Multithreading is an important ingredient in many production systems. This chapter describes how to create multiple threads in Java, and how to ensure they cooperate with each other when they access common resources.

© Olsen Software, 2016

Contents

1. Creating multiple threads
2. Synchronizing threads
3. Synchronization classes

Demo project: **DemoMultithreading**

2

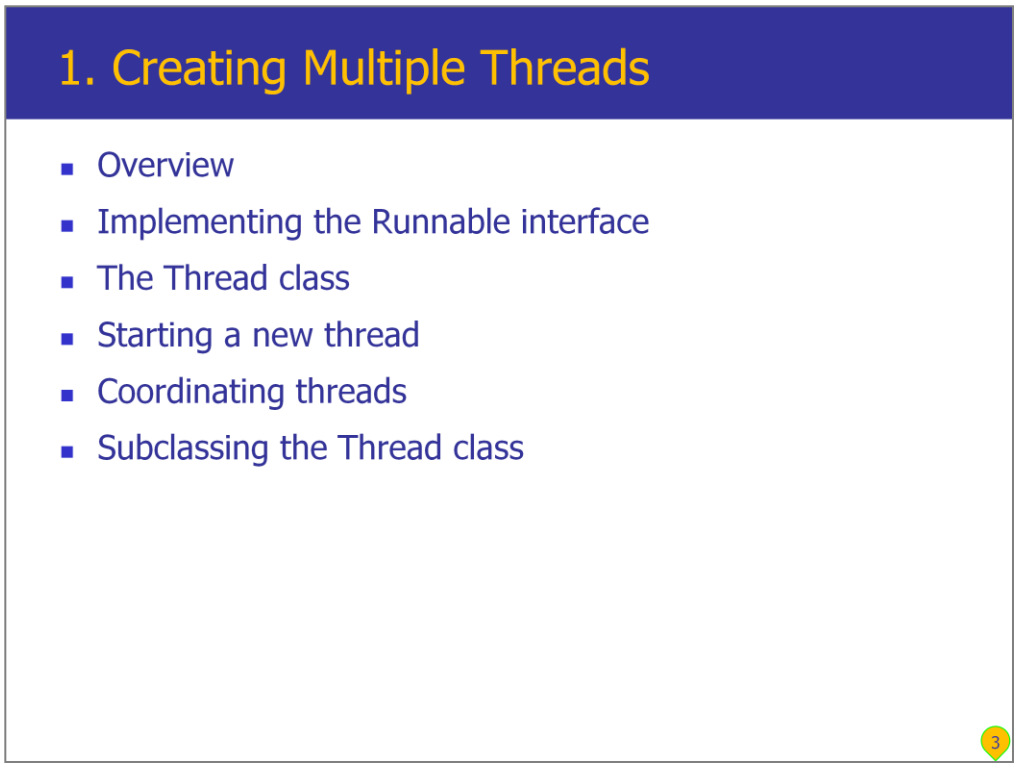
Section 1 describes how to create multiple threads in a Java application. We'll introduce the `Thread` class and the `Runnable` interface.

Section 2 discusses the issue of synchronization, and shows a couple of ways for threads to synchronize access to shared resources.

Section 3 takes the subject of synchronization a step further, and describes some of the special synchronization classes introduced in Java in recent years.

The demos for chapter are located in the `DemoMultithreading` project.

