Advanced Object-oriented Programming

Lecture 23

Shared Variables and Synchronization
Communicating Threads

In the previous lecture, we saw an example of a multi-threaded program where three threads shared one instance of a Queue class.

Sharing variables or instances is the most common way for concurrent programs to communicate — certainly for multithreaded programs.

For example, in the Producer-Consumers example, the shared queue allowed the producer to communicate values to the consumers.
Shared Variables

Even though threads might be sharing an *instance* rather than a *variable*, we nevertheless describe this situation as *communication through shared variables*.

(The terminology was introduced before object-orientation was common.)

Even though the three threads in the Producer-Consumers example might store a reference to the queue in *different* variables, all these variables point to the same instance.
Shared Variables

At a slightly more abstract level, we can think of different threads sharing some common resource.

Although this might seem like a very straightforward and innocuous situation, there are pitfalls that can trap the unwary programmer.

To introduce these pitfalls, we look at a program where two threads repeatedly print messages to standard output. (Adapted from Wigglesworth & McMillan, *Java Programming*, Thomson, 2004.)
At a slightly more abstract level, we can think of different threads sharing some common resource.

Although this might seem like a very straightforward and innocuous situation, there are pitfalls that can trap the unwary programmer.

To introduce these pitfalls, we look at a program where two threads repeatedly print messages to standard output. (Adapted from Wigglesworth & McMillan, *Java Programming*, Thomson, 2004.)
The MessagePrinter Class

Class **MessagePrinter** instances will run in a **Thread** and repeatedly print a given message to standard output.

class MessagePrinter implements Runnable {
    private String msg;
    MessagePrinter(String s) {
        msg = s;
    }
}
The MessagePrinter Class

Class `MessagePrinter` instances will run in a `Thread` and repeatedly print a given message to standard output.

class MessagePrinter implements Runnable {
    private String msg;
    MessagePrinter(String s) {
        msg = s;
    }
}
The MessagePrinter Class

Class `MessagePrinter` instances will run in a `Thread` and repeatedly print a given message to standard output.

class MessagePrinter implements Runnable {
    private String msg;
    MessagePrinter(String s) {
        msg = s;
    }
}
The run() Method

class MessagePrinter, contd.

```java
public void run() {
    for (int i = 0; i < 2000; i++) {
        for (int j = 0; j < msg.length(); j++) {
            System.out.print(msg.charAt(j));
        }
        System.out.print("\n");
    }
}
```

Print the string character-by-character
Repeat 2000 times
The run() Method

class MessagePrinter, contd.

```java
public void run() {
    for (int i = 0; i < 2000; i++) {
        for (int j = 0; j < msg.length(); j++) {
            System.out.print(msg.charAt(j));
        }
        System.out.print("\n");
    }
}
```

Print the string character-by-character
Repeat 2000 times
The run() Method

class MessagePrinter, contd.

public void run() {
    for (int i = 0; i < 2000; i++) {
        for (int j = 0; j < msg.length(); j++) {
            System.out.print(msg.charAt(j));
        }
        System.out.print("\n");
    }
}

Print the string character-by-character
Repeat 2000 times
The `main()` Method

```java
public static void main(String[] args) {
    MessagePrinter mp1 = 
        new MessagePrinter("Hello, world");
    MessagePrinter mp2 = 
        new MessagePrinter("Goodbye, world");

    Thread t1 = new Thread(mp1);
    Thread t2 = new Thread(mp2);

    t1.start();
    t2.start();
}
```
Hello, world
Hello, world
Hello, woGlodo
dHbeylel,o ,w owrolrdl
dG
oHoedlblyoe,, wwoorrllddd
HGeololdob,y ew,o rwlodr
lHde
lGlooo,d bwyoer,l dw
oHrelldl
oG,o owdobryled,

This output was actually obtained using an old version of Java, where threads’ quanta were much shorter than in the current version.
Controlling Shared Resources

The problem here is that both threads want access to the shared resource of `System.out`.

When each thread `gets` access to `System.out`, it prints out a string character by character. The time-slicing implemented by the Java virtual machine means that the running thread will probably be put back in the ready-pool before it’s printed out its entire message.

The result is scrambled messages appearing on standard output.
This is an example of interference:

**definition of interference**

data-corruption that occurs when two or more threads share some resource.

Interference is a common problem in concurrent programming, and various techniques exist to avoid this problem.
A critical section is a part of a program that should not be interrupted by time-slicing.

