JAVA PROGRAMMING

Week 9: Multithreaded Programming

Lecturer:

- HO Tuan Thanh, M.Sc.

Plan

1. Multithreading fundamentals
2. Synchronization

Plan

1. **Multithreading fundamentals**
2. Synchronization

Concurrency

- Computer users want to continue working in a word processor, while other applications download files, manage the print queue, and stream audio.
- Even a single application is often expected to do more than one thing at a time.

Concurrency

- The streaming audio application must simultaneously read the digital audio off the network, decompress it, manage playback, and update its display.
- Even the word processor should always be ready to respond to keyboard and mouse events, no matter how busy it is reformatting text or updating the display.

Concurrency

- Software that can do such things is known as concurrent software.

- java.util.concurrent packages.

Multithreading fundamentals

- Two distinct types of multitasking:
  - process-based and
  - thread-based.
- **Process-based multitasking** is the feature that allows your computer to run two or more programs concurrently.
  - A program is the smallest unit of code that can be dispatched by the scheduler.
- In a **thread-based multitasking** environment, the thread is the smallest unit of dispatchable code.
  - A single program can perform two or more tasks at once.
- Process-based multitasking is not under the control of Java. Multithreaded multitasking is.

Processes and Threads

- In concurrent programming, there are two basic units of execution: processes and threads.
- In the Java programming language, concurrent programming is mostly concerned with threads.

Processes and Threads

- A computer system normally has many active processes and threads.
- This is true even in systems that only have a single execution core, and thus only have one thread actually executing at any given moment.
- Processing time for a single core is shared among processes and threads through an OS feature called time slicing.

Processes and Threads

- It's becoming more and more common for computer systems to have multiple processors or processors with multiple execution cores.
- This greatly enhances a system's capacity for concurrent execution of processes and threads.
- But concurrency is possible even on simple systems, without multiple processors or execution cores.

Processes

- A process has a self-contained execution environment.
- A process generally has a complete, private set of basic run-time resources.
- In particular, each process has its own memory space.

Threads

- Threads are sometimes called lightweight processes.
- Both processes and threads provide an execution environment, but creating a new thread requires fewer resources than creating a new process.

Threads

- Threads exist within a process, every process has at least one.
- Threads share the process's resources, including memory and open files.
- This makes for efficient, but potentially problematic, communication.

Theads

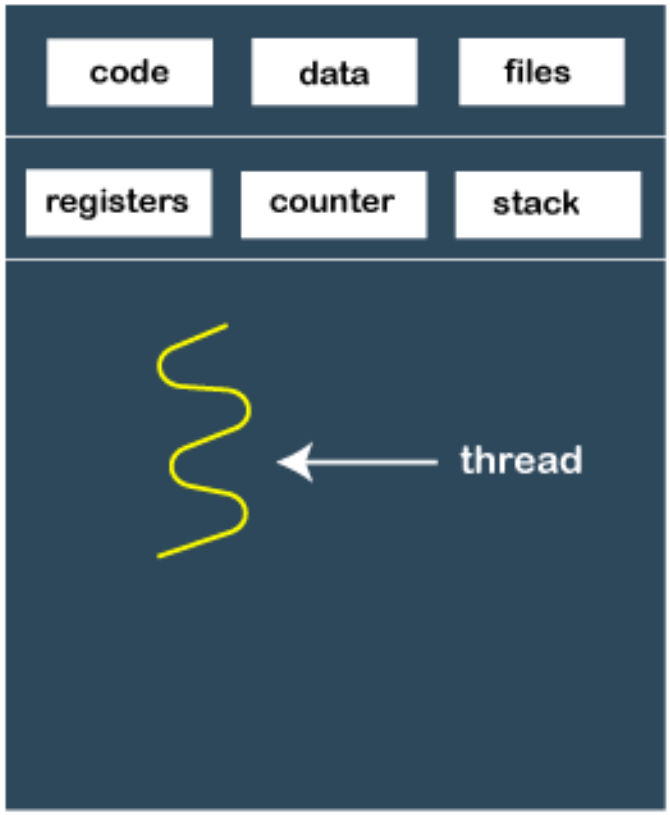
- Multithreaded execution is an essential feature of the Java platform.
- Every application has at least one thread, or several, if you count "system" threads that do things like memory management and signal handling.