© Olsen Software, 2016



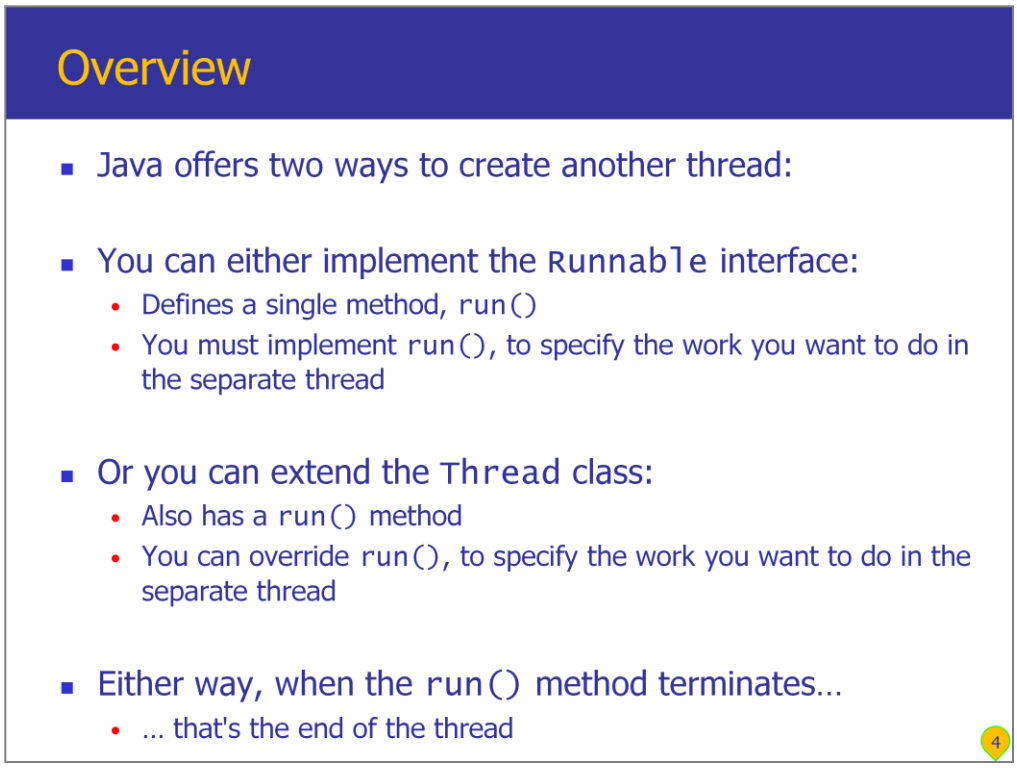
Most applications need multithreading - i.e. the ability to do more than one thing at the time (or to simulate this behaviour in a single-processor machine).

Java has built-in support for multithreading:

- The `Runnable` interface
- The `Thread` class
- Various methods in the `Object` class
- The `synchronized` keyword

We'll start looking at these capabilities in this section.

© Olsen Software, 2016



Java offers two different ways to create another thread:

The first approach is to implement the `Runnable` interface. Put your code-for-the-other-thread in the `run()` method. The code in the `run()` method will be executed in a different thread.

Another approach is to subclass the `Thread` class. The `Thread` class has a `run()` method of its own, so put your code-for-the-other-thread in here. The code in the `run()` method will be executed in a different thread.

Whichever approach you take, when your `run()` method terminates, the system thread terminates too.

On the following slides, we'll concentrate on the `Runnable` interface approach because this is generally favoured. Then we'll see how to achieve the same effect by subclassing `Thread`.

© Olsen Software, 2016

Implementing the Runnable Interface

- This class finds all prime numbers in a specified range
  - This is a time-consuming task, so we do it in a separate thread (i.e. in the `run()` method)

```
public class PrimeNumberFinder implements Runnable {

  private int from, to;
  private List<Integer> primes;

  public PrimeNumberFinder(int from, int to) {
    this.from = from;
    this.to = to;
    this.primes = new ArrayList<Integer>();
  }

  public void run() {
    for (int number = from; number <= to; number++)
      if (isPrime(number))  primes.add(number);
  }

  private boolean isPrime(int number) {
    for (int i = 2; i < number; i++)
      if (number % i == 0)  return false;
    return true;
  }
}
```

