Hello 999
Hello 1000
```



```
private volatile static boolean done = false;
```

Output - Concurrency (run) #2

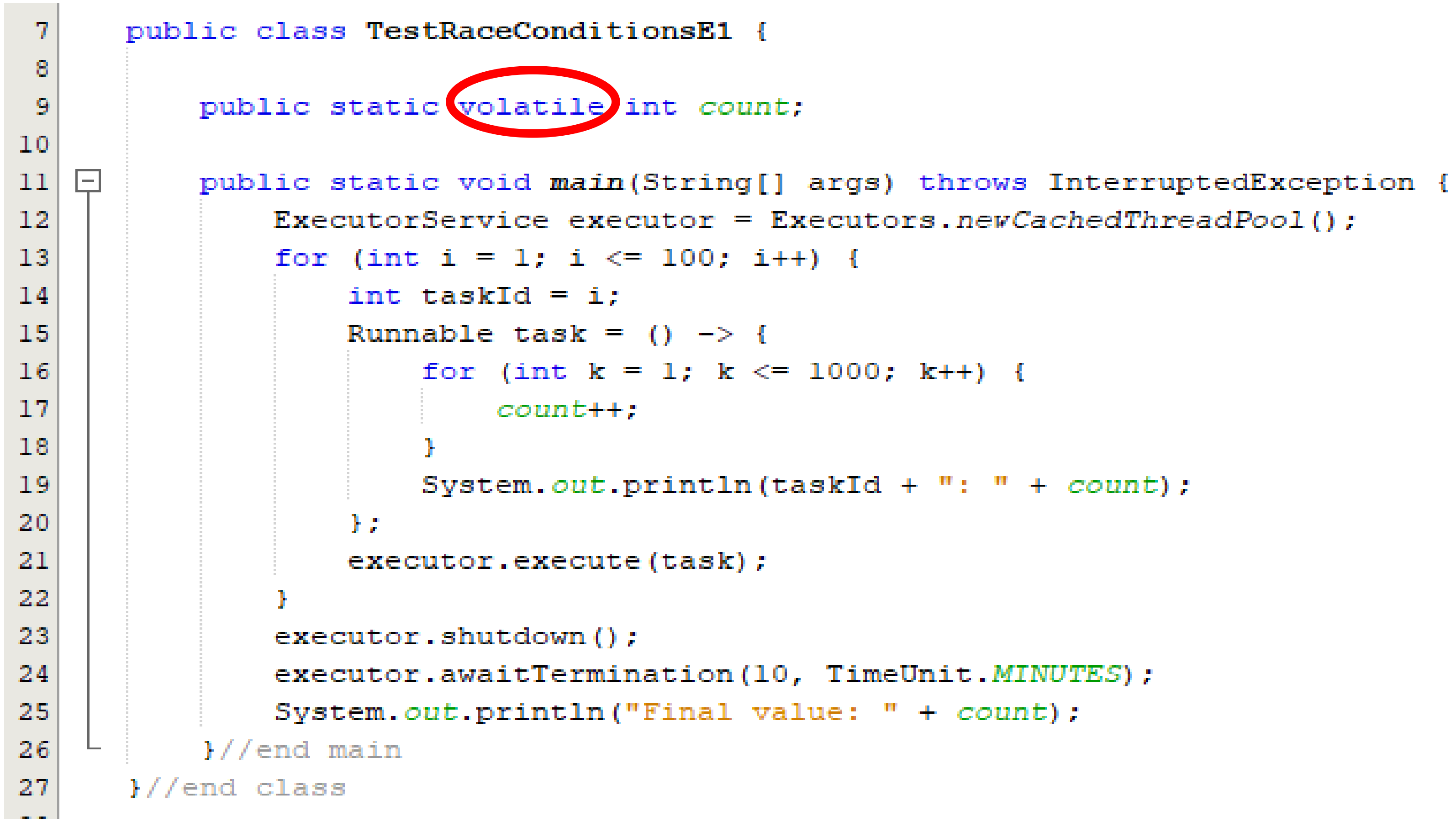
```
Hello 995
Hello 996
Hello 997
Hello 998
Hello 999
Hello 1000
Goodbye 37656787
```

Race Conditions

- A *race condition* is a special condition that may occur inside an *instruction sequence*.
    - Also known as a *critical section*.
- A *critical section* is a block of code that is executed by multiple threads and where the sequence of execution for the threads makes a difference in the outcome of this critical section.
- When the result of multiple threads executing a critical section may differ depending on the sequence in which the threads execute, the critical section is said to contain a race condition.
- How common are race conditions?
    - Very.

Race Conditions: Example 1

- Create 100 threads. Each thread increments the counter 100 times and prints the result.

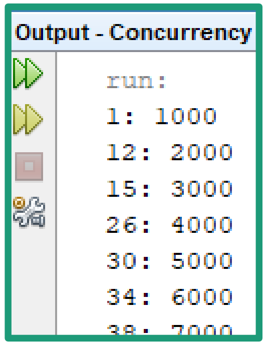