Theads

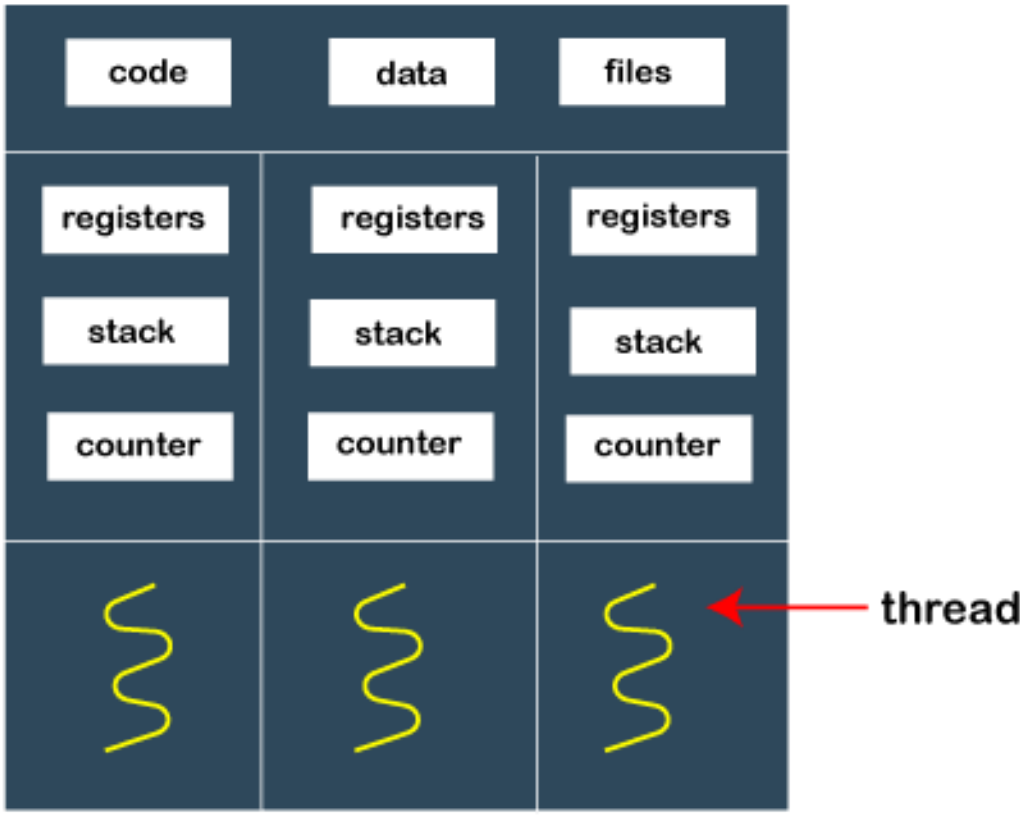
- But from the application programmer's point of view, you start with just one thread, called the main thread.
- This thread has the ability to create additional threads.

Advantage of multithreading

- It enables you to write very efficient programs because it lets you utilize the idle time that is present in most programs.
  - A program will often spend a majority of its execution time waiting to send or receive information to or from a device.
  - By using multithreading, your program can execute another task during this idle time.
- Java's multithreading features work in both single-core and multi-core systems.
  - In a single-core system: concurrently executing threads share the CPU, with each thread receiving a slice of CPU time → two or more threads do not actually run at the same time, but idle CPU time is utilized.
  - In multiprocessor/multicore systems: Two or more threads can be executed simultaneously.



Single-threaded process



Multi-threaded process

States of a thread

- A thread can be in one of several states:
  - running,
  - suspended (a temporary halt to its execution)
  - resumed,
  - blocked (when waiting for a resource),
  - terminated.
- Synchronization allows the execution of threads to be coordinated in certain well-defined ways.

Thread class and Runnable interface

- Thread class and its companion interface, Runnable are packaged in java.lang.
- Thread encapsulates a thread of execution.
- To create a new thread: your program will either extend Thread or implement the Runnable interface.

Method	Meaning
final String getName( )	Obtains a thread's name.
final int getPriority( )	Obtains a thread's priority.
final boolean isAlive( )	Determines whether a thread is still running.
final void join( )	Waits for a thread to terminate.
void run( )	Entry point for the thread.
static void sleep(long *milliseconds*)	Suspends a thread for a specified period of milliseconds.
void start( )	Starts a thread by calling its **run**( ) method.

Creating a thread [1]

- A thread is created by instantiating an object of type Thread.
  - Thread class encapsulates an object that is runnable.
- Two ways to create a runnable object:
  - Implement the Runnable interface.
  - Extend the Thread class.
- **Remember**: Both approaches still use the Thread class to instantiate, access, and control the thread. The only difference is how a thread-enabled class is created.

Creating a thread [2]

- Runnable interface abstracts a unit of executable code.
- You can construct a thread on any object that implements the Runnable interface.
- Runnable defines only one method called run():

```
public void run()
```

  - Inside run(), you will define the code that constitutes the new thread.
  - run() can also call other methods, use other classes, and declare variables just like the main thread.
  - The only difference: run() establishes the entry point for another, concurrent thread of execution within your program.
  - This thread will end when run() returns.

Creating a thread [3]

- After having created a class that implements Runnable, you will instantiate an object of type Thread on an object of that class.
- Constructor:

  Thread(Runnable threadOb)

  - threadOb : an instance of a class that implements the Runnable interface. This defines where execution of the thread will begin.
- The start() method: void start() makes the new thread start running when it is invoked
  - In essence, start() executes a call to run().