PrimeNumberFinder.java

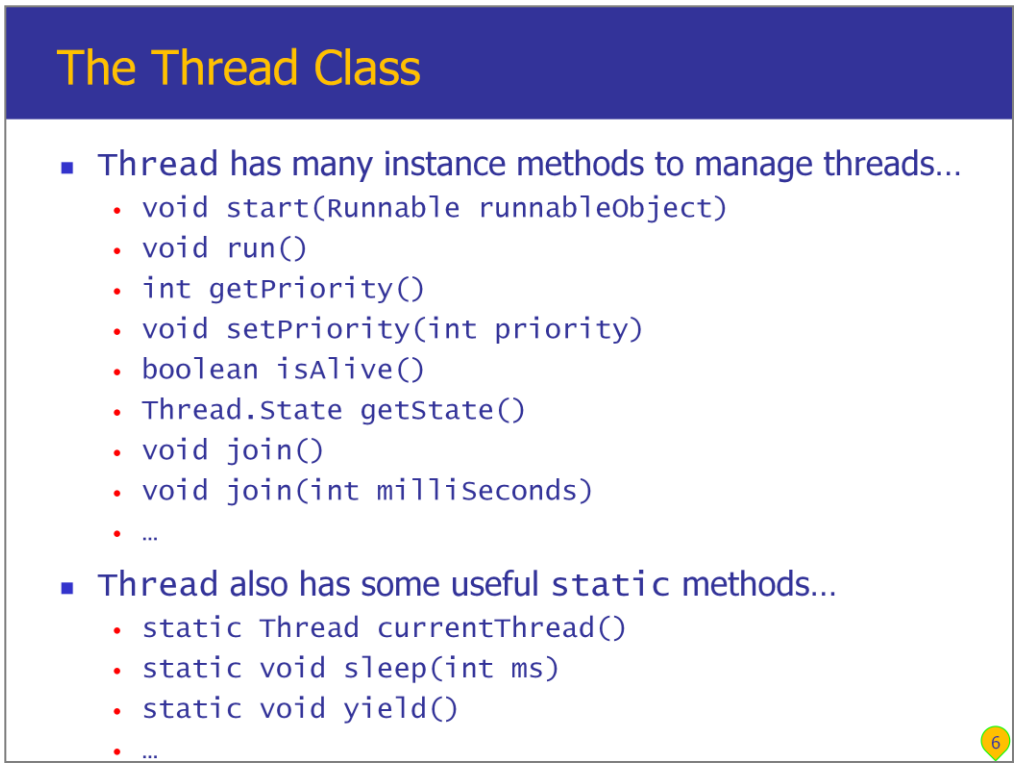
5

This example shows a simple implementation of the `Runnable` interface. Later in this section, we'll see an equivalent example that subclasses `Thread` instead.

The aim of the `PrimeNumberFinder` class is to find all the prime numbers in a specified range. This might take some time, so the class performs this work in a different thread, i.e. in the `run()` method.

Note that the `run()` method cannot receive any parameters, so we initialize the `PrimeNumberFinder` object beforehand with all the information it will need during the execution of the `run()` method.

© Olsen Software, 2016



In order to spin off a separate thread, you must use the `Thread` class.

The `Thread` class contains various methods for creating threads, getting/setting priority, getting the state of the thread, waiting for the thread to finish, pausing the current thread for a specified period, and so on.

For full details about the `Thread` class, see the JavaDoc documentation:

- http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html

© Olsen Software, 2016

Starting a New Thread

- You can create and start a new thread as follows:
  - Create a `Runnable` object
  - Create a `Thread` object
  - Call the `Thread` object's `start()` method, passing the runnable object as a parameter

```
private static void demoRunnableImplementation() {

  System.out.print("Enter 'from': ");
  int from = scanner.nextInt();

  System.out.print("Enter 'to': ");
  int to = scanner.nextInt();

  PrimeNumberFinder finder = new PrimeNumberFinder(from, to);
  Thread backgroundThread = new Thread(finder);
  backgroundThread.start();
  …
}
```

Main.java

7

This example shows how to use `Thread` to create a new thread. Note the following points:

- The first step is to create an instance of your runnable class. This doesn't start the other thread yet, it just creates the object in memory.
- Next, create a `Thread` object and pass your runnable object as a parameter. This doesn't create the other thread either, it just tells the `Thread` object which runnable object to launch when its ready. There's a bit of cleverness involved here - when you pass a `Runnable` object into the `Thread` constructor, the `Thread` object remembers this object as the one that contains the runnable-in-another-thread code.
- Finally, call the `start()` method on the `Thread` object. The `Thread` object will create a new system thread, and invoke the `run()` method on the runnable object you specified in the constructor.

© Olsen Software, 2016

Coordinating Threads

- The original thread can continue to do work while other threads execute
- If you want to wait for the other thread to finish, you can call the `join()` method

```
// Do some work on the main thread, while we're waiting…
// …


// Now let's wait for the other thread to finish (can specify a max wait time here).
try {
  backgroundThread.join();
}
catch (InterruptedException ex) {}

// Get the results from the background thread.
List<Integer> primes = finder.getPrimes();
System.out.println("Prime numbers: " + primes);
```

Main.java

8

If you kick off another thread, that thread will spin up and start doing some work while your main thread continues executing. If your main thread reaches a point in its logic where it really has to wait for the other thread to finish, you can call the `join()` method on the `Thread` object for the other thread. This will suspend your thread until the other thread has finished.