```java
public class TestRaceConditionsE1 {

    public static volatile int count;

    public static void main(String[] args) throws InterruptedException {
        ExecutorService executor = Executors.newCachedThreadPool();
        for (int i = 1; i <= 100; i++) {
            int taskId = i;
            Runnable task = () -> {
                for (int k = 1; k <= 1000; k++) {
                    count++;
                }
                System.out.println(taskId + ": " + count);
            };
            executor.execute(task);
        }
        executor.shutdown();
        executor.awaitTermination(10, TimeUnit.MINUTES);
        System.out.println("Final value: " + count);
    }//end main
}//end class
```

race_conditions.TestRaceConditionsEX1.java

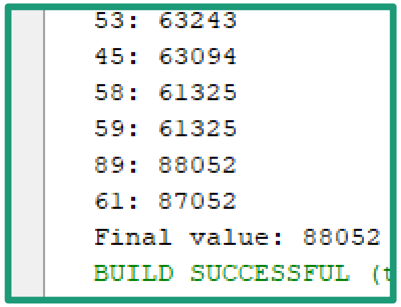
Race Conditions: Example 1

- The output generally starts off as expected:

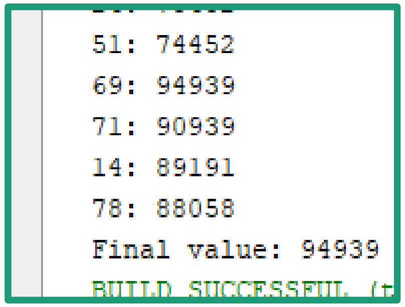


```
Output - Concurrency
run:
1: 1000
12: 2000
15: 3000
26: 4000
30: 5000
34: 6000
38: 7000
```

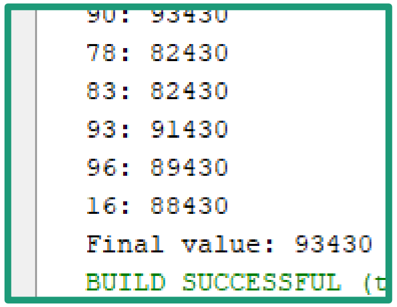
- However…no matter how many times you run it, the final answer is never as expected.