- sleep() method causes the thread from which it is called to suspend execution for the specified period of milliseconds.

static void sleep(long milliseconds)

throws InterruptedException

  - milliseconds: the number of milliseconds to suspend.

```
1.  class MyRunnable implements Runnable {
2.      String thrdName;
3.      MyRunnable(String name) {
4.          thrdName = name;
5.      }
6.      // Entry point of runnable.
7.      public void run() {
8.          System.out.println(thrdName + " starting.");
9.          try {
10.             for (int count = 0; count < 10; count++) {
11.                 Thread.sleep(400);
12.                 System.out.println("In " + thrdName +
13.                                    ", count is " + count);
14.             }
15.         } catch (InterruptedException exc) {
16.             System.out.println(thrdName + " interrupted.");
17.         }
18.         System.out.println(thrdName + " terminating.");
19.     }
20. }
```

```
1.  class UseThreads {
2.      public static void main(String args[]) {
3.          System.out.println("Main thread starting.");
4.          // First, construct a MyThread object.
5.          MyRunnable mr = new MyRunnable("Child #1");
6.          // Next, construct a thread from that object.
7.          Thread newThrd = new Thread(mr);
8.          // Finally, start execution of the thread.
9.          newThrd.start();
10.         for (int i = 0; i < 50; i++) {
11.             System.out.print(".");
12.             try {
13.                 Thread.sleep(100);
14.             } catch (InterruptedException exc) {
15.                 System.out.println("Main thread interrupted.");
16.             }
17.         }
18.         System.out.println("Main thread ending.");
19.     }
20. }
```

```
1.  class MyRunnable12 implements Runnable{
2.      Thread thrd;
3.      MyRunnable12(String name){ thrd = new Thread(this, name); }
4.      public static MyRunnable12 createAndStart(String name) {
5.          MyRunnable12 mr = new MyRunnable12(name);
6.          mr.thrd.start(); // start the thread
7.          return mr;
8.      }
9.      @Override
10.     public void run() {
11.         System.out.println(thrd.getName() + " starting.");
12.         try {
13.             for(int count = 0; count < 10; count++) {
14.                 Thread.sleep(400);
15.                 System.out.println("In " + thrd.getName() +
16.                                                 ", count is " + count);
17.             }
18.         }catch(InterruptedException exc) {
19.             System.out.println(thrd.getName() + " interrupted.");
20.         }
21.         System.out.println(thrd.getName() + " terminating.");
22.     }
23. }
```

Improved version

```
1.  public class ThreadVariations {
2.      public static void main(String[] args) {
3.          System.out.println("Main thread starting.");
4.          // Create and start a thread.
5.          MyRunnable12 mt = MyRunnable12.createAndStart("Child #1");
6.          for(int i =0; i < 50; i++) {
7.              System.out.print(".");
8.              try {
9.                  Thread.sleep(100);
10.             }catch(InterruptedException exc) {
11.                 System.out.println("Main thread interruped.");
12.             }
13.         }
14.         System.out.println("Main thread ending.");
15.     }
16. }
```

```
1.  class MyThread1 extends Thread {
2.      MyThread1(String name) {
3.          super(name); // name thread
4.          start(); // start the thread
5.      }
6.      public void run() {
7.          System.out.println(getName() + " starting.");
8.          try {
9.              for (int count = 0; count < 10; count++) {
10.                 Thread.sleep(400);
11.                 System.out.println("In " + getName() +
12.                                         ", count is " + count);
13.             }
14.         }catch (InterruptedException exc) {
15.             System.out.println(getName() + " interrupted.");
16.         }
17.         System.out.println(getName() + " terminating.");
18.     }
19. }
```

```
1.  class ExtendThread {
2.      public static void main(String args[]) {
3.          System.out.println("Main thread starting.");
4.          MyThread1 mt = new MyThread1("Child #1");
5.          for (int i = 0; i < 50; i++) {
6.              System.out.print(".");
7.              try {
8.                  Thread.sleep(100);
9.              } catch (InterruptedException exc) {
10.                 System.out.println("Main thread interrupted.");
11.             }
12.         }
13.         System.out.println("Main thread ending.");
14.     }
15. }
```

```
1.  class MyThread2 implements Runnable {
2.      Thread thrd;
3.      MyThread2(String name) {
4.          thrd = new Thread(this, name); thrd.start();
5.      }
6.      public void run() {
7.          System.out.println(thrd.getName() + " starting.");
8.          try {
9.              for (int count = 0; count < 10; count++) {
10.                 Thread.sleep(400);
11.                 System.out.println("In " + thrd.getName()
12.                                    + ", count is " + count);
13.             }
14.         } catch (InterruptedException exc) {
15.             System.out.println(thrd.getName()
16.                                                + " interrupted.");
17.         }
18.         System.out.println(thrd.getName() + " terminating.");
19.     }
20. }
```

```
1.  class MoreThreads {
2.      public static void main(String args[]) {
3.          System.out.println("Main thread starting.");
4.
5.          MyThread2 mt1 = new MyThread2("Child #1");
6.          MyThread2 mt2 = new MyThread2("Child #2");
7.          MyThread2 mt3 = new MyThread2("Child #3");
8.
9.          for (int i = 0; i < 50; i++) {
10.             System.out.print(".");
11.             try {
12.                 Thread.sleep(100);
13.             } catch (InterruptedException exc) {
14.                 System.out.println("Main thread interrupted.");
15.             }
16.         }
17.         System.out.println("Main thread ending.");
18.     }
19. }
```

```
Main thread starting.
Child #2 starting.
.Child #1 starting.
Child #3 starting.
...In Child #1, count is 0
In Child #3, count is 0
In Child #2, count is 0
....In Child #2, count is 1
In Child #1, count is 1
In Child #3, count is 1
....In Child #3, count is 2
In Child #2, count is 2
In Child #1, count is 2
....In Child #3, count is 3
In Child #2, count is 3
In Child #1, count is 3
....In Child #3, count is 4
In Child #2, count is 4
In Child #1, count is 4
....In Child #2, count is 5
In Child #3, count is 5
In Child #1, count is 5
....In Child #1, count is 6
In Child #3, count is 6
In Child #2, count is 6
....In Child #3, count is 7
In Child #2, count is 7
In Child #1, count is 7
....In Child #3, count is 8
In Child #2, count is 8
In Child #1, count is 8
....In Child #3, count is 9
Child #3 terminating.
In Child #2, count is 9
Child #2 terminating.
In Child #1, count is 9
Child #1 terminating.
..........Main thread ending.
```

Runnable vs Thread

- The first idiom, which employs a Runnable object, is more general, because the Runnable object can subclass a class other than Thread.
- The second idiom is easier to use in simple applications, but is limited by the fact that your task class must be a descendant of Thread.