That is, when a thread enters (starts executing) a critical section, no other thread should be allowed to enter that critical section before the first thread has finished executing the critical section.

In our example, we want the code that prints an entire message to be a critical section.
A critical section is a part of a program that should not be interrupted by time-slicing.

That is, when a thread enters (starts executing) a critical section, no other thread should be allowed to enter that critical section before the first thread has finished executing the critical section.

In our example, we want the code that prints an entire message to be a critical section.
A **critical section** is a part of a program that should not be interrupted by time-slicing.

That is, when a thread enters (starts executing) a critical section, no other thread should be allowed to enter that critical section before the first thread has finished executing the critical section.

In our example, we want the code that prints an entire message to be a critical section.
Monitors

In the 1950s, Tony Hoare introduced the notion of a monitor as a solution to the problem of critical sections.

A monitor can be thought of as a lock that has only one key, and that shuts off the critical section.

Before a thread enters a critical section, it must first get the key to the lock. Once it gets the key, it enters the critical section, and keeps the key until it has finished. On leaving the critical section, it gives back the key so that other threads can enter the critical section.
In the 1950s, Tony Hoare introduced the notion of a *monitor* as a solution to the problem of critical sections.

A monitor can be thought of as a *lock* that has only one *key*, and that shuts off the critical section.

Before a thread enters a critical section, it must first get the key to the lock.

Once it gets the key, it enters the critical section, and keeps the key until it has finished.

On leaving the critical section, it gives back the key so that other threads can enter the critical section.
In the 1950s, Tony Hoare introduced the notion of a monitor as a solution to the problem of critical sections.

A monitor can be thought of as a lock that has only one key, and that shuts off the critical section.

Before a thread enters a critical section, it must first get the key to the lock. Once it gets the key, it enters the critical section, and keeps the key until it has finished.

On leaving the critical section, it gives back the key so that other threads can enter the critical section.
In the 1950s, Tony Hoare introduced the notion of a *monitor* as a solution to the problem of critical sections.

A monitor can be thought of as a *lock* that has only one *key*, and that shuts off the critical section.

Before a thread enters a critical section, it must first get the key to the lock. Once it gets the key, it enters the critical section, and keeps the key until it has finished. On leaving the critical section, it gives back the key so that other threads can enter the critical section.
Monitors in Java

Java allows critical sections to be protected by monitors, by using the keyword `synchronized`:

```java
synchronized(Object) {
    // critical section goes here
}
```

where `Object` is a reference to an instance of any class.
Monitors in the Interpreter

The effect of this is:

- the interpreter creates a monitor for the code in the **synchronized** block;
- it assigns the key to the *Object*;
- before a running thread can enter the critical section, it must obtain the key from that object;
- if the key is not available, the thread is put back in the ready-pool;
- if that thread is later chosen to run, it must obtain the key before entering the critical section (otherwise back to the pool);
Monitors in the Interpreter

The effect of this is:

- the interpreter creates a monitor for the code in the `synchronized` block;
- it assigns the key to the `Object`;
- before a running thread can enter the critical section, it must obtain the key from that object;
- if the key is not available, the thread is put back in the ready-pool;
- if that thread is later chosen to run, it must obtain the key before entering the critical section (otherwise back to the pool);
The effect of this is:

- the interpreter creates a monitor for the code in the `synchronized` block;
- it assigns the key to the `Object`;
- before a running thread can enter the critical section, it must obtain the key from that object;
- if the key is not available, the thread is put back in the ready-pool;
- if that thread is later chosen to run, it must obtain the key before entering the critical section (otherwise back to the pool);
Monitors in the Interpreter

The effect of this is:

- the interpreter creates a monitor for the code in the `synchronized` block;
- it assigns the key to the `Object`;
- before a running thread can enter the critical section, it must obtain the key from that object;
- if the key is not available, the thread is put back in the ready-pool;
- if that thread is later chosen to run, it must obtain the key before entering the critical section (otherwise back to the pool);
Monitors in the Interpreter

The effect of this is:

- the interpreter creates a monitor for the code in the `synchronized` block;
- it assigns the key to the `Object`;
- before a running thread can enter the critical section, it must obtain the key from that object;
- if the key is not available, the thread is put back in the ready-pool;
- if that thread is later chosen to run, it must obtain the key before entering the critical section (otherwise back to the pool);
Monitors in the Interpreter

- once a running thread obtains the key, it keeps it until it has finished executing the synchronized block, at which point the key becomes available to other threads;
- if a thread has a key, it keeps it — even in the ready-pool and the blocked-pool.