```
53: 63243
45: 63094
58: 61325
59: 61325
89: 88052
61: 87052
Final value: 88052
BUILD SUCCESSFUL (t
```



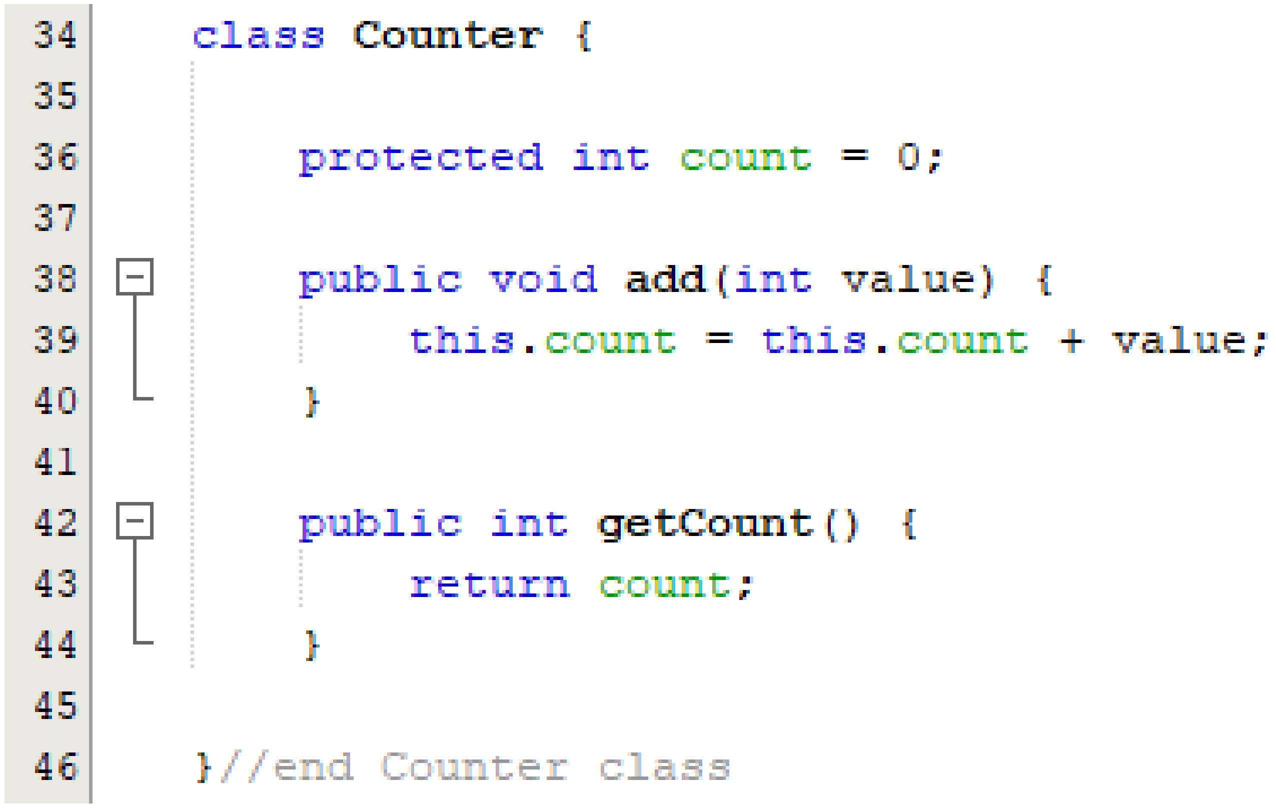
```
51: 74452
69: 94939
71: 90939
14: 89191
78: 88058
Final value: 94939
BUILD SUCCESSFUL (t
```