Runnable vs Thread

- This lesson focuses on the first approach, which separates the Runnable task from the Thread object that executes the task.
- Not only is this approach more flexible, but it is applicable to the high-level thread management APIs covered later.

Determining when a thread ends

- It is often useful to know when a thread has ended.
- Two means to determine if a thread has ended.

final boolean isAlive()

- Returns true if the thread upon which it is called is still running.
- It returns false otherwise.

final void join() throws InterruptedException

- Waits until the thread on which it is called terminates.
- Additional forms of join() allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate.

```
1.  class MoreThreads {
2.      /* version 2*/
3.      public static void main(String args[]) {
4.        System.out.println("Main thread starting.");
5.        MyThread2 mt1 = new MyThread2("Child #1");
6.        MyThread2 mt2 = new MyThread2("Child #2");
7.        MyThread2 mt3 = new MyThread2("Child #3");
8.        do {
9.          System.out.print(".");
10.         try { Thread.sleep(100);
11.         }catch(InterruptedException exc) {
12.           System.out.println("Main thread interrupted.");
13.         }
14.       } while (mt1.thrd.isAlive() ||
15.                mt2.thrd.isAlive() ||
16.           mt3.thrd.isAlive());
17.       System.out.println("Main thread ending.");
18.     }
19. }
```

```
1.  class MyRunnable3 implements Runnable {
2.      Thread thrd;
3.      MyRunnable3(String name) {
4.          thrd = new Thread(this, name);thrd.start();
5.      }
6.      public void run() {
7.          System.out.println(thrd.getName() + " starting.");
8.          try {
9.              for (int count = 0; count < 10; count++) {
10.                 Thread.sleep(400);
11.                 System.out.println("In " + thrd.getName()
12.                                         + ", count is " + count);
13.             }
14.         } catch (InterruptedException exc) {
15.             System.out.println(thrd.getName()
16.                                                 + " interrupted.");
17.         }
18.         System.out.println(thrd.getName() + " terminating.");
19.     }
20. }
```

```
1.  class JoinThreads {
2.      public static void main(String args[]) {
3.          System.out.println("Main thread starting.");
4.          MyRunnable3 mt1 = new MyRunnable3("Child #1");
5.          MyRunnable3 mt2 = new MyRunnable3("Child #2");
6.          MyRunnable3 mt3 = new MyRunnable3("Child #3");
7.          try {
8.              mt1.thrd.join();
9.              System.out.println("Child #1 joined.");
10.             mt2.thrd.join();
11.             System.out.println("Child #2 joined.");
12.             mt3.thrd.join();
13.             System.out.println("Child #3 joined.");
14.         } catch (InterruptedException exc) {
15.             System.out.println("Main thread interrupted.");
16.         }
17.         System.out.println("Main thread ending.");
18.     }
19. }
```

```
Main thread starting.
Child #1 starting.
Child #2 starting.
Child #3 starting.
In Child #2, count is 0
In Child #1, count is 0
In Child #3, count is 0
In Child #3, count is 1
In Child #1, count is 1
In Child #2, count is 1
In Child #3, count is 2
In Child #2, count is 2
In Child #1, count is 2
In Child #1, count is 3
In Child #3, count is 3
In Child #2, count is 3
In Child #1, count is 4
In Child #2, count is 4
In Child #3, count is 4
In Child #1, count is 5
In Child #2, count is 5
In Child #3, count is 5
In Child #1, count is 6
In Child #3, count is 6
In Child #2, count is 6
In Child #3, count is 7
In Child #1, count is 7
In Child #2, count is 7
In Child #2, count is 8
In Child #3, count is 8
In Child #1, count is 8
In Child #3, count is 9
In Child #2, count is 9
Child #2 terminating.
Child #3 terminating.
In Child #1, count is 9
Child #1 terminating.
Child #1 joined.
Child #2 joined.
Child #3 joined.
Main thread ending.
```

Join

- The join method allows one thread to wait for the completion of another.
- If t is a Thread object whose thread is currently executing, t.join(); causes the current thread to pause execution until t's thread terminates.

Thread priorities

- Each thread has associated with it a priority setting.
  - A thread's priority determines, in part, how much CPU time a thread receives relative to the other active threads.
- In general: over a given period of time, low-priority threads receive little. High-priority threads receive a lot.
- It is important to understand that another factors also affect how much CPU time a thread receives.
- When a child thread is started, its priority setting is equal to that of its parent thread.
- You can change a thread's priority by

```
final void setPriority(int level)
```

level specifies the new priority setting for the calling thread.

