Monitors

CSCI 201
Principles of Software Development

Jeffrey Miller, Ph.D.
jeffrey.miller@usc.edu
Outline

• Monitors
• Program
Monitor Overview

- A **monitor** is an object with mutual exclusion and synchronization capabilities
  - All objects in Java can be monitors (see Object API on next slide)
- The **synchronized** keyword enables the use of monitors
  - Methods or individual blocks of code in Java can be **synchronized**
- A thread enters the monitor by acquiring a **lock** on it and exits by releasing the lock
- An object has the monitor functionality invoked once a thread locks it using the **synchronized** keyword
# Object Class

## Method Summary

<table>
<thead>
<tr>
<th>Modifier and Type</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>protected Object</strong></td>
<td><strong>clone()</strong>&lt;br&gt;Copies and returns a copy of this object.</td>
</tr>
<tr>
<td><strong>boolean</strong></td>
<td><strong>equals(Object obj)</strong>&lt;br&gt;Indicates whether some other object is &quot;equal to&quot; this one.</td>
</tr>
<tr>
<td><strong>protected void</strong></td>
<td><strong>finalize()</strong>&lt;br&gt;Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.</td>
</tr>
<tr>
<td><strong>Class&lt;?&gt;</strong></td>
<td><strong>getClass()</strong>&lt;br&gt;Returns the runtime class of this Object.</td>
</tr>
<tr>
<td><strong>int</strong></td>
<td><strong>hashCode()</strong>&lt;br&gt;Returns a hash code value for the object.</td>
</tr>
<tr>
<td><strong>void</strong></td>
<td><strong>notify()</strong>&lt;br&gt;Wakes up a single thread that is waiting on this object's monitor.</td>
</tr>
<tr>
<td><strong>void</strong></td>
<td><strong>notifyAll()</strong>&lt;br&gt;Wakes up all threads that are waiting on this object's monitor.</td>
</tr>
<tr>
<td><strong>String</strong></td>
<td><strong>toString()</strong>&lt;br&gt;Returns a string representation of the object.</td>
</tr>
<tr>
<td><strong>void</strong></td>
<td><strong>wait()</strong>&lt;br&gt;Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.</td>
</tr>
<tr>
<td><strong>void</strong></td>
<td><strong>wait(long timeout)</strong>&lt;br&gt;Causes the current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.</td>
</tr>
<tr>
<td><strong>void</strong></td>
<td><strong>wait(long timeout, int nanos)</strong>&lt;br&gt;Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.</td>
</tr>
</tbody>
</table>
Thread States

- **Start**: When a thread is started
- **Ready**: When the OS/JVM switches the thread into the CPU
- **Running**: When the OS/JVM switches the thread out of the CPU or the thread yields the CPU
- **Sleeping**: When the amount of time specified for sleeping has elapsed
- **Waiting**: When a thread is signaled based on the resource on which it is waiting
- **Dead**: When a thread waits on a resource to become available
- **Dead**: When a thread puts itself to sleep for a certain amount of time
- **Dead**: When a thread has completed execution

**Thread States**

- **Start**
  - When a thread is started
- **Ready**
  - When the OS/JVM switches the thread into the CPU
- **Running**
  - When the OS/JVM switches the thread out of the CPU or the thread yields the CPU
- **Sleeping**
  - When the amount of time specified for sleeping has elapsed
- **Waiting**
  - When a thread is signaled based on the resource on which it is waiting
- **Dead**
  - When a thread waits on a resource to become available
  - When a thread puts itself to sleep for a certain amount of time
  - When a thread has completed execution
Monitor Overview

- The **Object** class has methods on the monitor that can be called, though this should be used cautiously.
  - Monitor functionality is implemented in the `synchronized` keyword, and calling monitor methods directly may produce different results when synchronizing the object.