```
90: 93430
78: 82430
83: 82430
93: 91430
96: 89430
16: 88430
Final value: 93430
BUILD SUCCESSFUL (t
```

Race Conditions: Example 2

- Consider a simple Counter class.

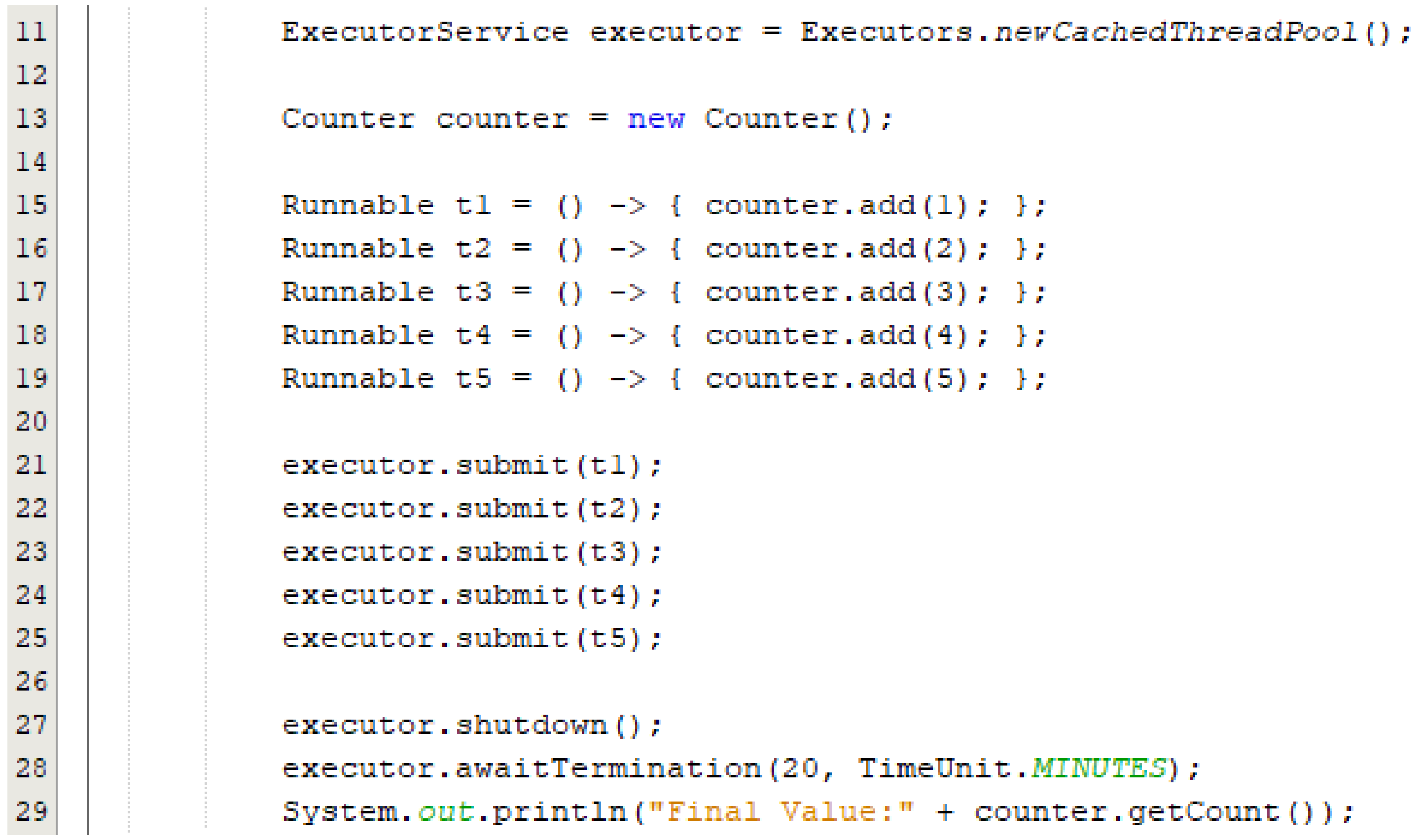


```
class Counter {

    protected int count = 0;

    public void add(int value) {
        this.count = this.count + value;
    }

    public int getCount() {
        return count;
    }

}//end Counter class
```

race_conditions.TestRaceConditionsEX2.java

Race Conditions: Example 2

- Create five tasks. Each task will increase the counter by 1,2,3,4,5 respectively.

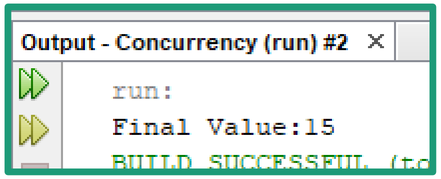


