Concurrency

Objectives

After completing this lesson, you should be able to:

- Describe operating system task scheduling
- Create worker threads using `Runnable` and `Callable`
- Use an `ExecutorService` to concurrently execute tasks
- Identify potential threading problems
- Use `synchronized` and the classes in the `java.concurrent.atomic` package to manage atomicity
- Use monitor locks to control the order of thread execution
- Use the `java.util.concurrent` collections

Task Scheduling

Modern operating systems use preemptive multitasking to allocate CPU time to applications. There are two types of tasks that can be scheduled for execution:

- **Processes:** A process is an area of memory that contains both code and data. A process has a thread of execution that is scheduled to receive CPU time slices.
- **Thread:** A thread is a scheduled execution of a process. Concurrent threads are possible. All threads for a process share the same data memory but may be following different paths through a code section.

Preemptive Multitasking

Modern computers often have more tasks to execute than CPUs. Each task is given an amount of time (called a time slice) during which it can execute on a CPU. A time slice is usually measured in milliseconds. When the time slice has elapsed, the task is forcefully removed from the CPU, and another task is given a chance to execute.

Legacy `Thread` and `Runnable`

Prior to Java 5, the `Thread` class was used to create and start threads. Code to be executed by a thread is placed in a class, which does either of the following:

- Extends the `Thread` class
  - Simpler code
- Implements the `Runnable` interface
  - More flexible
  - `extends` is still free.

Extending `Thread`

Extend `java.lang.Thread` and override the `run` method:

```
public class ExampleThread extends Thread {
    @Override
    public void run() {
        for(int i = 0; i < 10; i++) {
            System.out.println("i:" + i);
        }
    }
}
```

The `run` Method

The code to be executed in a new thread of execution should be placed in a `run` method. You should avoid calling the `run` method directly. Calling the `run` method does not start a new thread, and the effect would be no different than calling any other method.

Implementing `Runnable`

Implement `java.lang.Runnable` and implement the `run` method:

```
public class ExampleRunnable implements Runnable {
    private final String name;

    public ExampleRunnable(String name) {
        this.name = name;
    }

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(name + ":" + i);
        }
    }
}
```

The `run` Method

Just as when extending `Thread`, calling the `run` method does not start a new thread. The benefit of implementing `Runnable` is that you may still extend a class of your choosing.

The `java.util.concurrent` Package

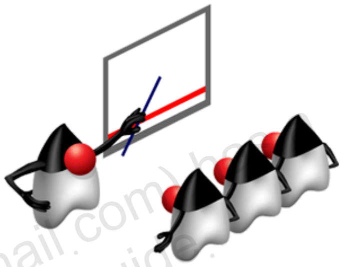
Java 5 introduced the `java.util.concurrent` package, which contains classes that are useful in concurrent programming. Features include:

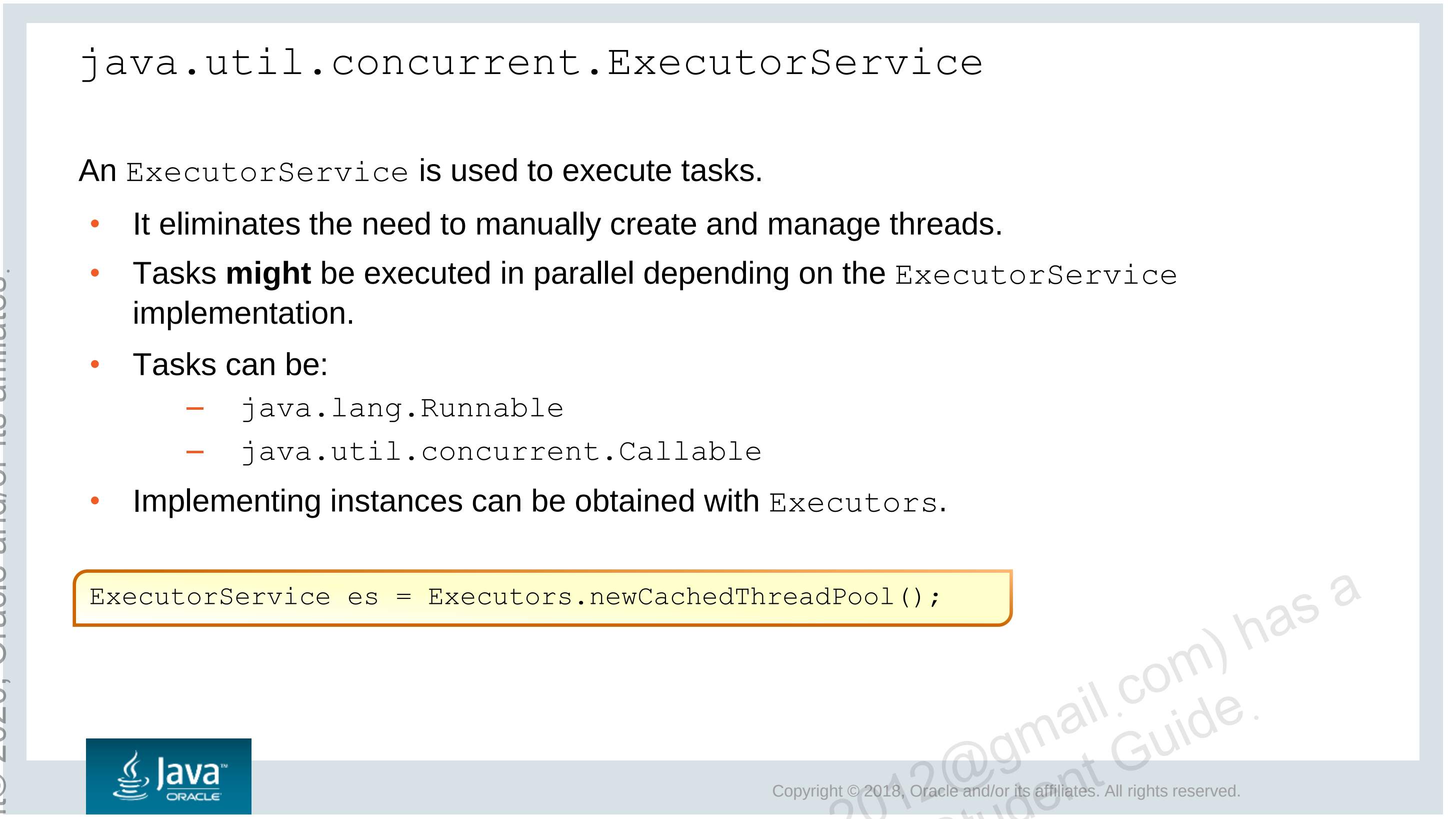
- Concurrent collections
- Synchronization and locking alternatives
- Thread pools
  - Fixed and dynamic thread count pools available
  - Parallel divide and conquer (Fork-Join) new in Java 7