- The value of level must be within the range MIN_PRIORITY (1) and MAX_PRIORITY (10).
- Default priority: NORM_PRIORITY (5)

- You can obtain the current priority setting by

```
final int getPriority()
```

```
1.  class Priority implements Runnable {
2.      int count; Thread thrd;
3.      static boolean stop = false; static String currentName;
4.      Priority(String name) {
5.          thrd = new Thread(this, name); count = 0;
6.          currentName = name;
7.      }
8.      public void run() {
9.          System.out.println(thrd.getName() + " starting.");
10.         do {
11.           count++;
12.         } while (stop == false && count < 10000000);
13.         stop = true;
14.         System.out.println("\n" + thrd.getName()
15.                                                 + " terminating.");
16.     }
17. }
18.
19.
20.
```

```
1.  class PriorityDemo {
2.      public static void main(String args[]) {
3.          Priority mt1 = new Priority("High Priority");
4.          Priority mt2 = new Priority("Low Priority");
5.          Priority mt3 = new Priority("Normal Priority #1");
6.          Priority mt4 = new Priority("Normal Priority #2");
7.          Priority mt5 = new Priority("Normal Priority #3");
8.          mt1.thrd.setPriority(Thread.NORM_PRIORITY + 2);
9.          mt2.thrd.setPriority(Thread.NORM_PRIORITY - 2);
10.         mt1.thrd.start(); mt2.thrd.start();
11.         mt3.thrd.start(); mt4.thrd.start(); mt5.thrd.start();
12.         try {
13.             mt1.thrd.join(); mt2.thrd.join();
14.             mt3.thrd.join(); mt4.thrd.join(); mt5.thrd.join();
15.         } catch (InterruptedException exc) {
16.             System.out.println("Main thread interrupted.");
17.         }
18.         ....
```

```
1.              ...
2.              System.out.println("\nHigh priority thread counted to " + mt1.count);
3.              System.out.println("Low priority thread counted to " + mt2.count);
4.              System.out.println("1st Normal priority thread counted to " + mt3.count);
5.              System.out.println("2nd Normal priority thread counted to " + mt4.count);
6.              System.out.println("3rd Normal priority thread counted to " + mt5.count);
7.          }
8.      }
9.
10.
11.
12.                 High priority thread counted to 10000000
                    Low priority thread counted to 6444752
                    1st Normal priority thread counted to 7071170
                    2nd Normal priority thread counted to 9116802
                    3rd Normal priority thread counted to 7330314
```

Plan

1. Multithreading fundamentals
2. Synchronization

Multithreaded problems

- Threads communicate primarily by sharing access to fields and the objects reference fields refer to.
- This form of communication is extremely efficient, but makes two kinds of errors possible: thread interference and memory consistency errors.
- The tool needed to prevent these errors is synchronization.

Synchronization

- When using multiple threads, it is sometimes necessary to coordinate the activities of two or more → synchronization.
- Reason for using synchronization:
  - When two or more threads need access to a **shared** resource that can be used by only one thread at a time.
  - When one thread is waiting for an event that is caused by another thread.
- Key to synchronization: monitor (controls access to an object)
  - A monitor works by implementing the concept of a **lock**.
  - When an object is locked by one thread, no other thread can gain access to the object. When the thread exits, the object is unlocked and is available for use by another thread.

Synchronization

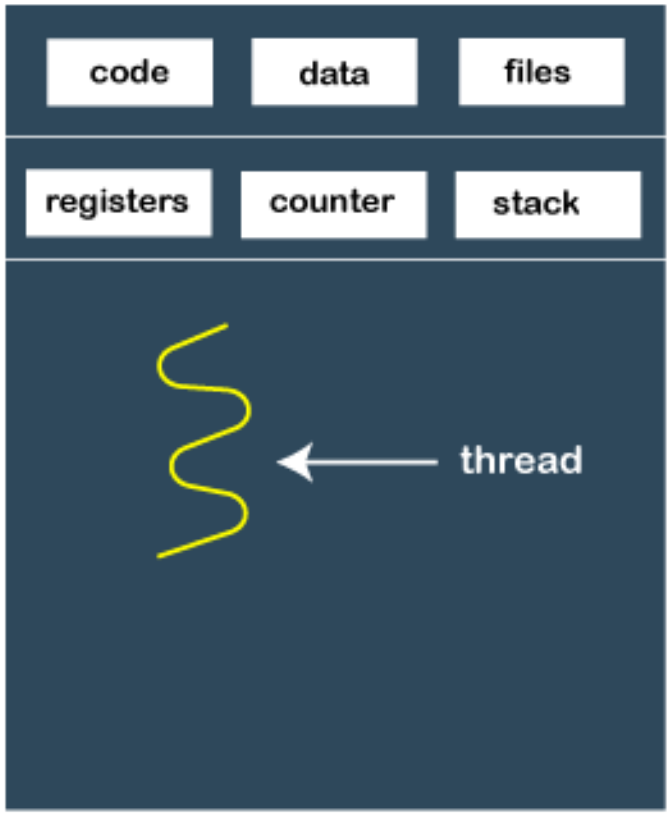
- All objects in Java have a monitor.
  - This feature is built into the Java language itself.

  →all objects can be synchronized.
- Keyword: synchronized.
- Two ways to synchronize:
  - synchronized method & synchronized statement.
  - both use the synchronized keyword.