```java
        ExecutorService executor = Executors.newCachedThreadPool();

        Counter counter = new Counter();

        Runnable t1 = () -> { counter.add(1); };
        Runnable t2 = () -> { counter.add(2); };
        Runnable t3 = () -> { counter.add(3); };
        Runnable t4 = () -> { counter.add(4); };
        Runnable t5 = () -> { counter.add(5); };

        executor.submit(t1);
        executor.submit(t2);
        executor.submit(t3);
        executor.submit(t4);
        executor.submit(t5);

        executor.shutdown();
        executor.awaitTermination(20, TimeUnit.MINUTES);
        System.out.println("Final Value:" + counter.getCount());
```

race_conditions.TestRaceConditionsEX2.java

Race Conditions: Example 2

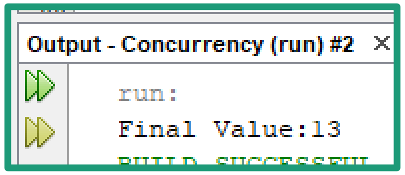
- What is the final output?
- 15? Sometimes!



Output - Concurrency (run) #2

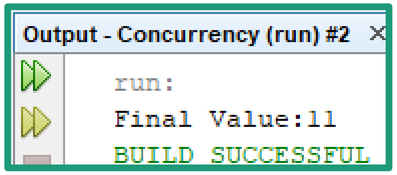
```
run:
Final Value:15
BUILD SUCCESSFUL (to
```

- Not always.



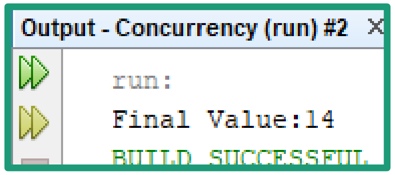
Output - Concurrency (run) #2

```
run:
Final Value:13
BUILD SUCCESSFUL
```



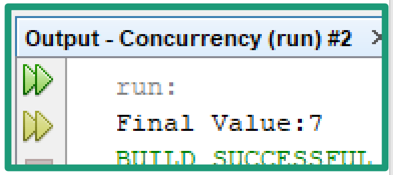
Output - Concurrency (run) #2

```
run:
Final Value:11
BUILD SUCCESSFUL
```



Output - Concurrency (run) #2

```
run:
Final Value:14
BUILD SUCCESSFUL
```



Output - Concurrency (run) #2

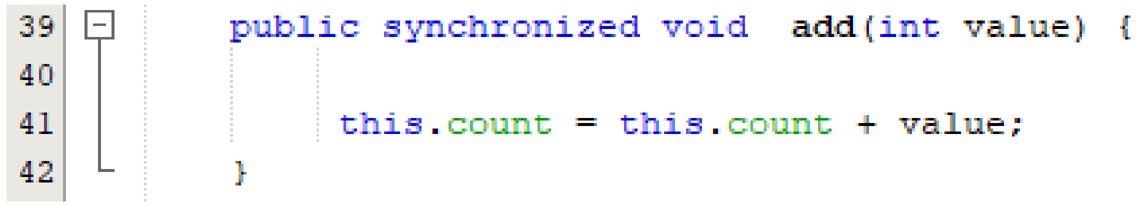
```
run:
Final Value:7
BUILD SUCCESSFUL
```

Race Conditions

- Solutions?
- Locks are commonly used to protect critical sections.
  - Make them atomic (synchronized)
  - Once a single thread is executing a critical section, no other threads can execute it until the first thread has left the critical section.

Synchronization

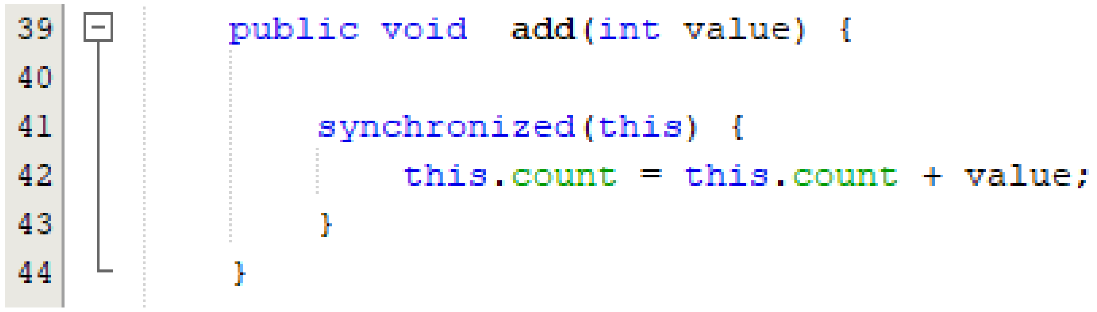
- Java uses the keyword `synchronized` to synchronize method invocation so that only one thread can be in a method at a time.
- In the previous example, marking the `add` method as `synchronized` solves the problem.