Recommended Threading Classes

Traditional `Thread` related APIs are difficult to code properly. Recommended concurrency classes include:

- `java.util.concurrent.ExecutorService`, a higher level mechanism used to execute tasks
  - It may create and reuse `Thread` objects for you.
  - It allows you to submit work and check on the results in the future.
- The Fork-Join framework, a specialized work-stealing `ExecutorService` new in Java 7





java.util.concurrent.ExecutorService

An `ExecutorService` is used to execute tasks.

- It eliminates the need to manually create and manage threads.
- Tasks **might** be executed in parallel depending on the `ExecutorService` implementation.
- Tasks can be:
  - `java.lang.Runnable`
  - `java.util.concurrent.Callable`
- Implementing instances can be obtained with `Executors`.

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

The Behavior of an ExecutorService

A cached thread pool `ExecutorService`:

- Creates new threads as needed
- Reuses its threads (Its threads do not die after finishing their task)
- Terminates threads that have been idle for 60 seconds

Other types of `ExecutorService` implementations are available:

```
int cpuCount = Runtime.getRuntime().availableProcessors();
ExecutorService es = Executors.newFixedThreadPool(cpuCount);
```

A fixed thread pool `ExecutorService`:

- Contains a fixed number of threads
- Reuses its threads (Its threads do not die after finishing their task)
- Queues up work until a thread is available
- Could be used to avoid overworking a system with CPU-intensive tasks

Example ExecutorService

This example illustrates using an `ExecutorService` to execute `Runnable` tasks:

```
package com.example;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ExecutorExample {
    public static void main(String[] args) {
        ExecutorService es = Executors.newCachedThreadPool();
        es.execute(new ExampleRunnable("one"));
        es.execute(new ExampleRunnable("two"));
        es.shutdown();
    }
}
```

Execute this Runnable task sometime in the future

Shut down the executor

Shutting Down an `ExecutorService`

Shutting down an `ExecutorService` is important because its threads are nondaemon threads and will keep your JVM from shutting down.

```
es.shutdown();

try {
    es.awaitTermination(5, TimeUnit.SECONDS);
} catch (InterruptedException ex) {
    System.out.println("Stopped waiting early");
}
```

Stop accepting new `Callable`s.

If you want to wait for the `Callable`s to finish

java.util.concurrent.Callable

The `Callable` interface:

- Defines a task submitted to an `ExecutorService`
- Is similar in nature to `Runnable`, but can:
  - Return a result using generics
  - Throw a checked exception

```
package java.util.concurrent;
public interface Callable<V> {
    V call() throws Exception;
}
```

Example `Callable` Task

```
public class ExampleCallable implements Callable {

  private final String name;
  private final int len;
  private int sum = 0;

  public ExampleCallable(String name, int len) {
    this.name = name;
    this.len = len;
  }

  @Override
  public String call() throws Exception {
    for (int i = 0; i < len; i++) {
      System.out.println(name + ":" + i);
      sum += i;
    }
    return "sum: " + sum;
  }
}
```

Return a String from this task: the sum of the series

java.util.concurrent.Future

The `Future` interface is used to obtain the results from a `Callable`'s `V call()` method.

```
Future<V> future = es.submit(callable);
//submit many callables
try {
    V result = future.get();
} catch (ExecutionException|InterruptedException ex) {

}
```

`ExecutorService` controls when the work is done.

Gets the result of the `Callable`'s `call` method (blocks if needed).

If the `Callable` threw an `Exception`

Waiting on a Future

Because the call to `Future.get()` will block, you must do one of the following:

- Submit all your work to the `ExecutorService` before calling any `Future.get()` methods.
- Be prepared to wait for that `Future` to obtain the result.
- Use a nonblocking method such as `Future.isDone()` before calling `Future.get()` or use `Future.get(long timeout, TimeUnit unit)`, which will throw a `TimeoutException` if the result is not available within a given duration.

Example

```
public static void main(String[] args) {

  ExecutorService es = Executors.newFixedThreadPool(4);
  Future<String> f1 = es.submit(new ExampleCallable("one",10));
  Future<String> f2 = es.submit(new ExampleCallable("two",20));

  try {
    es.shutdown();
    es.awaitTermination(5, TimeUnit.SECONDS);
    String result1 = f1.get();
    System.out.println("Result of one: " + result1);
    String result2 = f2.get();
    System.out.println("Result of two: " + result2);
  } catch (ExecutionException | InterruptedException ex) {
    System.out.println("Exception: " + ex);
  }

}
```

Wait 5 seconds for the tasks to complete

Get the results of tasks f1 and f2

Threading Concerns

- Thread Safety
  - Classes should continue to behave correctly when accessed from multiple threads.
- Performance: Deadlock and livelock
  - Threads typically interact with other threads. As more threads are introduced into an application, the possibility exists that threads will reach a point where they cannot continue.

Thread safety is really about the ability of a class to perform the same way when it is accessed by one thread or multiple threads. Fundamentally, a class performs actions and holds data. Using this definition of thread safety, a class is thread safe if the actions the class performs and the data stored are consistent when used accessed by multiple threads.

Deadlock is a situation where thread A is blocked waiting for a condition set by thread B, but thread B is also blocked waiting for a condition set by thread A.

