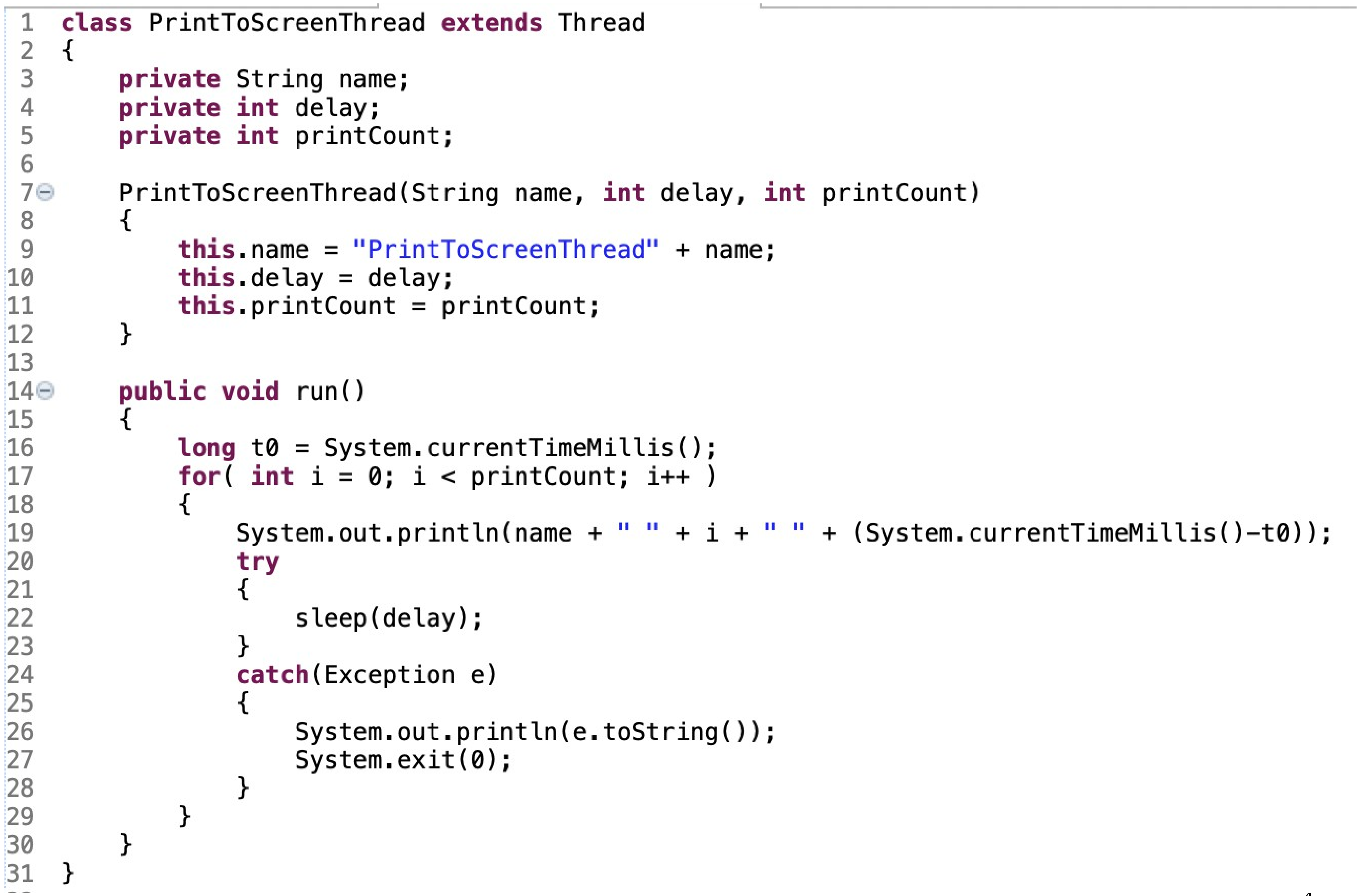
- Multi threaded applications in Java
  - Java has the Thread class to support multithreaded applications
- There are two ways to use the Thread class
  - Extend the Thread class, implementing the “public static void run()” method
    - The “public static void run()” method is where your program does its work
    - The program that is using your class calls the Thread.start() method and that calls your run() method
  - Implement the Runnable interface, implementing the “public static void run()” method
    - The “public static void run()” method is where your program does its work
    - The program that is using your class creates a Thread object wrapped around your Runnable object and then calls the Thread.start() method and that calls your run() method