```
    public synchronized void  add(int value) {

            this.count = this.count + value;
    }
```

Synchronization

- The synchronized keyword can be used to mark four different types of blocks:
  1. Instance methods.
  2. Static methods.
  3. Code blocks inside instance methods.
  4. Code blocks inside static methods.



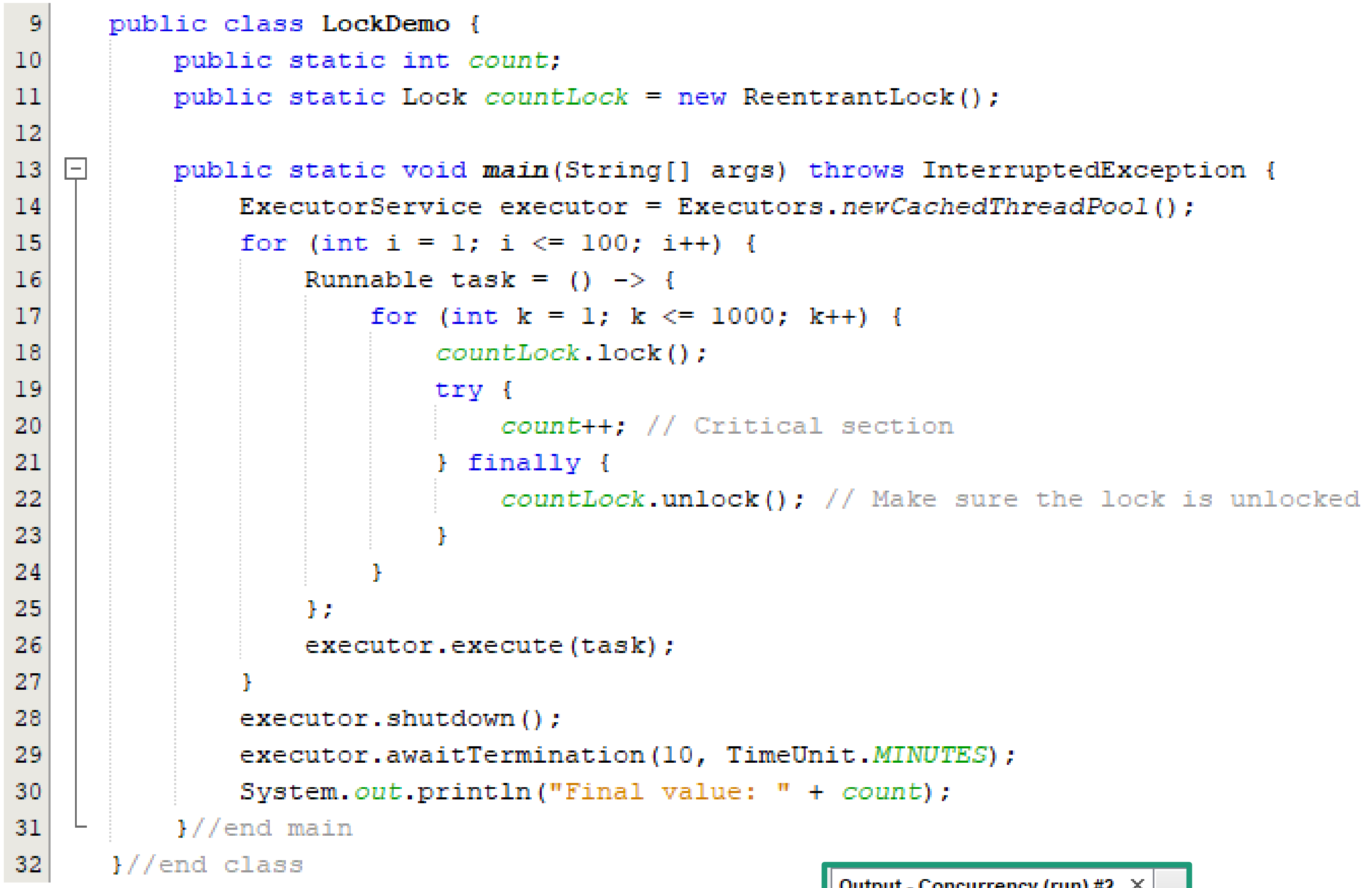
```java
public void  add(int value) {