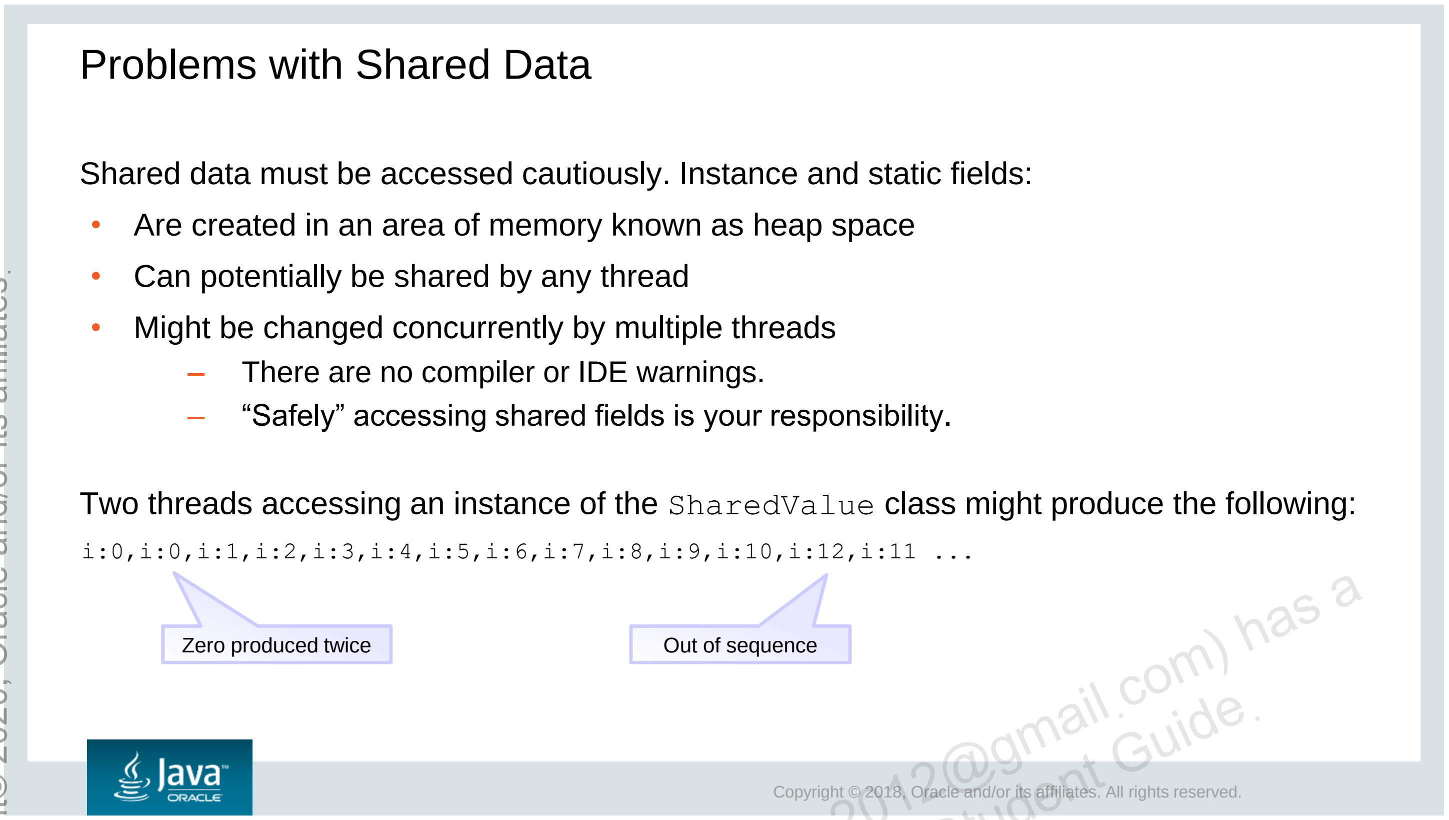
Livelock is a condition where a thread is not blocked, but cannot move forward because an operation it continually retries fails. Livelock is related to another condition, starvation, where a thread attempts to access a resource that it can never access—likely because other higher priority threads are continually accessing the resource.

Shared Data

Static and instance fields are potentially shared by threads.

```
public class SharedValue {
    private int i;

    // Return a unique value
    public int getNext() {
        return i++;
    }
}
```

Potentially shared variable



Problems with Shared Data

Shared data must be accessed cautiously. Instance and static fields:

- Are created in an area of memory known as heap space
- Can potentially be shared by any thread
- Might be changed concurrently by multiple threads
  - There are no compiler or IDE warnings.
  - “Safely” accessing shared fields is your responsibility.

Two threads accessing an instance of the `SharedValue` class might produce the following:

```
i:0,i:0,i:1,i:2,i:3,i:4,i:5,i:6,i:7,i:8,i:9,i:10,i:12,i:11 ...
```

Zero produced twice

Out of sequence

Debugging Threads

Debugging threads can be difficult because the frequency and duration of time each thread is allocated can vary for many reasons including:

- Thread scheduling is handled by an operating system, and operating systems may use different scheduling algorithms
- Machines have different counts and speeds of CPUs
- Other applications may be placing load on the system

This is one of those cases where an application may seem to function perfectly while in development, but strange problems might manifest after it is in production because of scheduling variations. It is your responsibility to safeguard access to shared variables.

Nonshared Data

Some variable types are never shared. The following types are always thread-safe:

- Local variables
- Method parameters
- Exception handler parameters
- Immutable data

Shared Thread-Safe Data

Any shared data that is immutable, such as `String` objects or final fields, are thread-safe because they can only be read and not written.

Atomic Operations

Atomic operations function as a single operation. A single statement in the Java language is not always atomic.

- `i++;`
  - Creates a temporary copy of the value in `i`
  - Increments the temporary copy
  - Writes the new value back to `i`
- `l = 0xffff_ffff_ffff_ffff;`
  - 64-bit variables might be accessed using two separate 32-bit operations.

What inconsistencies might two threads incrementing the same field encounter?

What if that field is long?

Inconsistent Behavior

One possible problem with two threads incrementing the same field is that a lost update might occur. Imagine if both threads read a value of 41 from a field, increment the value by one, and then write their results back to the field. Both threads will have done an increment, but the resulting value is only 42. Depending on how the Java Virtual Machine is implemented and the type of physical CPU being used, you may never or rarely see this behavior. However, you must always assume that it could happen.