- A thread can call `wait()` inside a monitor, which will release the lock on the object.
  - That thread must then be awakened using `notify()` or `notifyAll()` from another thread to be moved back into the Ready state.
**synchronized Keyword**

- The *synchronized* keyword puts a restriction on a method or block of code that only one thread can be inside that method or block at a time
  - No other thread will be able to enter that method or block of code if another thread is currently executing inside of it, *regardless of whether the thread is currently in the CPU or not*

- Before a block of *synchronized* code can execute, a lock must be obtained
  - A *lock* is a binary mechanism for exclusive use of a resource
  - *Locks* can only be acquired by one object at a time
synchronized Methods

- **synchronized Non-Static Methods**
  - The lock obtained is on the **object** on which the method was invoked
  - When a thread invokes a **synchronized** instance method of an object, the lock of that **object** is acquired first, then the method is executed, then the lock is released
    - Another thread invoking any **synchronized** method or block of code **on that object** is blocked until the lock is released

- **synchronized static Methods**
  - The lock obtained is on the **class** on which the method was invoked (even if the method was invoked from an instance of the class, which would be bad programming)
  - When a thread invokes a **synchronized static** method of a class, the lock on that **class** is acquired first, then the method is executed, then the lock is released
    - Another thread invoking any **synchronized static** method or block of code **on that class** is blocked until the lock is released
class SyncClass {
    synchronized void foo() {
        // foo line 1
        // foo line 2
    }
    synchronized void bar() {
        // bar line 1
        // bar line 2
    }
    void meth() {
        // meth line 1
        // meth line 2
    }
}

public class MainClass extends Thread {
    private static SyncClass sc;
    private int num;
    public MainClass(int num) {
        this.num = num;
    }
    public static void main(String [] args) {
        sc = new SyncClass();
        Thread t1 = new MainClass(1);
        Thread t2 = new MainClass(2);
        t1.start();
        t2.start();
    }
    public void run() {
        sc.foo();
    }
}

Thread t1 calls sc.foo(); and gets switched out of the CPU after line 3
Thread t2 calls sc.foo();
Will t2 be able to execute?

Not until t1 releases the lock on sc
Synchronization Example #2

1 class SyncClass {
2   synchronized void foo() {
3     // foo line 1
4     // foo line 2
5   }
6   synchronized void bar() {
7     // bar line 1
8     // bar line 2
9   }
10  void meth() {
11     // meth line 1
12     // meth line 2
13  }
14 }

Thread t1 calls \texttt{sc.foo()}; and gets switched out of the CPU after line 3

Thread t2 calls \texttt{sc.bar()};

Will t2 be able to execute?

15 public class MainClass extends Thread {
16   private static SyncClass sc;
17   private int num;
18   public MainClass(int num) {
19     this.num = num;
20   }
21   public static void main(String [] args) {
22     sc = new SyncClass();
23     Thread t1 = new MainClass(1);
24     Thread t2 = new MainClass(2);
25     t1.start();
26     t2.start();
27   }
28   public void run() {
29     if (num == 1) {
30       sc.foo();
31     }
32     else {
33       sc.bar();
34     }
35   }
36 }

\textbf{Not until t1 releases the lock on sc}
Thread t1 calls `sc.foo()`; and gets switched out of the CPU after line 3

Thread t2 calls `sc2.foo()`;

Will t2 be able to execute?

Yes, since t1 acquires the lock on `sc` and t2 acquires the lock on `sc2`
Thread t1 calls `sc.foo()`; and gets switched out of the CPU after line 3

Thread t2 calls `sc2.bar()`;

Will t2 be able to execute?

Not until t1 releases the lock on `SyncClass`
Thread t1 calls `SyncClass.foo()`; and gets switched out of the CPU after line 3

Thread t2 calls `SyncClass.bar()`;

Will t2 be able to execute?

Not until t1 releases the lock on SyncClass
Thread t1 calls `SyncClass.foo()`; and gets switched out of the CPU after line 3

Thread t2 calls `sc.bar()`;

Will t2 be able to execute?

Yes, since t1 has the lock on `SyncClass` and t2 has the lock on `sc`
Synchronization Example #7

```java
1  class SyncClass {
2     static synchronized void foo() {
3       // foo line 1
4       // foo line 2
5     }
6     synchronized void bar() {
7         meth();
8       // bar line 2
9     }
10    void meth() {
11       // meth line 1
12       // meth line 2
13    }
14  }

Thread t1 calls `sc.bar();` and gets switched out of the CPU after line 11 in `meth();`

Thread t2 calls `sc.meth();`

Will t2 be able to execute?

Yes, since t1 has the lock on `sc` and t2 doesn’t need a lock
```

```java
15  public class MainClass extends Thread {
16     private static SyncClass sc;
17     private static SyncClass sc2;
18     private int num;
19     public MainClass(int num) {
20         this.num = num;
21     }
22     public static void main(String [] args) {
23         sc = new SyncClass();
24         sc2 = new SyncClass();
25         Thread t1 = new MainClass(1);
26         Thread t2 = new MainClass(2);
27         t1.start();
28         t2.start();
29     }
30     public void run() {
31         if (num == 1) {
32             sc.bar();
33         }
34         else {
35             sc.meth();
36         }
37     }
38  }
```