- Some methods in the Thread class
  - start() - we don't implement this method, it calls our run() method
  - static sleep(long milliseconds) - pauses the code
  - static sleep(long milliseconds, int nanoseconds) - pauses the code
    - Don't assume that this implies your computer's clock/timers can support nanosecond time resolution
  - isAlive() - returns true if the Thread is still running, false if it is not
  - join() - waits for the Thread to complete
  - There are plenty of other methods
- Note if you directly call the run() method, it will execute like a non-thread method

  - That is, it will block the calling method until it completes

- Simple example (no shared data)
- We create a class that extends Thread, PrintToScreenThread, and a class that implements the Runnable interface, PrintToScreenRunnable
- Both classes simply print a message to the screen a specified number of times with a specified delay between printing the message
- Both classes have three fields
  - name – to identify which copy it is
  - delay – how long to pause between printing to the screen
  - printCount – how many times to print the message
- We should see the various messages interleaved based on the delay times for each thread

- PrintToScreenThread

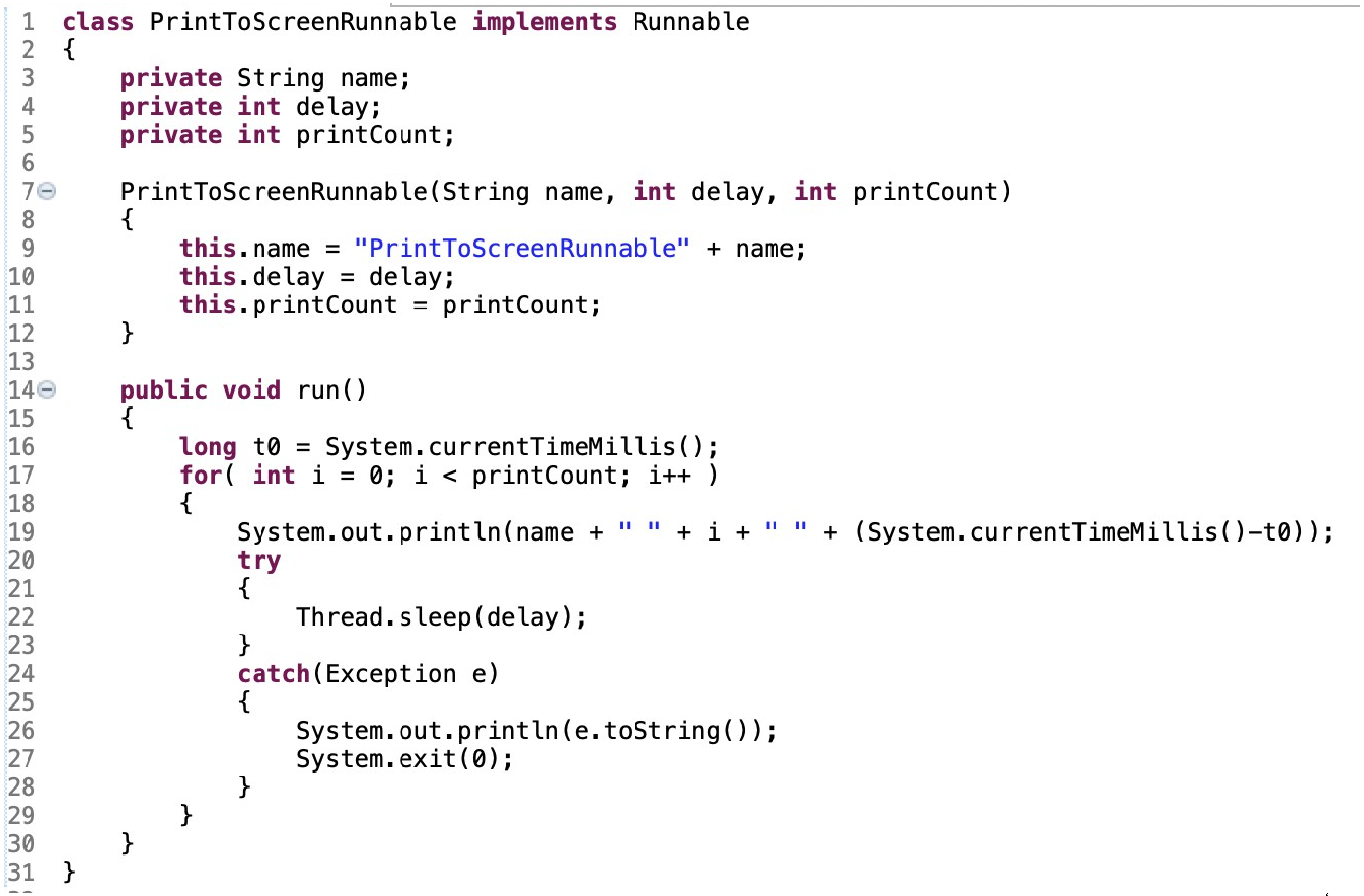


```java
class PrintToScreenThread extends Thread
{
    private String name;
    private int delay;
    private int printCount;

    PrintToScreenThread(String name, int delay, int printCount)
    {
        this.name = "PrintToScreenThread" + name;
        this.delay = delay;
        this.printCount = printCount;
    }

    public void run()
    {
        long t0 = System.currentTimeMillis();
        for( int i = 0; i < printCount; i++ )
        {
            System.out.println(name + " " + i + " " + (System.currentTimeMillis()-t0));
            try
            {
                sleep(delay);
            }
            catch(Exception e)
            {
                System.out.println(e.toString());
                System.exit(0);
            }
        }
    }
}
```