If you have a long value of `0x0000_0000_ffff_ffff` and increment it by `1`, the result should be `0x0000_0001_0000_0000`. However, because it is legal for a 64-bit field to be accessed using two separate 32-bit writes, there could temporarily be a value of `0x0000_0001_ffff_ffff` or even `0x0000_0000_0000_0000` depending on which bits are modified first. If a second thread was allowed to read a 64-bit field while it was being modified by another thread, an incorrect value could be retrieved.

Out-of-Order Execution

- Operations performed in one thread may not appear to execute in order if you observe the results from another thread.
  - Code optimization may result in out-of-order operation.
  - Threads operate on cached copies of shared variables.
- To ensure consistent behavior in your threads, you must synchronize their actions.
  - You need a way to state that an action happens before another.
  - You need a way to flush changes to shared variables back to main memory.

Synchronizing Actions

Every thread has a *working memory* in which it keeps its own *working copy* of variables that it must use or assign. As the thread executes a program, it operates on these working copies. There are several actions that will synchronize a thread's *working memory* with main memory:

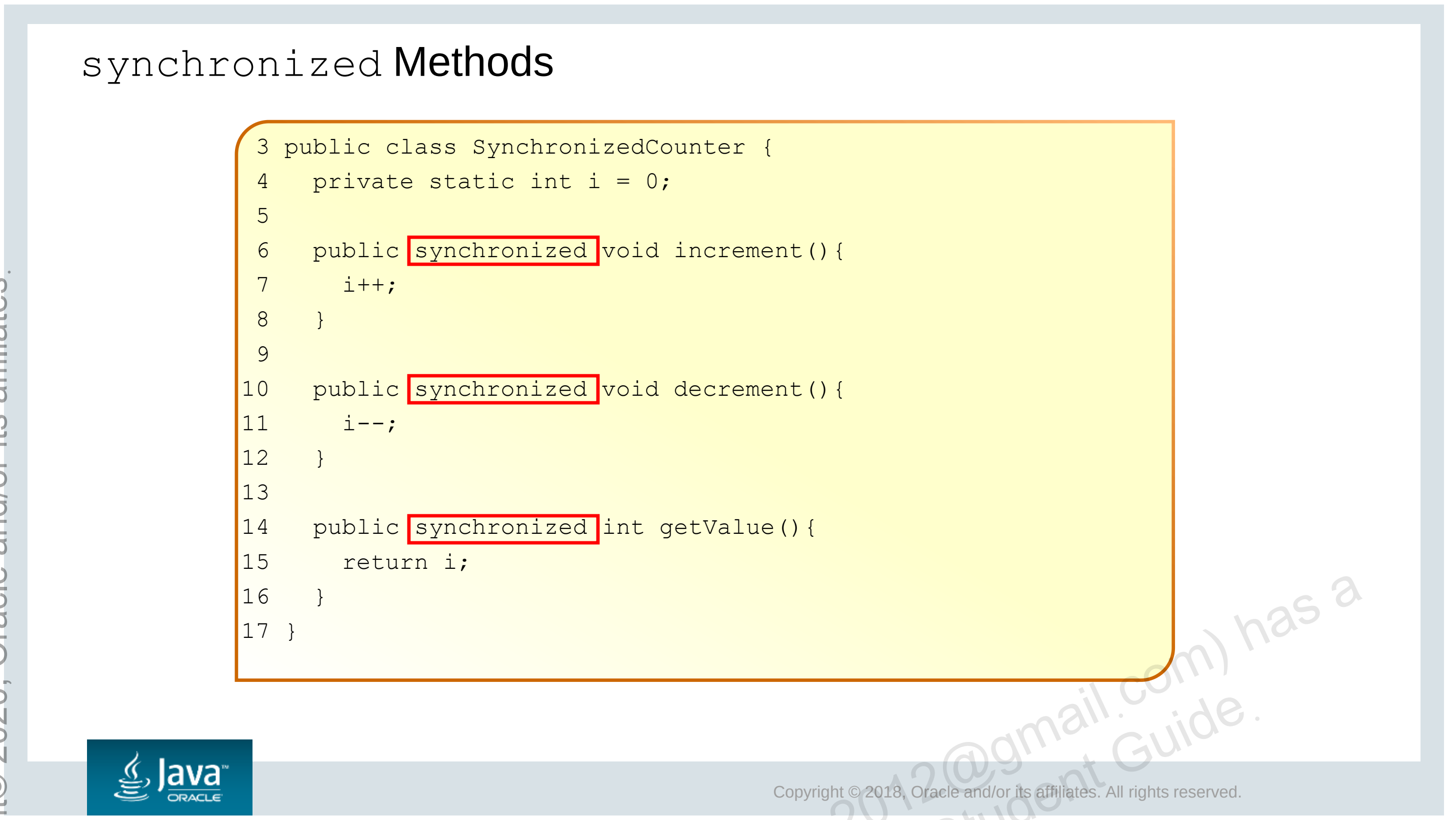
- A volatile read or write of a variable (the `volatile` keyword)
- Locking or unlocking a monitor (the `synchronized` keyword)
- The first and last action of a thread
- Actions that start a thread or detect that a thread has terminated

The `synchronized` Keyword

The `synchronized` keyword is used to create thread-safe code blocks. A `synchronized` code block:

- Causes a thread to write all of its changes to main memory when the end of the block is reached
- Is used to group blocks of code for exclusive execution
  - Threads block until they can get exclusive access
  - Solves the atomic problem

Synchronized code blocks are used to ensure that data that is not thread-safe will not be accessed concurrently by multiple threads.



`synchronized` Methods

```
public class SynchronizedCounter {
  private static int i = 0;

  public synchronized void increment(){
    i++;
  }

  public synchronized void decrement(){
    i--;
  }

  public synchronized int getValue(){
    return i;
  }
}
```

Synchronized Method Behavior

In the example in the slide, you can call only one method at a time in a `SynchronizedCounter` object because all its methods are `synchronized`. In this example, the synchronization is per `SynchronizedCounter`. Two `SynchronizedCounter` instances could be used concurrently.