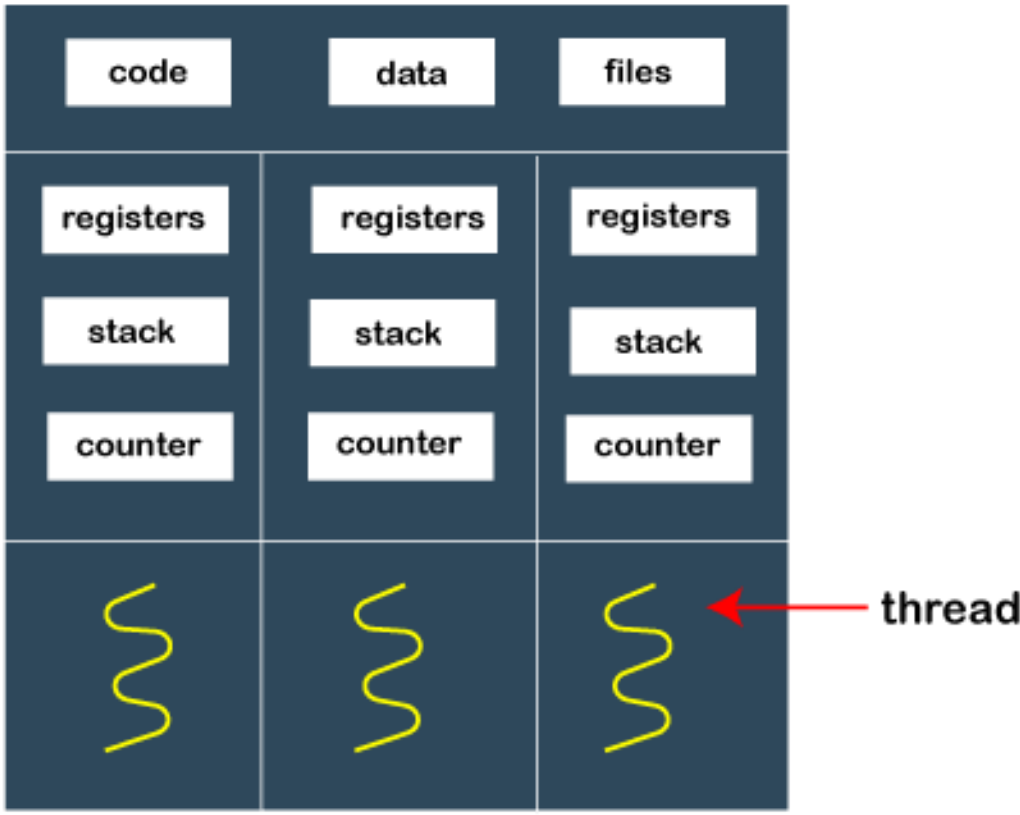
Synchronized methods

- Use the synchronized keyword.
- **When that method is called**: the calling thread enters the object's monitor, which then locks the object.
- **While locked**: no other thread can enter the method, or enter any other synchronized method defined by the object's class.
- **When the thread returns from the method**: the monitor unlocks the object, allowing it to be used by the next thread.

→ synchronization is achieved with virtually no programming effort on your part.



Single-threaded process



Multi-threaded process

```
1.  class SumArray {
2.      private int sum;
3.      synchronized int sumArray(int nums[]) {
4.          sum = 0; // reset sum
5.          for (int i = 0; i < nums.length; i++) {
6.              sum += nums[i];
7.              System.out.println("Running total for "
8.                      + Thread.currentThread().getName()
9.                      + " is " + sum);
10.             try {
11.                 Thread.sleep(10); // allow task-switch
12.             } catch (InterruptedException exc) {
13.                 System.out.println("Thread interrupted.");
14.             }
15.         }
16.         return sum;
17.     }
18. }
```

```
1.  class MyThread4 implements Runnable {
2.      Thread thrd;
3.      static SumArray sa = new SumArray();
4.      int a[]; int answer;
5.      MyThread4(String name, int nums[]) {
6.          thrd = new Thread(this, name);
7.          a = nums; thrd.start(); // start the thread
8.      }
9.      public void run() {
10.         int sum;
11.         System.out.println(thrd.getName() + " starting.");
12.         answer = sa.sumArray(a);
13.         System.out.println("Sum for " + thrd.getName()
14.                                 + " is " + answer);
15.         System.out.println(thrd.getName() + " terminating.");
16.     }
17. }
```

```
1.  class Sync {
2.      public static void main(String args[]) {
3.          int a[] = { 1, 2, 3, 4, 5 };
4.          MyThread4 mt1 = new MyThread4("Child #1", a);
5.          MyThread4 mt2 = new MyThread4("Child #2", a);
6.
7.          try {
8.              mt1.thrd.join();
9.              mt2.thrd.join();
10.         } catch (InterruptedException exc) {
11.             System.out.println("Main thread interrupted.");
12.         }
13.     }
14. }
```

Which objects are locked here?

```
Child #2 starting.
Running total for Child #2 is 1
Child #1 starting.
Running total for Child #2 is 3
Running total for Child #2 is 6
Running total for Child #2 is 10
Running total for Child #2 is 15
Running total for Child #1 is 1
Sum for Child #2 is 15
Child #2 terminating.
Running total for Child #1 is 3
Running total for Child #1 is 6
Running total for Child #1 is 10
Running total for Child #1 is 15
Sum for Child #1 is 15
Child #1 terminating.
```

Child #2 runs first and completes fully.
After that, Child #1 runs completely, with no interpolation

Synchronized statement

- Although creating synchronized methods within classes that you create is an easy and effective means of achieving synchronization, it will not work in all cases.
- Example: you might want to synchronize access to some method that is not modified by synchronized.
  - This can occur because you want to use a class that was not created by you but by a third party, and you do not have access to the source code.