- PrintToScreenRunnable



```java
class PrintToScreenRunnable implements Runnable
{
    private String name;
    private int delay;
    private int printCount;

    PrintToScreenRunnable(String name, int delay, int printCount)
    {
        this.name = "PrintToScreenRunnable" + name;
        this.delay = delay;
        this.printCount = printCount;
    }

    public void run()
    {
        long t0 = System.currentTimeMillis();
        for( int i = 0; i < printCount; i++ )
        {
            System.out.println(name + " " + i + " " + (System.currentTimeMillis()-t0));
            try
            {
                Thread.sleep(delay);
            }
            catch(Exception e)
            {
                System.out.println(e.toString());
                System.exit(0);
            }
        }
    }
}
```

- testThread.java
  - Class to test out if our threads appear to be working
  - It takes three command line parameters
    - Number of threads and runnable threads
    - How many iterations through the loop each thread does
    - Optional – if the threads should be run as threads or non-threads
      - Execute start() or run()
  - The delay each thread uses for each iteration through the loop is increase by 100ms for each thread
    - Thread 0 has no delay
    - Thread 1 has 100ms delay
    - Thread 2 has 200ms delay
    - And so on
  - The program allocates the threads and runnables, starts them running, and finally outputs when they are done running and how long they ran for

- testThread.java
  - Execute and discuss the output
  - Briefly look at the code
  - If we execute execute the threads as non-threads, execute Thread.run() or Runnable.run(), the threads run serially instead of in parallel
    - We see that each one blocks while executing

- Sharing data between threads
  - If the shared data is created prior to the threads being instantiated and the shared data will not be updated by any class or thread, then there are no restrictions on accessing the data
    - We don't need to ensure that only once class/thread is accessing the data at any time
  - If the data is going to be accessed and updated by multiple threads at the same time, then we need to ensure that
    - Only one thread is accessing the data at any time
    - We want to ensure that when any class/thread is accessing the shared data, whether just reading or reading and writing, the entire operation is completed prior to any other class/thread accessing the shared data

- Example of shared data mistake
  - Let's say we have a multi threaded implementation of our BankAccount class (the BankAcount class just won't go away)
    - The current savings balance is $100 for Jane Smith
    - Thread A wants to deposit $10 to the savings balance of Jane Smith
    - Thread B wants to deposit $20 to the savings balance of Jane Smith
    - If they both attempt to make the update at the same time, what could happened
      - Thread A and B both read the savings balance as $100
      - Thread A updates it to $110
      - Thread B updates it to $120
      - And Jane Smith has now lost $10 due to our poor coding

- There are multiple ways to ensure only one thread is accessing shared data at a time, one way is using a semaphore
- The java.util.concurrent.Semaphore class can be used to ensure that a limited number of threads or classes can access a piece of shared data at the same time
  - The semaphore allocates a number of "permits" that are available, and keeps track of how many "permits" are available and in use
  - The simplest version is that exactly one thread/class can access the data at a time
    - Serves as a mutually exclusive lock on the data
  - We instantiate a Semaphore with a single permit with a fairness setting of true (permits are granted in first in first out order)