If the methods were not `synchronized`, calling `decrement` while `getValue` is accessed might result in unpredictable behavior.

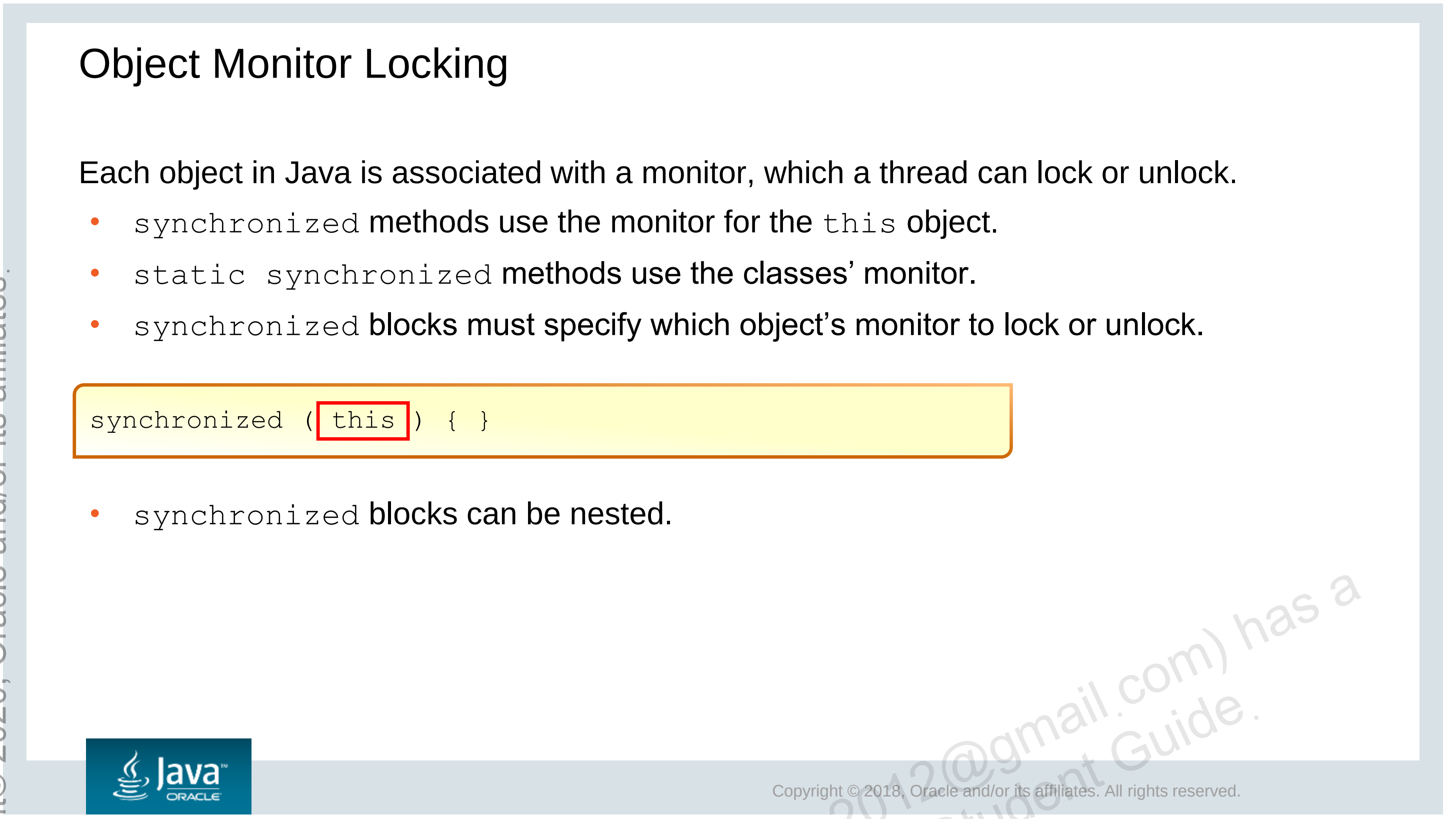
`synchronized` Blocks

```
public void run(){
  for (int i = 0; i < countSize; i++){
    synchronized(this){
      count.increment();
      System.out.println(threadName
           + " Current Count: " + count.getValue());
    }
  }
}
```

Synchronization Bottlenecks

Synchronization in multithreaded applications ensures reliable behavior. Because `synchronized` blocks and methods are used to restrict a section of code to a single thread, you are potentially creating performance bottlenecks. `synchronized` blocks can be used in place of `synchronized` methods to reduce the number of lines that are exclusive to a single thread.

Use synchronization as little as possible for performance, but as much as needed to guarantee reliability.



Object Monitor Locking

Each object in Java is associated with a monitor, which a thread can lock or unlock.

- `synchronized` methods use the monitor for the `this` object.
- `static synchronized` methods use the classes' monitor.
- `synchronized` blocks must specify which object's monitor to lock or unlock.

```
synchronized ( this ) { }
```

- `synchronized` blocks can be nested.

Nested **`synchronized`** Blocks

A thread can lock multiple monitors simultaneously by using nested `synchronized` blocks.

Threading Performance

To execute a program as quickly as possible, you must avoid performance bottlenecks. Some of these bottlenecks are:

- Resource Contention: Two or more tasks waiting for exclusive use of a resource
- Blocking I/O operations: Doing nothing while waiting for disk or network data transfers
- Underutilization of CPUs: A single-threaded application uses only a single CPU

Multithreaded Servers

Even if you do not write code to create new threads of execution, your code might be run in a multithreaded environment. You must be aware of how threads work and how to write thread-safe code. When creating code to run inside of another piece of software (such as a middleware or application server), you must read the product's documentation to discover whether threads will be created automatically. For instance, in a Java EE application server, there is a component called a Servlet that is used to handle HTTP requests. Servlets must always be thread-safe because the server starts a new thread for each HTTP request.