→It is not possible for you to add synchronized to the appropriate methods within the class: simply put calls to the methods defined by this class inside a synchronized block.

- General form of a synchronized block:

```
synchronized(objref) {
    // statements to be synchronized
}
```

  - objref is a reference to the object being synchronized.
  - Once a synchronized block has been entered, no other thread can call a synchronized method on the object referred to by objref until the block has been exited.

```
1.  class SumArray1 {
2.      private int sum;
3.      int sumArray1(int nums[]) {
4.          sum = 0; // reset sum
5.          for (int i = 0; i < nums.length; i++) {
6.              sum += nums[i];
7.              System.out.println("Running total for "
8.                      + Thread.currentThread().getName()
9.                      + " is " + sum);
10.             try {
11.                 Thread.sleep(10); // allow task-switch
12.             } catch (InterruptedException exc) {
13.                 System.out.println("Thread interrupted.");
14.             }
15.         }
16.         return sum;
17.     }
18. }
```

```
1.  class MyThread5 implements Runnable {
2.      Thread thrd;
3.      static SumArray sa = new SumArray();
4.      int a[]; int answer;
5.      MyThread5(String name, int nums[]) {
6.          thrd = new Thread(this, name);
7.          a = nums; thrd.start(); // start the thread
8.      }
9.      public void run() {
10.         int sum;
11.         System.out.println(thrd.getName() + " starting.");
12.         // synchronize calls to sumArray()
13.         synchronized (sa) { answer = sa.sumArray(a); }
14.         System.out.println("Sum for " + thrd.getName()
15.                                 + " is " + answer);
16.         System.out.println(thrd.getName() + " terminating.");
17.     }
18. }
```

```
1.  class Sync1 {
2.      public static void main(String args[]) {
3.          int a[] = { 1, 2, 3, 4, 5 };
4.          MyThread5 mt1 = new MyThread5("Child #1", a);
5.          MyThread5 mt2 = new MyThread5("Child #2", a);
6.
7.          try {
8.              mt1.thrd.join();
9.              mt2.thrd.join();
10.         } catch (InterruptedException exc) {
11.             System.out.println("Main thread interrupted.");
12.         }
13.     }
14. }
```

Thread communication

- A thread called **T** is executing inside a synchronized method and needs access to a resource called **R** that is temporarily unavailable. What should **T** do?
  - If T enters some form of polling loop that waits for R, T ties up the object, preventing other threads' access to it → partially defeats the advantages of programming for a multithreaded environment.
  - A better solution: have T temporarily relinquish control of the object, allowing another thread to run. When R becomes available, T can be notified and resume execution → interthread communication (one thread can notify another that it is blocked and be notified that it can resume execution).
- Java supports interthread communication with the wait(), notify(), and notifyAll() methods.

final void wait()

throws InterruptedException

final void wait(long millis)

throws InterruptedException

final void wait(long millis, int nanos)

throws InterruptedException

final void notify()

resumes one waiting thread

final void notifyAll()

notifies all threads, with the scheduler determining which thread gains access to the object.

```
1.  class TickTock {
2.      String state; // contains the state of the clock
3.      synchronized void tick(boolean running) {
4.          if (!running) { state = "ticked"; notify(); return; }
5.          System.out.print("Tick ");
6.          state = "ticked";  notify();
7.          try { while (!state.equals("tocked")) wait();
8.          } catch (InterruptedException exc) {
9.              System.out.println("Thread interrupted.");
10.         }
11.     }
12.     synchronized void tock(boolean running) {
13.         if (!running) {  state = "tocked"; notify();  return; }
14.         System.out.println("Tock");
15.         state = "tocked";  notify();
16.         try { while (!state.equals("ticked")) wait();
17.         } catch (InterruptedException exc) {
18.             System.out.println("Thread interrupted.");
19.         }
20.     }
21. }
```

```
1.  class MyThread6 implements Runnable {
2.      Thread thrd; TickTock ttOb;
3.      MyThread6(String name, TickTock tt) {
4.          thrd = new Thread(this, name);
5.          ttOb = tt; thrd.start(); // start the thread
6.      }
7.      public void run() {
8.          if (thrd.getName().compareTo("Tick") == 0) {
9.              for (int i = 0; i < 5; i++)
10.                 ttOb.tick(true);
11.             ttOb.tick(false);
12.         } else {
13.             for (int i = 0; i < 5; i++)
14.                 ttOb.tock(true);
15.             ttOb.tock(false);
16.         }
17.     }
18. }
```

```
1.  class ThreadCom {
2.      public static void main(String args[]) {
3.          TickTock tt = new TickTock();
4.          MyThread6 mt1 = new MyThread6("Tick", tt);
5.          MyThread6 mt2 = new MyThread6("Tock", tt);
6.
7.          try {
8.              mt1.thrd.join();
9.              mt2.thrd.join();
10.         } catch (InterruptedException exc) {
11.             System.out.println("Main thread interrupted.");
12.         }
13.     }
14. }
```

Deadlock, race condition

- Deadlock is, as the name implies, a situation in which one thread is waiting for another thread to do something, but that other thread is waiting on the first → both threads are suspended, waiting on each other, and neither executes.
- A race condition occurs when two (or more) threads attempt to access a shared resource at the same time, without proper synchronization.