- java.util.concurrent.Semaphore (cont)
  - We use the acquire() method to get a permit to access the data
    - This method blocks until a permit is available
    - The tryAcquire() methods can be used for non-blocking
  - We use the release() method to release the permit once we are done accessing the shared data
  - Assuming a single permit semaphore, then when the acquire() method returns, your thread/class has exclusive access to the shared data and can read and write to it while other classes/threads wait to access it
  - Once your thread/class has completed any updates or reads of the shared data, executing the release() method allows other classes/threads to access the shared data
- There is a reduction in performance related to accessing shared data using a semapore (due to waiting for acquire())
  - Takes additional time to get access the data

- Simple example of use a Semaphore
  - We have some shared data that we need to ensure only one class/thread accesses at a time
  - We create a Semaphore with a single permit
  - We give all of the classes and threads a copy of the semaphore that need access to the shared data
  - Whenever a class/thread needs access to the data
    - Execute Semaphore.acquire() to get a permit
    - Access the data
    - Execute Semaphore.release() to release the permit
  - If we have a single permit, this scheme only allows a single class/thread access to the shared data at a time
- Note a semaphore assumes all classes/threads that access the shared data properly implement the acquire()/release() pairs

- Another way to provide shared data access is with Synchronized methods
  - Only one thread/class can execute the method of the object at a time
  - Simply add “synchronized” to the method declaration
  - According to the java documentation, https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html, for a given class, only one synchronized method can be executed at a time for a given object
    - We need to ensure that all methods that access the shared data are synchronized
- There is a reduction in performance related to accessing shared data via Synchronized methods (due to waiting for synchronized methods to execute)
  - Takes additional time to access the data

- Another way to provide shared data access is with a ReentrantLock
  - From the Java documentation, https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantLock.html
    - A reentrant mutual exclusion Lock with the same basic behavior and semantics as the implicit monitor lock accessed using synchronized methods and statements, but with extended capabilities.
  - Usage is similar to a semaphore
    - Use lock() method to get exclusive access to the lock
      - Blocks until the lock is available
    - Use unlock() method to make the lock available to others