Performance Issue: Examples

- **Deadlock** results when two or more threads are blocked forever, waiting for each other.

```
synchronized(obj1) {
   synchronized(obj2) {
   }
}
```

Thread 1 pauses after locking obj1’s monitor.

```
synchronized(obj2) {
   synchronized(obj1) {
   }
}
```

Thread 2 pauses after locking obj2’s monitor.

- **Starvation** and **Livelock**

Starvation and livelock are much less common a problem than deadlock, but are still problems that every designer of concurrent software is likely to encounter.

Starvation

Starvation describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress. This happens when shared resources are made unavailable for long periods by “greedy” threads. For example, suppose an object provides a synchronized method that often takes a long time to return. If one thread invokes this method frequently, other threads that also need frequent synchronized access to the same object will often be blocked.

Livelock

A thread often acts in response to the action of another thread. If the other thread’s action is also a response to the action of another thread, *livelock* may result. As with deadlock, livelocked threads are unable to make further progress. However, the threads are not blocked; they are simply too busy responding to each other to resume work.

`java.util.concurrent` Classes and Packages

The `java.util.concurrent` package contains a number of classes that help with your concurrent applications. Here are just a few examples.

- `java.util.concurrent.atomic` package
  - Lock free thread-safe variables
- `CyclicBarrier`
  - A class that blocks until a specified number of threads are waiting for the thread to complete.
- Concurrency collections

The use of synchronized code blocks can result in performance bottlenecks. Several components of the `java.util.concurrent` package provide alternatives to using synchronized code blocks.

The `java.util.concurrent.atomic` Package

The `java.util.concurrent.atomic` package contains classes that support lock-free thread-safe programming on single variables.

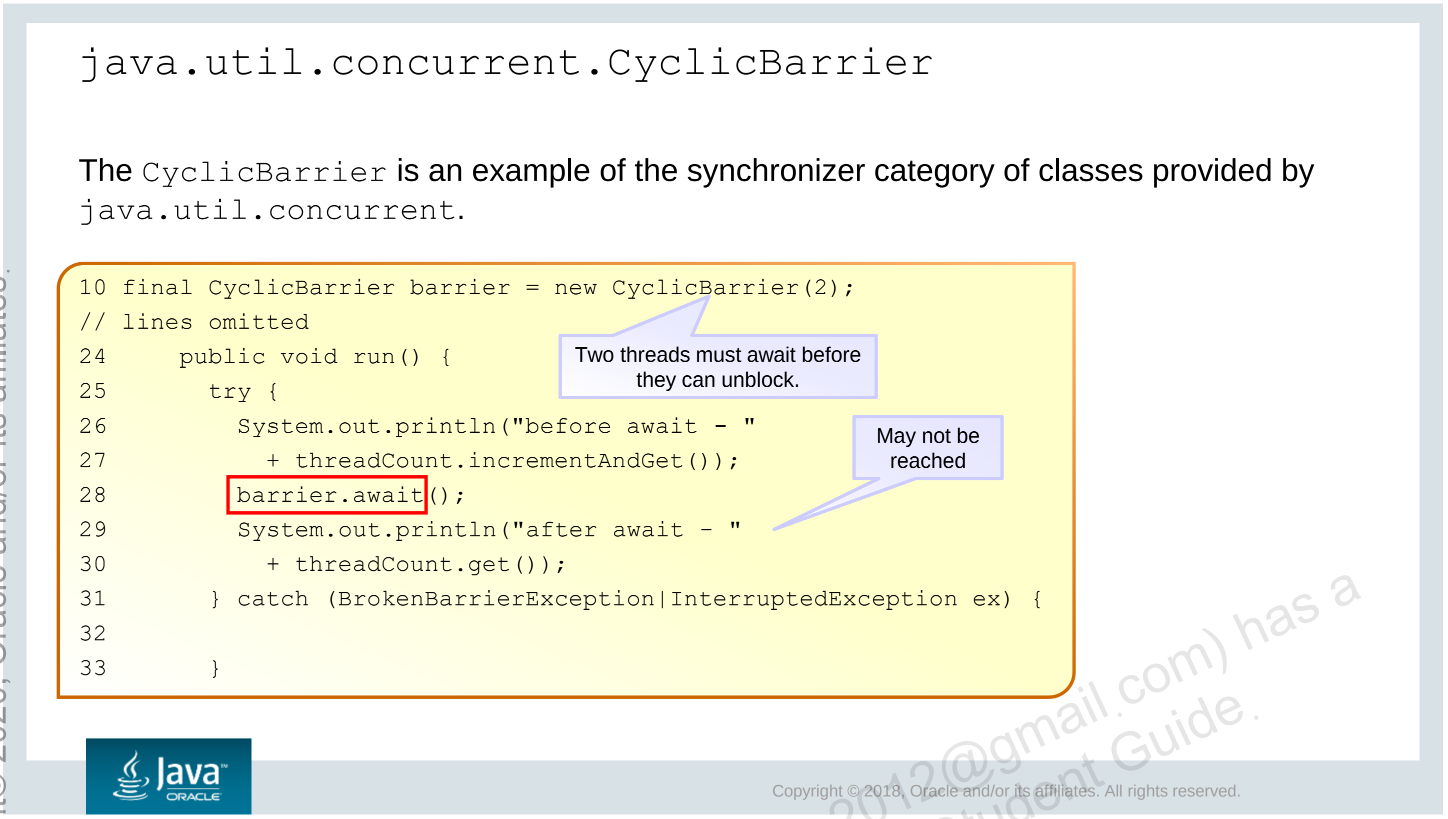
```
    public static void main(String[] args) {
        AtomicInteger ai = new AtomicInteger(5);
        System.out.println("New value: "
          + ai.incrementAndGet());
        System.out.println("New value: "
          + ai.getAndIncrement());
        System.out.println("New value: "
          + ai.getAndIncrement());

    }
```

An atomic operation increments value to 6 and returns the value.

There is no need to use the `synchronized` keyword with atomic variables. Methods exist to increment a value before or after the value is returned.