Note that this entire process is managed by the interpreter: all the programmer does is specify the synchronized block and the object that keeps the key.
Monitors in the Interpreter

- once a running thread obtains the key, it keeps it until it has finished executing the **synchronized** block, at which point the key becomes available to other threads;
- if a thread has a key, it keeps it — even in the ready-pool and the blocked-pool.

Note that this entire process is managed by the interpreter: all the programmer does is specify the **synchronized** block and the object that keeps the key.
Monitors in the Interpreter

- once a running thread obtains the key, it keeps it until it has finished executing the `synchronized` block, at which point the key becomes available to other threads;
- if a thread has a key, it keeps it — even in the ready-pool and the blocked-pool.

Note that this entire process is managed by the interpreter: all the programmer does is specify the `synchronized` block and the object that keeps the key.
Monitors and MessagePrinters

In our example, we’ll synchronize the block of code that prints a message to standard output. This means that only one thread at a time can execute that code.

The choice of an object to keep the key is pretty much arbitrary; we’ll create an instance of the Integer class for this purpose.
class MessagePrinter v2

class MessagePrinter implements Runnable {
    private String msg;
    private static Integer lock = new Integer(0);
    MessagePrinter(String s) {
        msg = s;
    }
}

This will be the ‘key-keeper’
This is a ‘class’ field, so all instances of MessagePrinter share
the same key-keeper
class MessagePrinter v2

class MessagePrinter implements Runnable {
    private String msg;
    private static Integer lock = new Integer(0);
    MessagePrinter(String s) {
        msg = s;
    }
}

This will be the ‘key-keeper’

This is a ‘class’ field, so all instances of MessagePrinter share the same key-keeper
class MessagePrinter implements Runnable {
    private String msg;
    private static Integer lock = new Integer(0);
    MessagePrinter(String s) {
        msg = s;
    }
}

This will be the ‘key-keeper’

This is a ‘class’ field, so all instances of MessagePrinter share the same key-keeper
In the run() Method

```java
synchronized(lock) {
    for (int j=0; j<msg.length(); j++) {
        System.out.print(msg.charAt(j));
        try {
            Thread.sleep(10);
        } catch (InterruptedException e) {
        }
    }
    System.out.print("\n");
}
```

Critical section: any Thread has to get the key from lock before running this
In the run() Method

synchronized (lock) {
    for (int j = 0; j < msg.length(); j++) {
        System.out.print(msg.charAt(j));
        try {
            Thread.sleep(10);
        } catch (InterruptedException e) {
        }
    }
    System.out.print("\n");
}

Critical section: any Thread has to get the key from lock before running this
Notes

Note we’ve added a call to `sleep()` in this method. The effect of this is to make a ‘fair’ interleaving of the two threads more likely.

The main reason we’ve added this call to `sleep()`, however, is to show that the interpreter’s implementation of monitors still works even when a thread is certain to be ‘taken out of the running’ in the middle of the critical section.
The Output

terminal output

Hello, world
Hello, world
Goodbye, world
Hello, world
Goodbye, world
Hello, world
Goodbye, world
Non-Static Key-keepers

Recall that we made the field `lock` static:

```java
private static Integer lock = new Integer(0);
```

Earlier versions of Java used to require that Key-keepers could not contain any non-static references.

The compiler would give an error message if we left out the `static` modifier from the declaration of `lock`.

In Java 1.6, if we remove the `static` modifier,

```java
private Integer lock = new Integer(0);
```

the code will compile, but it is a semantic error.
Non-Static Key-keepers

Recall that we made the field `lock` static:

```java
private static Integer lock = new Integer(0);
```

Earlier versions of Java used to require that Key-keepers could not contain any non-static references.

The compiler would give an error message if we left out the static modifier from the declaration of `lock`.

In Java 1.6, if we remove the static modifier,

```java
private Integer lock = new Integer(0);
```

the code will compile, but it is a semantic error.
Key-Keepers should be Unique

If we have a non-static field `lock` keeping the key to the critical section, then each instance of `MessagePrinter` would have its own key-keeper for the critical section.