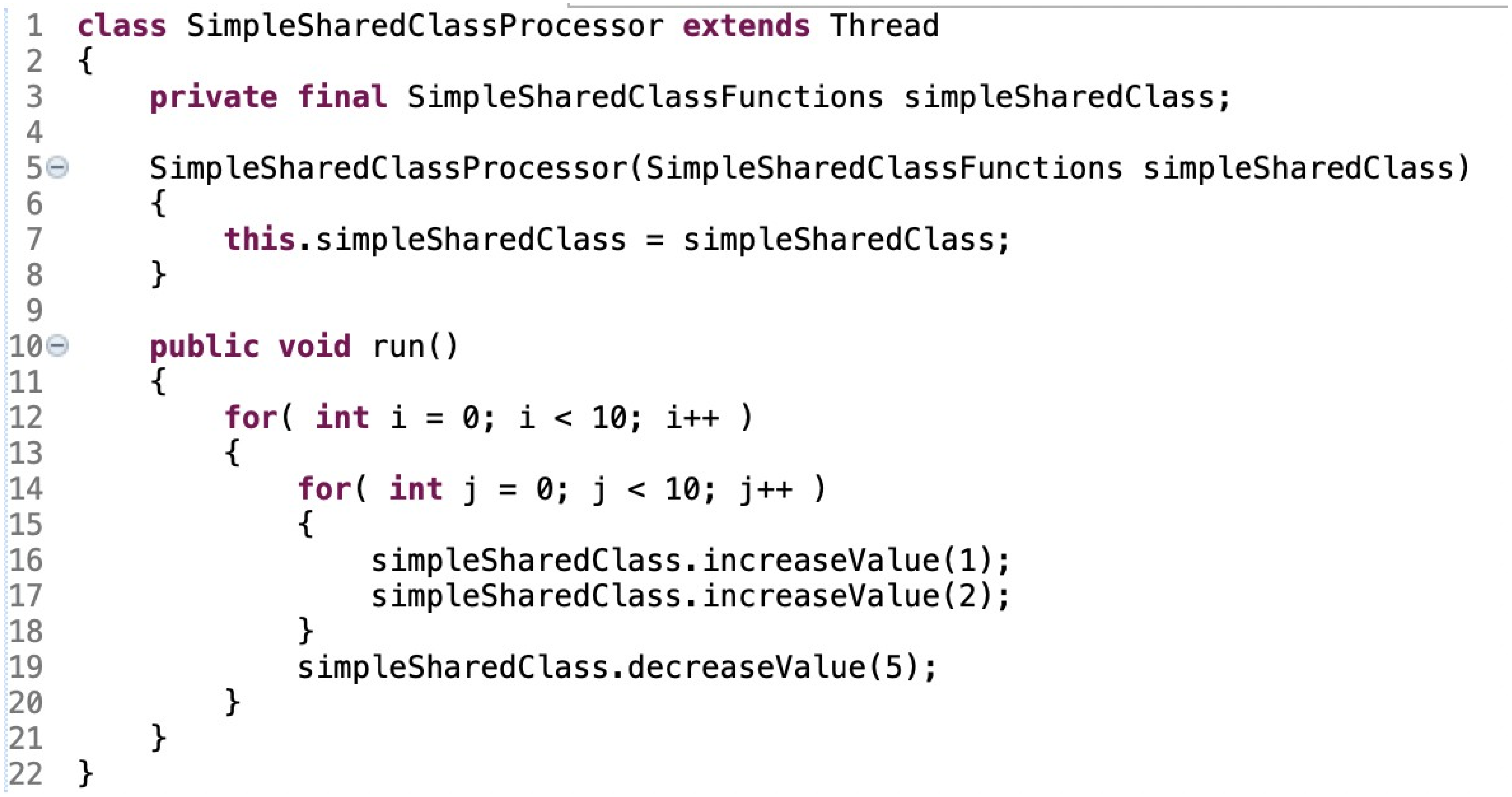
- Another way to provide shared data access is with a ReentrantLock (cont)
  - Usage is similar to a semaphore (cont)
    - Use tryLock() to get the lock immediately if it is available
      - Does not block if it is not available
    - Use tryLock(long timeout, TimeUnit unit) to get the lock within the specified time
      - Will block until timeout expires
  - The java documentation recommends putting a try/finally block immediately after the lock() method call, with the code accessing the shared data in the try block, and the unlock() method call in the finally block
- There is a reduction in performance related to accessing shared data using a lock (due to waiting for lock())

  - Takes additional time to access the data

- Sample code
  - SimpleSharedClass.java
    - Class with a single shared value, “value”, implementing the SimpleSharedClassFunctions interface, with no special handling of the shared value
  - SimpleSharedClassSemaphore.java
    - Class with a single shared value, “value”, implementing the SimpleSharedClassFunctions interface, using a semaphore to handle access to the shared value
  - SimpleSharedClassLock.java
    - Class with a single shared value, “value”, implementing the SimpleSharedClassFunctions interface, using a ReentrantLock to handle access to the shared value
  - SimpleSharedClassFunctions.java
    - Interface that specifies the functions applicable to the shared value
  - SimpleSharedClassSync.java
    - Class with a single shared value, “value”, implementing the SimpleSharedClassFunctions interface, using synchronized methods to handle access to the shared value
  - SimpleSharedClassProcessor.java
    - Class that extends Thread and in the run() methods makes updates to the shared value in the SimpleSharedClassFunctions class passed to it's constructor
  - makeSimpleSharedClass.java
    - Program that tests and shows the effect of not using a semaphore or synchronized methiods to access shared data

- Sample code (cont)
  - We have four classes, SimpleSharedClass, SimpleSharedClassSemaphore, SimpleSharedClassLock, and SimpleSharedClassSync that implement SimpleSharedClassFunctions and have a single field, “int value”, that is shared data across all threads that access the class
  - The four classes implement four different versions of “protecting” the shared data from being accessed concurrently
  - SimpleSharedClass.java
    - Class with no special handling of the shared data “value”
    - This is the wrong way to handle the shared data
  - SimpleSharedClassSemaphore.java
    - Class using a semaphore to handle access to the shared data “value”
  - SimpleSharedClassLock.java
    - Class using a lock to handle access to the shared data “value”
  - SimpleSharedClassSync.java
    - Class using synchronized methods to handle access to the shared data “value”