The output is:

```
New value: 6
New value: 6
New value: 7
```



java.util.concurrent.CyclicBarrier

The CyclicBarrier is an example of the synchronizer category of classes provided by java.util.concurrent.

```
final CyclicBarrier barrier = new CyclicBarrier(2);
// lines omitted
    public void run() {
      try {
        System.out.println("before await - "
          + threadCount.incrementAndGet());
        barrier.await();
        System.out.println("after await - "
          + threadCount.get());
      } catch (BrokenBarrierException|InterruptedException ex) {

      }
```

Two threads must await before they can unblock.

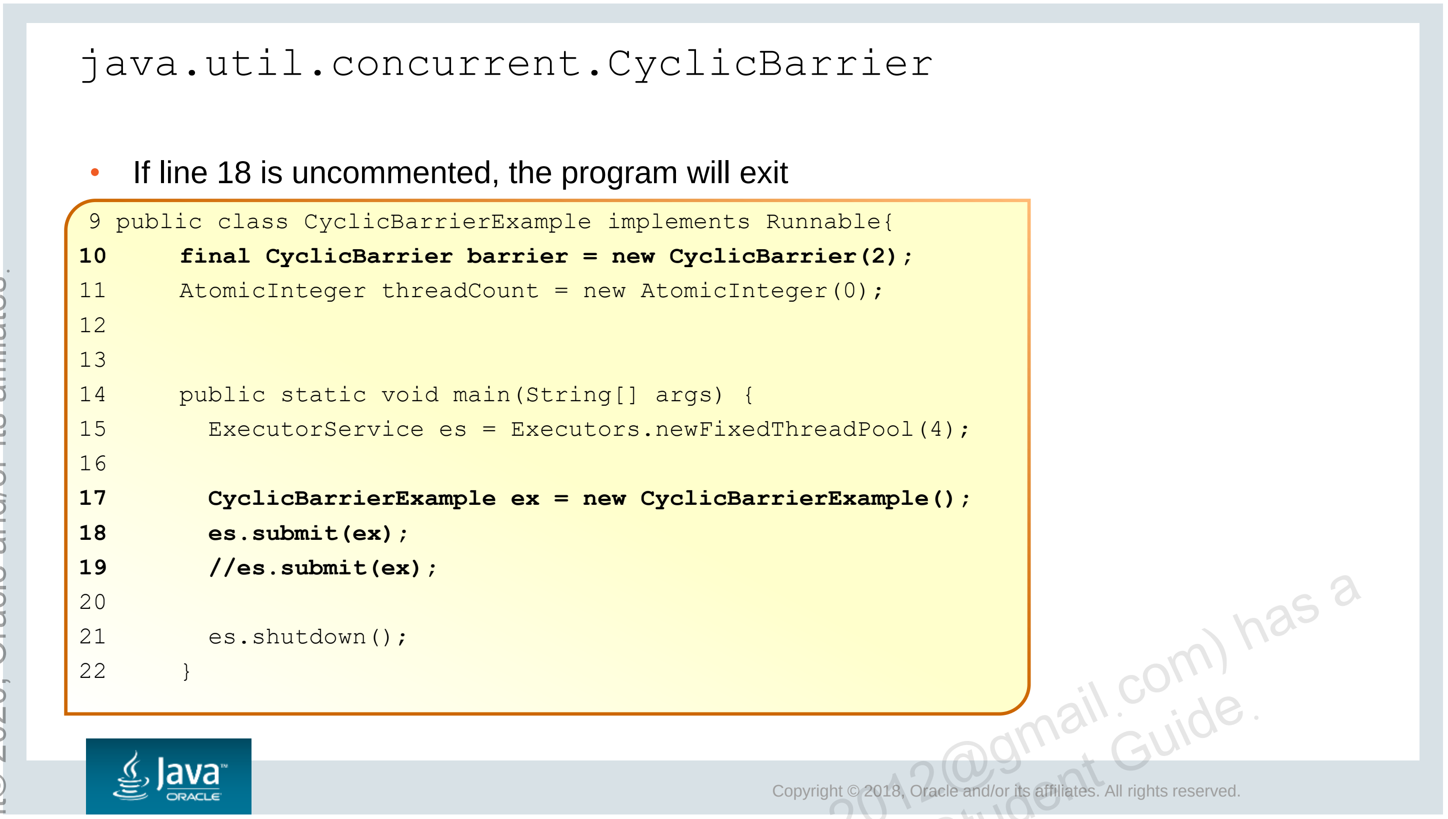
May not be reached

CyclicBarrier Behavior

In this example, if only one thread calls await() on the barrier, that thread may block forever. After a second thread calls await(), any additional call to await() will again block until the required number of threads is reached. A CyclicBarrier contains a method, await(long timeout, TimeUnit unit), which will block for a specified duration and throw a TimeoutException if that duration is reached.

Synchronizers

A framework of classes in the java.util.concurrent package that provide mechanics for atomically managing synchronization state, blocking and unblocking threads, and queuing. The CyclicBarrier class is an example.



java.util.concurrent.CyclicBarrier

- If line 18 is uncommented, the program will exit

```
 9 public class CyclicBarrierExample implements Runnable{
10     final CyclicBarrier barrier = new CyclicBarrier(2);
11     AtomicInteger threadCount = new AtomicInteger(0);
12
13
14     public static void main(String[] args) {
15       ExecutorService es = Executors.newFixedThreadPool(4);
16
17       CyclicBarrierExample ex = new CyclicBarrierExample();
18       es.submit(ex);
19       //es.submit(ex);
20
21       es.shutdown();
22     }
```

If the main method runs as shown, the application will just wait. If line 18 is uncommented, then the program will exit normally.