The result would be that each thread would have its own instance of `MessagePrinter`, and each instance of `MessagePrinter` would have its own key-keeper for the critical section.

Obtaining the key for the critical section would be trivial: each thread would go through its own key-keeper. The result would be — again — scrambled messages.
Key-Keepers should be Unique

Similarly, consider:

```java
synchronized(new Integer(0)) {
    for (int j=0; j<msg.length(); j++) {
        System.out.print(msg.charAt(j));
        try {
            Thread.sleep(10);
        } catch (InterruptedException e) {
        }
    }
    System.out.print("\n");
}
```
Key-Keepers should be Unique

In this case, each time the code is executed, a new key-keeper will be created. Again, the result is scrambled messages. Generally, the best approach is to use a static field for the key-keeper.
Threads and GUIs

One of the main reasons for using multithreaded programming is to allow a user to continue interacting with an application while some (lengthy) computation is in progress.

For applications with a GUI, it may be important — and it will probably be desirable — to let users interact with the GUI while event-handling code (from a previous interaction) is being executed.
Threads and GUIs

One of the main reasons for using multithreaded programming is to allow a user to continue interacting with an application while some (lengthy) computation is in progress.

For applications with a GUI, it may be important — and it will probably be desirable — to let users interact with the GUI while event-handling code (from a previous interaction) is being executed.
Threads and GUls

We’ll look at the following questions:

- How is event-handling code executed?
- Does the programmer need to create threads?
- Can GUI components suffer from interference?
How is Event-Handling Code Executed?

When the Java interpreter starts up, there is one thread, ‘main’, that executes the `main()` method.
For GUIs, the `main()` method typically creates and displays a Frame; all further computation is then driven by component-generated event (button-clicks, etc.).
The ‘main’ thread terminates (dies), and this would normally mean that the Java interpreter would exit.
However...
How is Event-Handling Code Executed?

When an AWT or swing Frame is created, a special thread is created by the Java interpreter to handle component-generated events and their event-handling code. This thread is called the Event-Dispatch Thread.

All event-handling code is executed in this thread (i.e., all the actionPerformed(), etc., methods).

The Event-dispatch Thread terminates when all Frames have been closed.
Does the Programmer *Need* to Create Threads?

No.

By default, the Event-Dispatch Thread will be used to execute all event-handling code.

However, there may be occasions when it’s *desirable* to create a separate thread for lengthy event-handling code (e.g., reading a large file across a network).
Does the Programmer *Need* to Create Threads?

No.

By default, the Event-Dispatch Thread will be used to execute all event-handling code.

However, there may be occasions when it’s *desirable* to create a separate thread for lengthy event-handling code (e.g., reading a large file across a network).
Does the Programmer *Need* to Create Threads?

No.

By default, the Event-Dispatch Thread will be used to execute all event-handling code.

However, there may be occasions when it’s *desirable* to create a separate thread for lengthy event-handling code (e.g., reading a large file across a network).
Can GUI Components Suffer from Interference?

If only the Event-Dispatch thread is used: no. (Interference is when data corruption arises from *more* than one thread accessing a shared resource.)

If the programmer has explicitly created new threads in the event-handling code, and these threads access a shared resource (e.g., a TextArea): yes, interference is a possibility.
A component is **thread-safe** if its methods are synchronized to prevent interference.

**Most** components in the AWT package are thread-safe; **most** components in the swing package are *not* thread-safe.

So if you create threads in event-handling code, take care!
How Many Threads Do You Need?

Java uses one thread to handle all user interaction with a GUI. This makes sense, because most responses to user interactions can be done quickly (save a file, modify/bring up a new GUI element, etc.); but creating and maintaining threads uses a lot of time and memory.

There are many situations where a program needs to handle a lot of requests for tasks that can be handled quickly. Typical examples are GUIs and servers. For example, web servers may receive many requests in a short period; but each request takes only a short time to handle (get the file; send it). In such cases, it makes sense to use a small number of threads to do all the work.
How Many Threads Do You Need?

Java uses one thread to handle all user interaction with a GUI. This makes sense, because most responses to user interactions can be done quickly (save a file, modify/bring up a new GUI element, etc.); but creating and maintaining threads uses a lot of time and memory.

There are many situations where a program needs to handle a lot of requests for tasks that can be handled quickly. Typical examples are GUIs and servers. For example, web servers may receive many requests in a short period; but each request takes only a short time to handle (get the file; send it). In such cases, it makes sense to use a small number of threads to do all the work.
How Many Threads Do You Need?