- SimpleSharedClassProcessor
  - This is the class that extends Thread and makes the method calls to the four versions of SimpleSharedClass to update the shared data
  - Since the simpleSharedClass is of type SimpleSharedClassFunctions, the interface that the four classes implement, all four clases are of that type



```java
class SimpleSharedClassProcessor extends Thread
{
    private final SimpleSharedClassFunctions simpleSharedClass;

    SimpleSharedClassProcessor(SimpleSharedClassFunctions simpleSharedClass)
    {
        this.simpleSharedClass = simpleSharedClass;
    }

    public void run()
    {
        for( int i = 0; i < 10; i++ )
        {
            for( int j = 0; j < 10; j++ )
            {
                simpleSharedClass.increaseValue(1);
                simpleSharedClass.increaseValue(2);
            }
            simpleSharedClass.decreaseValue(5);
        }
    }
}
```

- makeSimpleSharedClass
  - This program demonstrates how the shared data can be “corrupted” by not properly ensuring that only one thread or class is accessing the shared data at any time
  - The program does the following
    - Allocates a Semaphore for use by SimpleSharedClassSemaphore
      - `java.util.concurrent.Semaphore semaphore = new java.util.concurrent.Semaphore(1, true);`
    - Allocates a Semaphore for use by SimpleSharedClassLock
      - `java.util.concurrent.ReentrantLock lock = new java.util.concurrent.ReentrantLock(true);`

- MakeSimpleSharedClass (cont)
  - The program does the following (cont)
    - Instantiates an instance of each of the four SimpleSharedClass types {SimpleSharedClass, SimpleSharedClassSync, SimpleSharedClassSemaphore, SimpleSharedClassLock}
      - SimpleSharedClass sharedClass = new SimpleSharedClass(delayInMilliSeconds, delayInNanoSeconds);
      - SimpleSharedClassSync sharedClassSync = new SimpleSharedClassSync(delayInMilliSeconds, delayInNanoSeconds);
      - SimpleSharedClassSemaphore sharedClassSemaphore = new SimpleSharedClassSemaphore(delayInMilliSeconds, delayInNanoSeconds, semaphore);
      - SimpleSharedClassLock sharedClassLock = new SimpleSharedClassSemaphore(delayInMilliSeconds, delayInNanoSeconds, lock);

- MakeSimpleSharedClass (cont)
  - The program does the following (cont)
    - Allocates four SimpleSharedClassProcessor arrays of size numberOfThreads
      - SimpleSharedClassProcessor[] simpleSharedClassProcessor = new SimpleSharedClassProcessor[numberOfThreads];
      - SimpleSharedClassProcessor[] simpleSharedClassProcessorSync = new SimpleSharedClassProcessor[numberOfThreads];
      - SimpleSharedClassProcessor[] simpleSharedClassProcessorSemaphore = new SimpleSharedClassProcessor[numberOfThreads];
      - SimpleSharedClassProcessor[] simpleSharedClassProcessorLock = new SimpleSharedClassProcessor[numberOfThreads];

- MakeSimpleSharedClass (cont)
  - The program does the following (cont)
    - Instantiates each array element for each of the four SimpleSharedClassProcessor arrays, with each array element of each of the four arrays sharing a copy of the appropriate object

```
for( int i = 0; i < simpleSharedClassProcessor.length; i++ )
{
    simpleSharedClassProcessor[i] = new
    SimpleSharedClassProcessor(sharedClass);
    simpleSharedClassProcessorSync[i] = new
    SimpleSharedClassProcessor(sharedClassSync);
    simpleSharedClassProcessorSemaphore[i] = new
    SimpleSharedClassProcessor(sharedClassSemaphore);
    simpleSharedClassProcessorLock[i] = new
    SimpleSharedClassProcessor(sharedClassLock);
}
```

    - Each array element of each of the four arrays is wrapped around the same class/object

- MakeSimpleSharedClass (cont)
  - The program does the following (cont)
    - Start each Thread running

```
for( int i = 0; i < simpleSharedClassProcessor.length; i++ )
{
    simpleSharedClassProcessor[i].start();
    simpleSharedClassProcessorSync[i].start();
    simpleSharedClassProcessorSemaphore[i].start();
    simpleSharedClassProcessorLock[i].start();
}
```

    - And then waits for each of the threads to complete running
      - The code is checking to see when all of the copies of each array all return false to isAlive() for the first time
      - When the above first happens, the program outputs that the array has completed