Synchronization Example #8

1. class SyncClass {
2.     static synchronized void foo() {
3.         // foo line 1
4.         // foo line 2
5.     }
6.     synchronized void bar() {
7.         meth();
8.         // bar line 2
9.     }
10.    void meth() {
11.        // meth line 1
12.        // meth line 2
13.    }
14. }

Thread t1 calls `sc.bar()`; and gets switched out of the CPU after line 11 in `meth()`.

Thread t2 calls `sc.bar()`;

Will t2 be able to execute?

15. public class MainClass extends Thread {
16.    private static SyncClass sc;
17.    private static SyncClass sc2;
18.    private int num;
19.    public MainClass(int num) {
20.        this.num = num;
21.    }
22.    public static void main(String[] args) {
23.        sc = new SyncClass();
24.        sc2 = new SyncClass();
25.        Thread t1 = new MainClass(1);
26.        Thread t2 = new MainClass(2);
27.        t1.start();
28.        t2.start();
29.    }
30.    public void run() {
31.        if (num == 1) {
32.            sc.bar();
33.        }
34.        else {
35.            sc.bar();
36.        }
37.    }
38. }

Not until t1 releases the lock on `sc`. t1 holds onto the lock on `sc` until it completes the `bar()` method.
synchronized Statements

- We do not need to synchronize entire methods if only a part of the method needs to be synchronized.
- A `synchronized` statement can be used to acquire a lock on any object (not just the current object) or on a class:
  ```java
  synchronized(obj) {
    // synchronized code
  }
  synchronized(String.class) {
    // synchronized code
  }
  ```
- The lock would have to be obtained on the object `obj` or the class before the code in that block could execute.
  - If the lock cannot be obtained, the thread will block at that line until it can obtain the lock.
- Note that any `synchronized` method can be converted into a `synchronized` block of code:
  ```java
  public synchronized void meth() {
    // code
  }
  ```
  ```java
  public void meth() {
    synchronized(this) {
      // code
    }
  }
  ```
AddAPenny Example Revisited

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

public class AddAPenny implements Runnable {
    private static PiggyBank piggy = new PiggyBank();

    public void run() {
        piggy.deposit(1);
    }

    public static void main(String[] args) {
        ExecutorService executor = Executors.newCachedThreadPool();
        for (int i=0; i < 100; i++) {
            executor.execute(new AddAPenny());
        }
        executor.shutdown();
        // wait until all tasks are finished
        while(!executor.isTerminated()) {
            Thread.yield();
        }
        System.out.println("Balance = " + piggy.getBalance());
    }
}

class PiggyBank {
    private int balance = 0;
    public int getBalance() {
        return balance;
    }

    public void deposit(int amount) {
        int newBalance = balance + amount;
        Thread.yield();
        balance = newBalance;
    }
}
```

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

public class AddAPenny implements Runnable {
    private static PiggyBank piggy = new PiggyBank();

    public void run() {
        piggy.deposit(1);
    }

    public static void main(String[] args) {
        ExecutorService executor = Executors.newCachedThreadPool();
        for (int i = 0; i < 100; i++) {
            executor.execute(new AddAPenny());
        }
        executor.shutdown();
        // wait until all tasks are finished
        while (!executor.isTerminated()) {
            Thread.yield();
        }
        System.out.println("Balance = " + piggy.getBalance());
    }
}

class PiggyBank {
    private int balance = 0;
    public int getBalance() {
        return balance;
    }
    public synchronized void deposit(int amount) {
        int newBalance = balance + amount;
        Thread.yield();
        balance = newBalance;
    }
}
Outline

- Monitors
- Program
Program

- Download the `AddAndRemoveAPenny` code from the course web site and execute it
  - Make sure you understand why the output is what it is
- What modification could you make to the code to force it to hang if the total amount of withdrawals exceeds the total amount of deposits?
- Modify the code to remove having an equal number of threads that withdraw and deposit
  - Does the code always terminate in either case?
  - How can you make the code always terminate?