    synchronized(this) {
        this.count = this.count + value;
    }
}
```

Locks

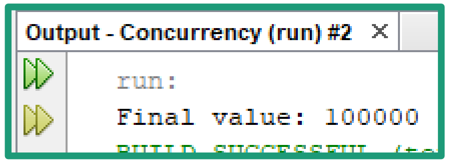
- The synchronized mechanism was Java's first mechanism for synchronizing access to objects shared by multiple threads.
- The synchronized mechanism isn't very advanced though.
    - Locks are an alternative.
- As a example, look to solve the problem within *race_conditions.TestRaceConditionsEX1.java*

  with a lock.

Locks



```java
public class LockDemo {
    public static int count;
    public static Lock countLock = new ReentrantLock();

    public static void main(String[] args) throws InterruptedException {
        ExecutorService executor = Executors.newCachedThreadPool();
        for (int i = 1; i <= 100; i++) {
            Runnable task = () -> {
                for (int k = 1; k <= 1000; k++) {
                    countLock.lock();
                    try {
                        count++; // Critical section
                    } finally {
                        countLock.unlock(); // Make sure the lock is unlocked
                    }
                }
            };
            executor.execute(task);
        }
        executor.shutdown();
        executor.awaitTermination(10, TimeUnit.MINUTES);
        System.out.println("Final value: " + count);
    }//end main
}//end class
```



Output - Concurrency (run) #2