- MakeSimpleSharedClass (cont)
  - Running and the expected output
  - Since SimpleSharedClassProcessor increases “value” by 1 and 2 ten times in the inner loop, and decrease it by 5 in the outer loop, and the outer loop executes 10 times
  - value increase by (1 + 2) x 10 = 30 in the inner loop
  - value decreases by 5 in the outer loop
  - The outer loop is executed 10 times, so value is incresed by 10 x 30 and decreased by 10 x 5
  - So, value is increased by 300 – 50 = 250 in each Thread for each of {SimpleSharedClass, SimpleSharedClassSync, SimpleSharedClassSemaphore, SimpleSharedClassLock}
  - Since there are 10 threads per class, that means each class should end with value = 2500
  - Running is simple “java SimpleSharedClassProcessor”

- MakeSimpleSharedClass (cont)
  - Output from “java SimpleSharedClassProcessor”

```
the correct output should be value = 2500
sharedClass.toString() value = 241 (266ms)
sharedClassSync.toString() value = 2500 (2648ms)
sharedClassLock.toString() value = 2500 (2654ms)
sharedClassSemaphore.toString() value = 2500 (2654ms)
```

- MakeSimpleSharedClass (cont)
  - What the output is telling us
    - sharedClass.toString() value = 241 (266ms)
      - The 10 threads associated with SimpleSharedClass took 266ms to execute, and ended with value = 241
      - We got the wrong result, but we got it pretty quickly
    - sharedClassSync.toString() value = 2500 (2648ms)
      - The 10 threads associated with SimpleSharedClassSync took 2648ms to execute, and ended with value = 2500
      - We got the correct result, but it took a little longer than getting the incorrect result
    - sharedClassLock.toString() value = 2500 (2654ms)
      - The 10 threads associated with SimpleSharedClassSync took 2654ms to execute, and ended with value = 2500
      - We got the correct result, but it took a little longer than getting the incorrect result
    - sharedClassSemaphore.toString() value = 2500 (2654ms)
      - The 10 threads associated with SimpleSharedClassSync took 2654ms to execute, and ended with value = 2500
      - We got the correct result, but it took a little longer than getting the incorrect result

- Some final thoughts
  - Using synchronized methods is quite easy to use, simply add “synchronized” to the methods that access the shared data
    - This will block access to the other synchonized methods for the entire time that the method is executing
  - Using a Semaphore we can have multiple methods executing that access the shared data, but the portion that access and uses the shared data needs to use the Semaphore to ensure only one thread/class is accessing the shared data at a time
    - The code needs to correctly implement the acquire/release pairs and enforce not accessing the shared data without a “permit”

- Some final thoughts (cont)
  - Using a ReentrantLock we can have multiple methods executing that access the shared data, but the portion that access and uses the shared data needs to use the ReentrantLock to ensure only one thread/class is accessing the shared data at a time
    - The code needs to correctly implement the lock/unlock pairs and enforce not accessing the shared data without the lock

- Summary of what we did
  - For each of {SimpleSharedClass, SimpleSharedClassSync, SimpleSharedClassSemaphore, SimpleSharedClassLock}
    - We created 10 threads that access the shared data “value”
    - Each thread modified “value” using the increaseValue() and decreaseValue() methods
    - Since SimpleSharedClass did not do anything to ensure only one thread modified “value” at a time, at the end of execution, “value” had the incorrect value
    - Since SimpleSharedClassSync, SimpleSharedClassSemaphore, and SimpleSharedClassLock used synchronized methods, a semaphore, and a reentrant lock to ensure only one thread accessed “value” at any time, at the end of execution, “value” had the correct value

- Some other thoughts/questions
  - What would happen if a synchronized method in class A calls another synchronized method in class A?
    - What if it is a different instantiation of A?
  - What would happen if a synchronized method in class A calls a synchronized method in class B?
    - What if a synchronized method was alreading executing in class B?
    - What if the synchronized method in class B was waiting for a different synchronized method in class A to complete?
  - What if just one method in class A that accesses shared data doesn't acquire a “permit” prior to accessing the shared data?

- java.util.concurrent.locks
  - The java documentation, https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/package-summary.html, has additional information on locks
- java.util.concurrent.atomic
  - The java documentation, https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html, has information on classes that support single variable multiple thread access without using locks