Thread-Safe Collections

The `java.util` collections are not thread-safe. To use collections in a thread-safe fashion:

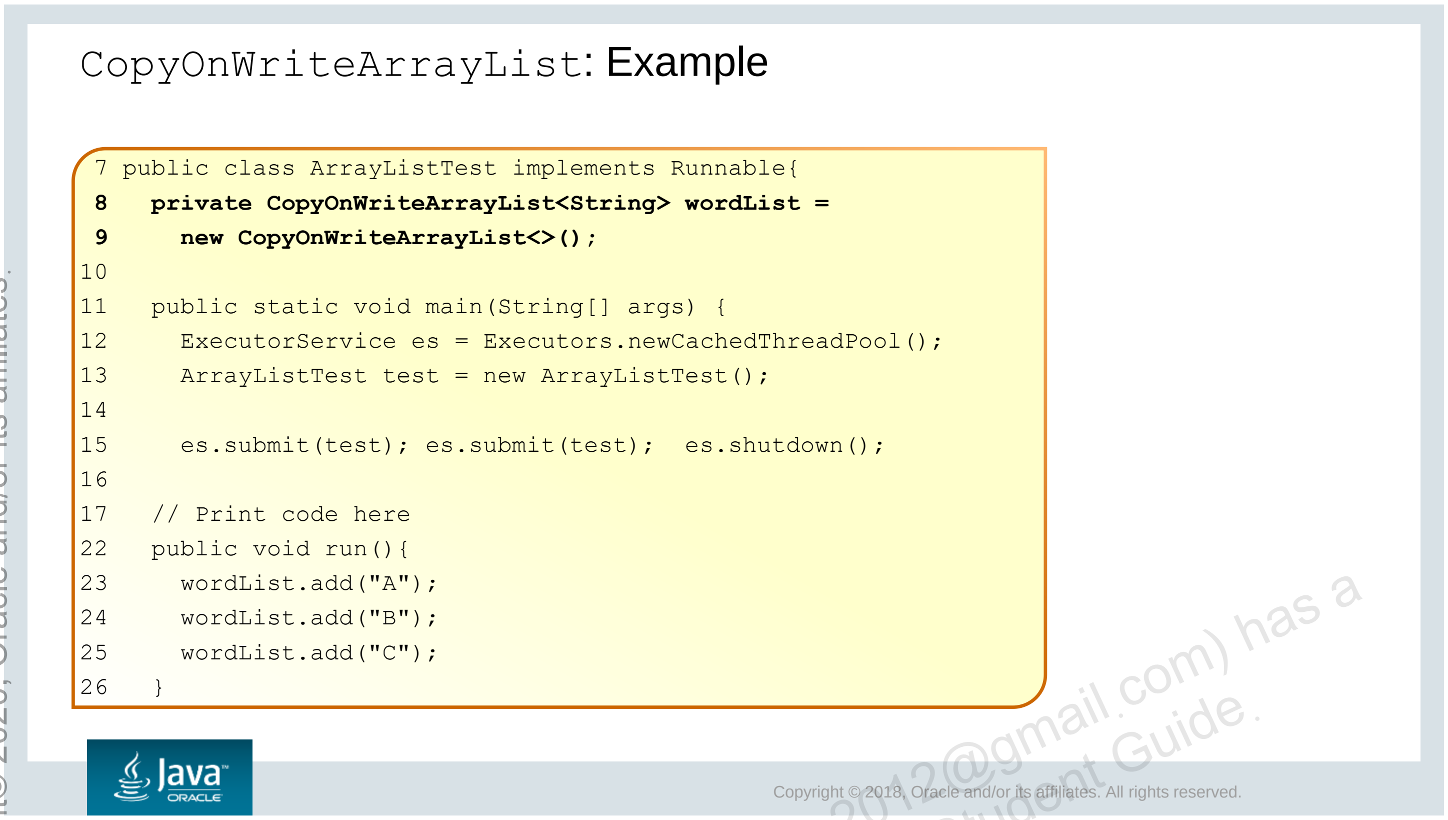
- Use synchronized code blocks for all access to a collection if writes are performed
- Create a synchronized wrapper using library methods, such as `java.util.Collections.synchronizedList(List<T>)`
- Use the `java.util.concurrent` collections

Note: Just because a `Collection` is made thread-safe, this does not make its elements thread-safe.

Concurrent Collections

The `ConcurrentLinkedQueue` class supplies an efficient, scalable, thread-safe, nonblocking FIFO queue. Five implementations in `java.util.concurrent` support the extended `BlockingQueue` interface, which defines blocking versions of put and take: `LinkedBlockingQueue`, `ArrayBlockingQueue`, `SynchronousQueue`, `PriorityBlockingQueue`, and `DelayQueue`.

Besides queues, this package supplies `Collection` implementations designed for use in multithreaded contexts: `ConcurrentHashMap`, `ConcurrentSkipListMap`, `ConcurrentSkipListSet`, `CopyOnWriteArrayList`, and `CopyOnWriteArraySet`. When many threads are expected to access a given collection, a `ConcurrentHashMap` is normally preferable to a synchronized `HashMap`, and a `ConcurrentSkipListMap` is normally preferable to a synchronized `TreeMap`. A `CopyOnWriteArrayList` is preferable to a synchronized `ArrayList` when the expected number of reads and traversals greatly outnumber the number of updates to a list.



CopyOnWriteArrayList: Example

```
public class ArrayListTest implements Runnable{
  private CopyOnWriteArrayList<String> wordList =
    new CopyOnWriteArrayList<>();

  public static void main(String[] args) {
    ExecutorService es = Executors.newCachedThreadPool();
    ArrayListTest test = new ArrayListTest();

    es.submit(test); es.submit(test);  es.shutdown();

  // Print code here
  public void run(){
    wordList.add("A");
    wordList.add("B");
    wordList.add("C");
  }
```

A CopyOnWriteArrayList is a thread safe ArrayList implementation found in the java.util.concurrency library.

Note: The three es statements were combined onto one line so the source code would fit in the slide.

Summary

In this lesson, you should have learned how to:

- Describe operating system task scheduling
- Use an `ExecutorService` to concurrently execute tasks
- Identify potential threading problems
- Use `synchronized` and the classes in the `java.concurrent.atomic` package to manage atomicity
- Use monitor locks to control the order of thread execution
- Use the `java.util.concurrent` collections

Practice 13: Overview

This practice covers the following topics:

- 13-1: Using the `java.util.concurrent` package
- 13-2: Creating a Network Client using the `java.util.concurrent` package

In this practice, you create a multithreaded network client.

Quiz

An `ExecutorService` will always attempt to use all of the available CPUs in a system.

a. True

b. False

Quiz

Variables are thread-safe if they are:

a. `local`

b. `static`

c. `final`

d. `private`