© Olsen Software, 2016

Subclassing the Thread class

- The other way to do multithreading is to subclass the `Thread` class…
- Define a class that subclasses `Thread`
  - Define any constructor parameters as needed, to initialize state
  - Override `run()`, and put your background-thread -code here
  - Example: see `PrimeNumberFinderThread.java`
- Then in your client code:
  - Create an instance of your `Thread` subclass
  - Call the object's `start()` method
  - This will cause the thread's own `run()` method to be called
  - Example: see `Main.java`, `demoThreadSubclass()` method

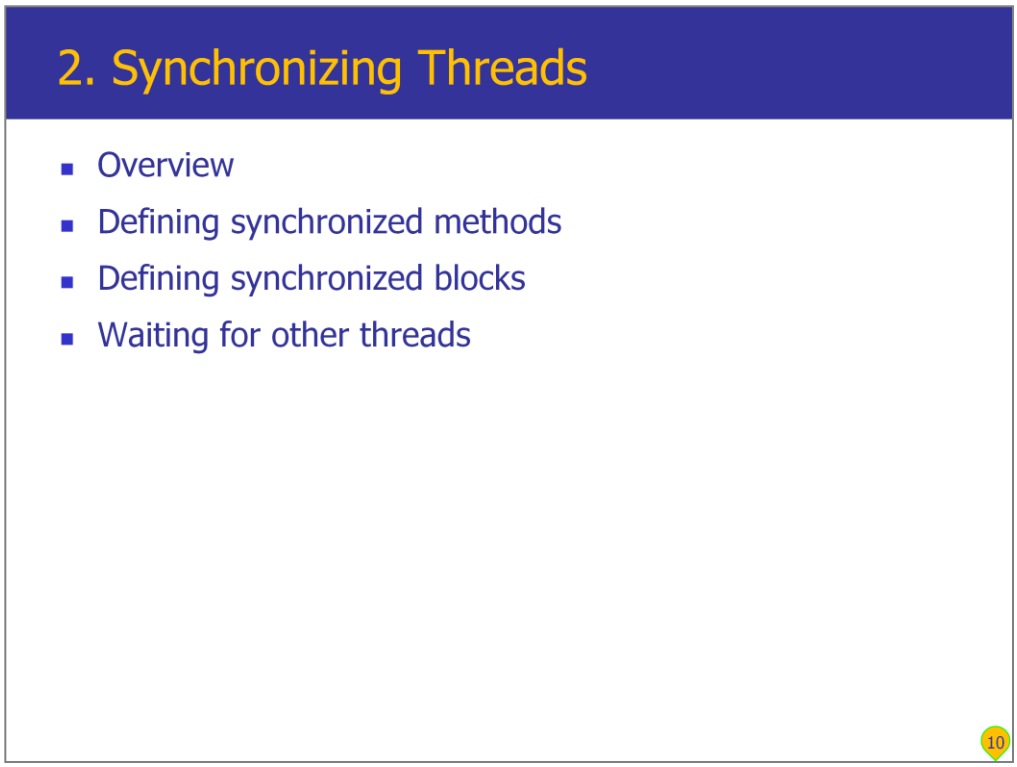
9

This slide explains the other way to achieve multithreading in Java, i.e. by subclassing `Thread` and putting your background-thread code directly in its own `run()` method (i.e. there's no longer a separate class that implements the Runnable interface).

For a complete worked example of this approach, see the following code in the demo project:

- `PrimeNumberFinderThread.java`
- `Main.java`, in the `demoThreadSubclass()` method.