Java uses one thread to handle all user interaction with a GUI. This makes sense, because most responses to user interactions can be done quickly (save a file, modify/bring up a new GUI element, etc.); but creating and maintaining threads uses a lot of time and memory.

There are many situations where a program needs to handle a lot of requests for tasks that can be handled quickly. Typical examples are GUIs and servers. For example, web servers may receive many requests in a short period; but each request takes only a short time to handle (get the file; send it).

In such cases, it makes sense to use a small number of threads to do all the work.
How Many Threads Do You Need?

Java uses one thread to handle all user interaction with a GUI. This makes sense, because most responses to user interactions can be done quickly (save a file, modify/bring up a new GUI element, etc.); but creating and maintaining threads uses a lot of time and memory.

There are many situations where a program needs to handle a lot of requests for tasks that can be handled quickly. Typical examples are GUIs and servers. For example, web servers may receive many requests in a short period; but each request takes only a short time to handle (get the file; send it).

In such cases, it makes sense to use a small number of threads to do all the work.
One Thread Does It All

Writing a thread to perform many tasks might go like this:

```java
class Worker implements Runnable {
    public void run() {
        while (stillNeeded) {
            get next task
            do it
        }
    }
}
```

Run this in a thread, and that thread will execute many tasks.
What To Do

In this context a task is just some Java code e.g., in the case of GUIs, the event-handling code for saving a file, bringing up a new GUI element, etc., or, in the case of a server, getting and transmitting a file, etc.

How can we treat such snippets of code — tasks — in a generic way?

Interfaces!
Let’s say a task is an instance of some class that implements Runnable (i.e., a task is whatever code is contained in such an instance’s run() method).
In this context a task is just some Java code
e.g., in the case of GUIs, the event-handling code for saving a
file, bringing up a new GUI element, etc.,
or, in the case of a server, getting and transmitting a file, etc.

How can we treat such snippets of code — tasks — in a
generic way?

Interfaces!
Let’s say a task is an instance of some class that implements
Runnable
(i.e., a task is whatever code is contained in such an instance’s
run() method).
How To Do It

This lets us give a bit more detail:

class Worker implements Runnable {
    public void run() {
        Runnable r;
        while (stillNeeded) {
            // get next task
            r = next task
            // do it
            r.run();
        }
    }
}
What To Do

If we have a lot of tasks, how can we make them available to our hard-working thread to execute?

We can put them onto a Queue, then create some number of Worker threads that repeatedly get tasks from the Queue and execute them.

```java
in some main method, somewhere
Queue<Runnable> jobs = new Queue<Runnable>();
for (int i = 0; i < 200; i++) {
    jobs.add(new MessagePrinter("task" + i));
}
```
If we have a lot of tasks, how can we make them available to our hard-working thread to execute?

We can put them onto a Queue, then create some number of Worker threads that repeatedly get tasks from the Queue and execute them.

```java
Queue<Runnable> jobs = new Queue<Runnable>();
for (int i = 0; i < 200; i++) {
    jobs.add(new MessagePrinter("task" + i));
}
```
Back to our hard-working thread:
it should read its next task to execute from the queue:

```java
class Worker implements Runnable {

    private Queue<Runnable> tasks;

    public Worker(Queue<Runnable> q) {
        tasks = q;
    }
}
```
Back to our hard-working thread: it should read its next task to execute from the queue:

```
class Worker implements Runnable {
    private Queue<Runnable> tasks;
    public Worker(Queue<Runnable> q) {
        tasks = q;
    }
```
Our hard-working threads are Consumers. They repeatedly read tasks from a queue and execute them. (This gives a hint how to implement the stillNeeded part.)
public void run() {
    Runnable r;
    while (stillNeeded) {
        r = tasks.getNext();
        r.run();
    }
}

Our hard-working threads are Consumers. They repeatedly read tasks from a queue and execute them. (This gives a hint how to implement the stillNeeded part.)
What To Do

What about the Producer?

```java
Queue<Runnable> jobs = new Queue<Runnable>();
for (int i = 0; i < 200; i++) {
    jobs.add(new MessagePrinter("task" + i));
}
for (int i = 0; i < 5; i++) {
    new Worker(jobs).start();
}
```

Create 5 threads to consume/execute the tasks and start them running
What To Do

What about the Producer?