```
run:
Final value: 100000
BUILD SUCCESSFUL (to
```

using_locks.LockDemo.java

Locks

- The first thread to execute the lock method, locks the `countLock` object and proceeds into the critical section.
- If another thread attempts to call `lock` on the same object it is blocked until the first thread executes the call to `unlock`.
  - Guarantees that only one thread at a time can execute the critical section.
- Often programmers find it difficult to work with locks.
  - Often use the wrong lock or create deadlock.

Deadlock

- A situation wherein two or more competing actions are waiting for the other to finish, and thus neither ever does.

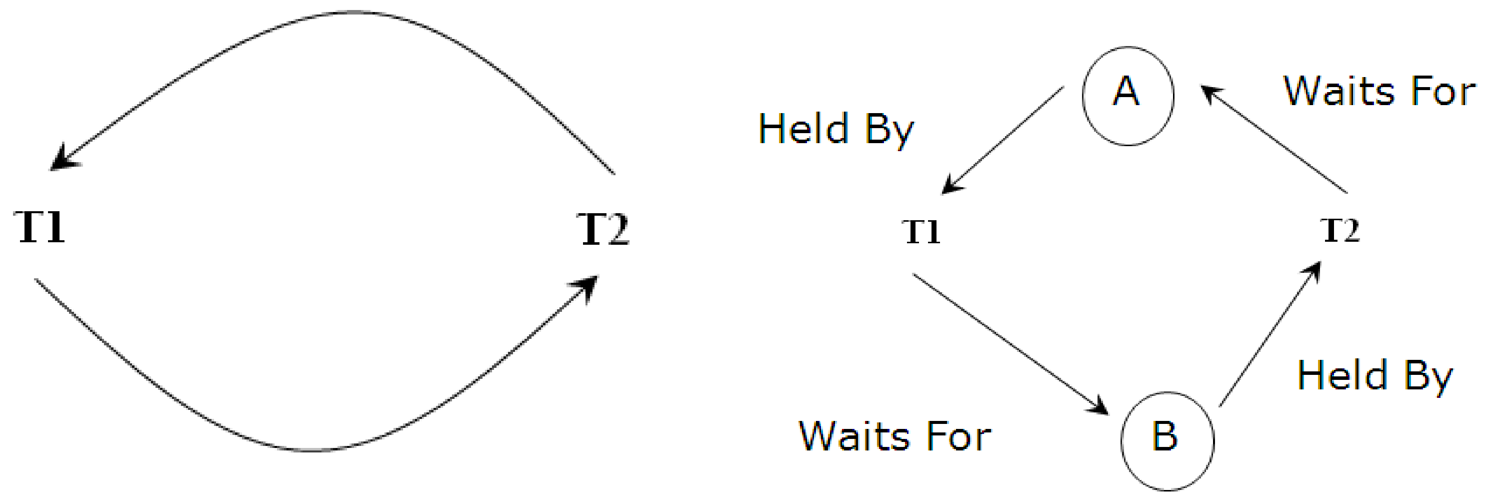
- Refers to a condition when two or more processes are each waiting for each other to release a resource, or more than two processes are waiting for resources in a circular chain.

- A common problem in multiprocessing where many processes share a lock.

Deadlock

Thread 1		Thread 2	
Operations	Locks	Operations	Locks
a.withdraw(100)	Lock on account A		
		b.withdraw(200)	Lock on account B
b.deposit(100)	Wait for T2’s lock on B		
…..		a.deposit(200)	Waits for T1’s lock on A
…..		…..	
…..		…..	

Deadlock



Deadlock Prevention

1. Lock all the objects used by a transaction when it starts.

- Simple but not very good.
- Unnecessarily restricts access to shared resources.
- Sometimes its impossible to predict which objects will be needed at the start of a transaction.
- Can result in premature locking.
- Impacts concurrency.

Deadlock Prevention

2. Timeouts.

- Commonly used.
- Each thread is give a period of time in which to execute.
- In an overloaded system the number of transactions timing out will increase.

Deadlock Prevention

3. Assign threads a priority.

- Assign a priority to the threads so that only one (or a few) thread backs up.
- The rest of the threads continue taking the locks they need as if no deadlock had occurred.
- If the priority assigned to the threads is fixed, the same threads will always be given higher priority.
- To avoid this you may assign the priority randomly whenever a deadlock is detected.

Future Reading

- Completable Futures.
- Threadsafe Data Structures.
- Semaphores.

References

Y. Daniel Liang (2017) *Intro to Java Programming and Data Structures, Comprehensive Version*. 11/E. Pearson. ISBN-13 978-0134670942 (Link)

Paul J Deitel(2016) *Java How To Program*. 10/E. ISBN-13 9780134800271 (Link)

Cay S. Horstmann (2018) Core Java SE 9 For the Impatient. 2/E. ISBN-13 978-0-13-469472-6 (Link)

http://tutorials.jenkov.com/java-concurrency/

https://winterbe.com/posts/2015/04/07/java8-concurrency-tutorial-thread-executor-examples/#callables-and-futures