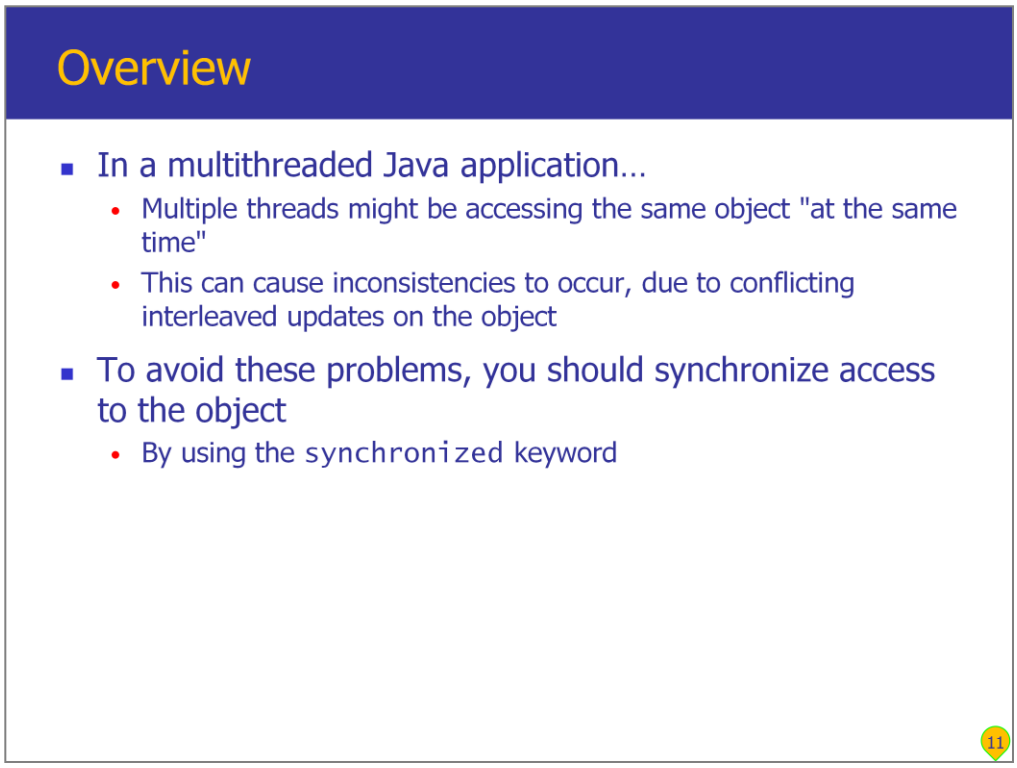
© Olsen Software, 2016



Multithreaded applications have the capability to trample on data used by other threads. This isn't good...

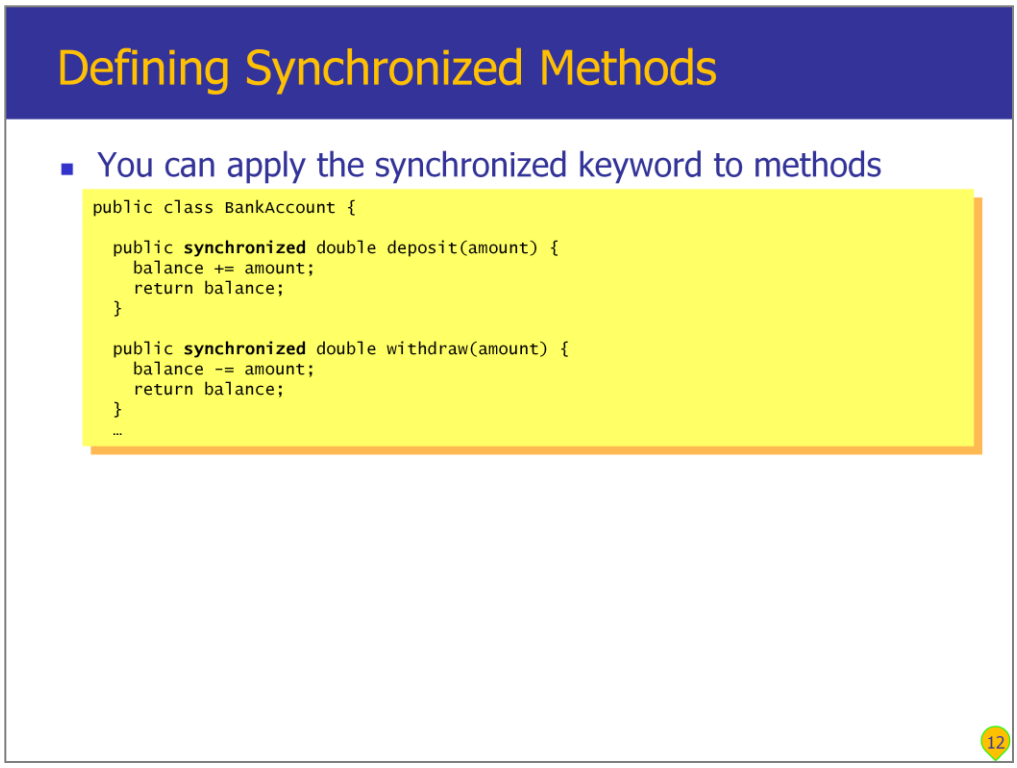
Java provides various synchronization mechanisms that allow you to ensure thread-safe access to common data shared by multiple threads in your application. We'll take a look at some simple language mechanisms in this section, and then we'll look at some more sophisticated synchronization techniques in the next section.