Suspending, resuming, and stopping threads

- final void resume()
- final void suspend()
- final void stop()
- While these methods seem to be a perfectly reasonable and convenient approach to managing the execution of threads, they must no longer be used.
  - Deprecated by Java 2.

```
1.  class MyThread8 implements Runnable {
2.      Thread thrd; boolean suspended; boolean stopped;
3.      MyThread8(String name) {
4.          thrd = new Thread(this, name);
5.          suspended = false; stopped = false; thrd.start();
6.      }
7.      public void run() {
8.          System.out.println(thrd.getName() + " starting.");
9.          try {
10.             for (int i = 1; i < 1000; i++) {
11.                 System.out.print(i + " ");
12.                 if ((i % 10) == 0) {
13.                     System.out.println(); Thread.sleep(250);
14.                 }
15.                 synchronized (this){
16.                     while(suspended){wait();}
17.                     if(stopped) break;
18.                 }
19.             }
```

```
            } catch (InterruptedException exc) {
                System.out.println(thrd.getName() + " interrupted.");
            }
            System.out.println(thrd.getName() + " exiting.");
        }
        synchronized void mystop() {
            stopped = true;
            suspended = false;
            notify();
        }
        synchronized void mysuspend() {
            suspended = true;
        }
        synchronized void myresume() {
            suspended = false;
            notify();
        }
}
```

```
1.  class Suspend {
2.      public static void main(String args[]) {
3.          MyThread8 ob1 = new MyThread8("My Thread");
4.          try {
5.              Thread.sleep(1000);
6.              //let ob1 thread start executing
7.              ob1.mysuspend();
8.              System.out.println("Suspending thread.");
9.              Thread.sleep(1000);
10.
11.             ob1.myresume();
12.             System.out.println("Resuming thread.");
13.             Thread.sleep(1000);
14.
15.             ob1.mysuspend();
16.             System.out.println("Suspending thread.");
17.             Thread.sleep(1000);
```

```
1.                  ob1.myresume();
2.                  System.out.println("Resuming thread.");
3.                  Thread.sleep(1000);
4.
5.                  ob1.mysuspend();
6.                  System.out.println("Stopping thread.");
7.                  ob1.mystop();
8.              } catch (InterruptedException e) {
9.                  System.out.println("Main thread Interrupted");
10.             }
11.             try {
12.                 ob1.thrd.join();
13.             } catch (InterruptedException e) {
14.                 System.out.println("Main thread Interrupted");
15.             }
16.             System.out.println("Main thread exiting.");
17.         }
18. }
```

```
My Thread starting.
1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
Suspending thread.
Resuming thread.
41 42 43 44 45 46 47 48 49 50
51 52 53 54 55 56 57 58 59 60
61 62 63 64 65 66 67 68 69 70
71 72 73 74 75 76 77 78 79 80
Suspending thread.
Resuming thread.
81 82 83 84 85 86 87 88 89 90
91 92 93 94 95 96 97 98 99 100
101 102 103 104 105 106 107 108 109 110
111 112 113 114 115 116 117 118 119 120
Stopping thread.
My Thread exiting.
Main thread exiting.
```

Daemon Threads

- Daemon threads in Java are background threads that perform supporting tasks for user threads.
- They run alongside the main application and are terminated automatically when all user threads (non-daemon threads) finish execution.
- Ex: background logging services.

DaemonThreadDemo

- DaemonThreadDemo.java

SupportingDaemonDemo

- SupportingDaemonDemo.java

Multithreading in Swing application

- In a typical Swing application, there should be 3 threads.
- The main thread is for the main() function.
- The event distpatching thread is used to handle Swing events.
- The heavy part should not be run in either the main thead or the event dispatching thread.
  - It makes to app unresponded.
- The heavy code should be placed in another thread.
  - By using SwingWorker class.
  - By creating your own thread.

ExecutorServices


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

class Task implements Runnable {
    private final int taskId;

    public Task(int taskId) {
        this.taskId = taskId;
    }

    @Override
    public void run() {
        System.out.println("Task " + taskId + " is running on thread: " + Thread.currentThread().getName());
        try {
            Thread.sleep(1000); // Simulate work with a 1-second delay
        } catch (InterruptedException e) {
            System.err.println("Task " + taskId + " was interrupted.");
        }
        System.out.println("Task " + taskId + " completed.");
    }
}
```

ExecutorServices


```
public class ExecutorServiceDemo {
    public static void main(String[] args) {
        // Create a fixed thread pool with 3 threads
        ExecutorService executorService = Executors.newFixedThreadPool(3);

        // Submit 10 tasks to the executor service
        for (int i = 0; i < 10; i++) {
            Task task = new Task(i);
            executorService.submit(task);
        }

        // Shut down the executor service
        executorService.shutdown();
        System.out.println("All tasks submitted.");
    }
}
```

QUESTION ?