in some main method, somewhere

Queue<Runnable> jobs = new Queue<Runnable>();
for (int i = 0; i < 200; i++) {
    jobs.add(new MessagePrinter("task" + i));
}

for (int i = 0; i < 5; i++) {
    new Worker(jobs).start();
}

Create 5 threads to consume/execute the tasks
and start them running
What To Do

What about the Producer?

```java
Queue<Runnable> jobs = new Queue<Runnable>();
for (int i = 0; i < 200; i++) {
    jobs.add(new MessagePrinter("task" + i));
}
for (int i = 0; i < 5; i++) {
    new Worker(jobs).start();
}
```

Create 5 threads to consume/execute the tasks and start them running
 Wouldn’t it be nice if someone wrote a class of worker threads that executed any number of tasks concurrently? (and took care of the stillNeeded part)

Then the programmer need only write the tasks to be executed, and not worry about creating too many threads (which can be expensive in resource use).

Someone has. The class is java.util.concurrent.Executor
Wouldn’t it be Nice

Wouldn’t it be nice if someone wrote a class of worker threads that executed any number of tasks concurrently? (and took care of the stillNeeded part)

Then the programmer need only write the tasks to be executed, and not worry about creating too many threads (which can be expensive in resource use).

Someone has. The class is java.util.concurrent.Executor
Class `java.util.concurrent` has static methods `newFixedThreadPool` and `newCachedThreadPool` that return an `ExecutorService` instance.

An `ExecutorService` is an object that has a void method `execute(Runnable)`. 

`newFixedThreadPool()` creates a ‘pool’ consisting of a fixed number of ‘worker’ threads that will execute the given tasks. 

`newCachedThreadPool()` also creates a ‘pool’ of threads that will execute the given tasks. When a task is to be executed, an existing thread in the pool will be used, if one is available; if no thread is available, a new one will be created.
Class `java.util.concurrent` has static methods `newFixedThreadPool` and `newCachedThreadPool` that return an `ExecutorService` instance.

An `ExecutorService` is an object that has a void method `execute(Runnable)`.  

`newFixedThreadPool()` creates a ‘pool’ consisting of a fixed number of ‘worker’ threads that will execute the given tasks.

`newCachedThreadPool()` also creates a ‘pool’ of threads that will execute the given tasks. When a task is to be executed, an existing thread in the pool will be used, if one is available; if no thread is available, a new one will be created.
Using Executors

the previous example, redone

```java
ExecutorService workers = Executors.newFixedThreadPool(5);
for (int i = 0; i < 200; i++) {
    workers.execute(
        new MessagePrinter("task" + i));
}
workers.shutdown();
```

Create a pool of five worker threads
have the worker threads execute 200 tasks
Make the threads stop...
Using Executors

Create a pool of five worker threads
have the worker threads execute 200 tasks
Make the threads stop...
Using Executors

the previous example, redone

```java
ExecutorService workers = Executors.newFixedThreadPool(5);
for (int i = 0; i < 200; i++) {
    workers.execute(new MessagePrinter("task" + i));
}
workers.shutdown();
```

Create a pool of five worker threads
have the worker threads execute 200 tasks
Make the threads stop...
shutdown

Method `shutdown()` in interface `ExecutorService` causes the worker threads to terminate once all tasks in the queue have been read and executed.

Method `shutdownNow()` in interface `ExecutorService` causes the worker threads stop reading new tasks from the queue, and to terminate once all tasks that are currently being executed have terminated.

Using `shutdownNow()` might mean that not all tasks get executed.
Method `shutdown()` in interface `ExecutorService` causes the worker threads to terminate once all tasks in the queue have been read and executed.

Method `shutdownNow()` in interface `ExecutorService` causes the worker threads stop reading new tasks from the queue, and to terminate once all tasks that are currently being executed have terminated.

Using `shutdownNow()` might mean that not all tasks get executed.
Method `shutdown()` in interface `ExecutorService` causes the worker threads to terminate once all tasks in the queue have been read and executed.

Method `shutdownNow()` in interface `ExecutorService` causes the worker threads to stop reading new tasks from the queue, and to terminate once all tasks that are currently being executed have terminated.

Using `shutdownNow()` might mean that not all tasks get executed.
That’s All, Folks!

Summary

- Interference
- Critical Sections
- Monitors
- Synchronization

Next:

Synchronization and deadlock