© Olsen Software, 2016



The `synchronized` keyword is the simplest way to ensure your code accesses objects in a thread-safe manner. Internally, the `synchronized` keyword relies on object locks to ensure that an object isn't accessed simultaneously by multiple threads.

© Olsen Software, 2016



When a thread calls a `synchronized` method on an object:

- The JVM tests whether the object's "monitor" is already locked by another thread, and waits for it to be released if necessary.
- This thread locks the object's monitor, and enters the method.

© Olsen Software, 2016

Defining Synchronized Blocks

- You can also apply the synchronized keyword to blocks

```
public class BankAccount {

  List<Transaction> transactions = new ArrayList<Transaction>();

  public void processTransactions()  {
    …
    synchronized (transactions) {
      // We have thread-safe access to transactions list in here…
      // …
    }
    …
  }
```

- What are the benefits of synchronizing on a block, rather than on a method?

13

Rather than applying the `synchronized` keyword to an entire method, you can define a `synchronized` block as shown in the slide. There are two benefits to this approach:

- You can specify any object you want in the `synchronized` statement. For example, different `synchronized` blocks could lock on different objects. This reduces the chances of one thread having to wait on another thread, because they are waiting to access different objects.
- You can minimize the number of statements enclosed inside the `{}` of the `synchronized` block. This reduces the length of time the lock will be held, which thereby reduces the chances that another thread will be left waiting for too long.

© Olsen Software, 2016

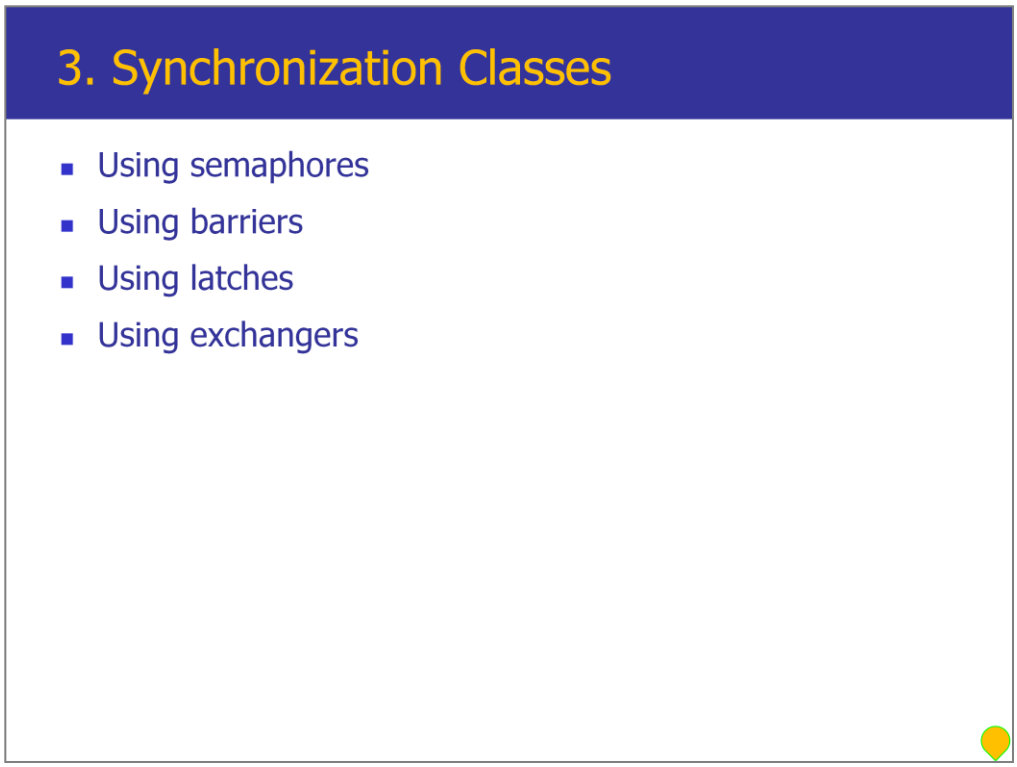
Waiting for Other Threads

- The `Object` class defines 3 methods that allow threads to co-operatively wait for each other…
  - Note: You can only call these methods in a synchronization scope
- `wait()`
  - Tells the calling thread to give up the monitor and go to sleep…
  - Until another thread enters the same monitor and calls `notify()`
- `notify()`
  - Wakes up the first thread that called `wait()` on the same object
- `notifyAll()`
  - Wakes up all the threads that called `wait()` on the same object
  - The highest priority thread will run first

14

In addition to the `join()` method in the `Thread` class that we saw earlier in this chapter, there are also 3 useful thread-related methods in the `Object` class itself. You can call these methods inside a synchronization scope, to control the state of the lock on the current object.

© Olsen Software, 2016



Java 6 introduced various synchronization classes that give you control over how synchronization and locking occurs:

- `Semaphore`
- `CountDownLatch`
- `CyclicBarrier`
- `Exchanger<V>`

We'll explain these classes in the following slides. Code examples for this section are available in the `demo.synchronizers` package.

© Olsen Software, 2016

Using Semaphores

- `Semaphore`:
  - Allows counted number of threads to access a resource concurrently
- To acquire one or more permit(s):
  - Call `acquire()`, blocks until permit(s) available
  - Decrements the number of permits available
- To release one or more permit(s):
  - Call `release()`
  - Increments the number of permits available
- Additional capabilities:
  - `tryAcquire()`, `availablePermits()`, `drainPermits()`, `reducePermits()`

This slide summarizes the capabilities of the `Semaphore` class. For an example of how to use this class, see `DemoSemaphores.java` in the demo project.

For full information about this class, see the JavaDoc documentation:

- http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Semaphore.html

© Olsen Software, 2016

Using Latches

- `CountDownLatch`:
  - Is initialized with a given count
  - Causes threads to wait until the count reaches zero (one-shot)
  - Allows a coordinating thread to subdivide work across several threads, and wait until they have all completed
- To wait on a `CountDownLatch`:
  - Call `await()`, blocks until the latch's count reaches zero
- To signal a `CountDownLatch`:
  - Call `countDown()`
  - When count reaches zero, all threads waiting on latch are released
  - Any subsequent `await()` calls on the latch continue unhindered

This slide summarizes the capabilities of the `CountDownLatch` class. For an example of how to use this class, see `DemoCountDownLatch.java` in the demo project.

For full information about this class, see the JavaDoc documentation:

- http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CountDownLatch.html

© Olsen Software, 2016

Using Barriers

- `CyclicBarrier`:
  - Allows several threads to wait for each other to reach a common barrier point
  - Can be reset after it's fired (hence the term "cyclic barrier")
- A `CyclicBarrier` supports an optional `Runnable` command
  - Run once per barrier point, after the last thread arrives (but before any thread has been released)
  - Useful for updating shared state before any parties continue
- To await all parties' arrival at a barrier:
  - Call `await()`, with an optional timeout

This slide summarizes the capabilities of the `CyclicBarrier` class. For an example of how to use this class, see `DemoCyclicBarrier.java` in the demo project.

For full information about this class, see the JavaDoc documentation:

- http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CyclicBarrier.html

© Olsen Software, 2016

Using Exchangers

- `Exchanger<V>`:
  - Allows threads to swap elements within pairs
  - Effectively, a bidirectional form of `SynchronousQueue`
  - Useful in pipeline-based solutions
- To exchange a value using an `Exchanger<V>`:
  - Call `exchange(v)`, to wait for another thread to arrive at this execution point
  - Causes your specified value to be transferred to other thread…
  - … and you receive the other thread's object in exchange

This slide summarizes the capabilities of the `Exchanger` class. For an example of how to use this class, see `DemoExchanger.java` in the demo project.

For full information about this class, see the JavaDoc documentation:

- http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Exchanger.html

© Olsen Software, 2016

Any Questions?

20

© Olsen Software, 2016