Locks

CSCI 201
Principles of Software Development

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

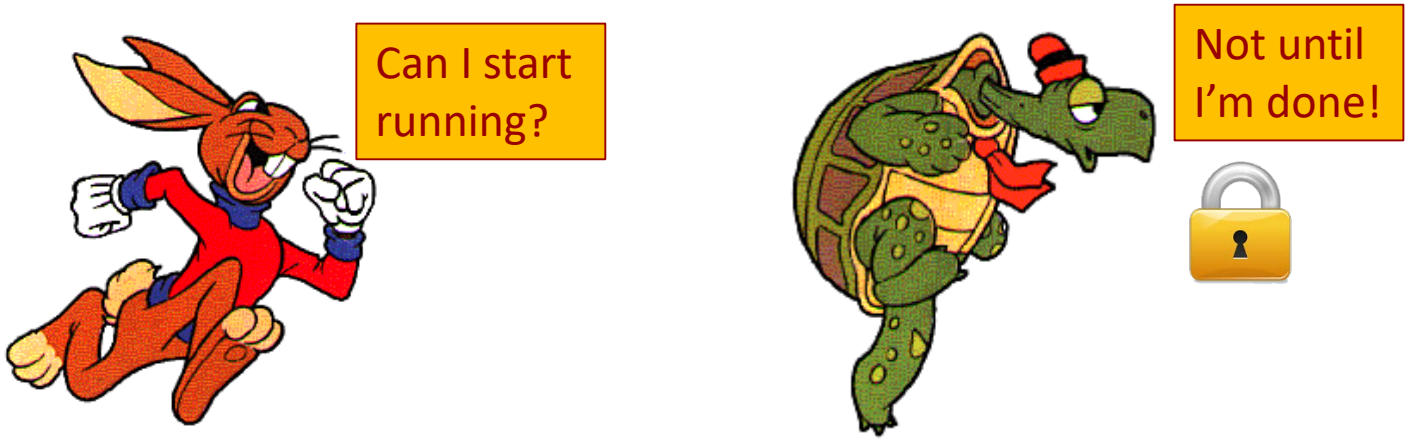
USC Viterbi
School of Engineering

Outline

- Locks and Conditions

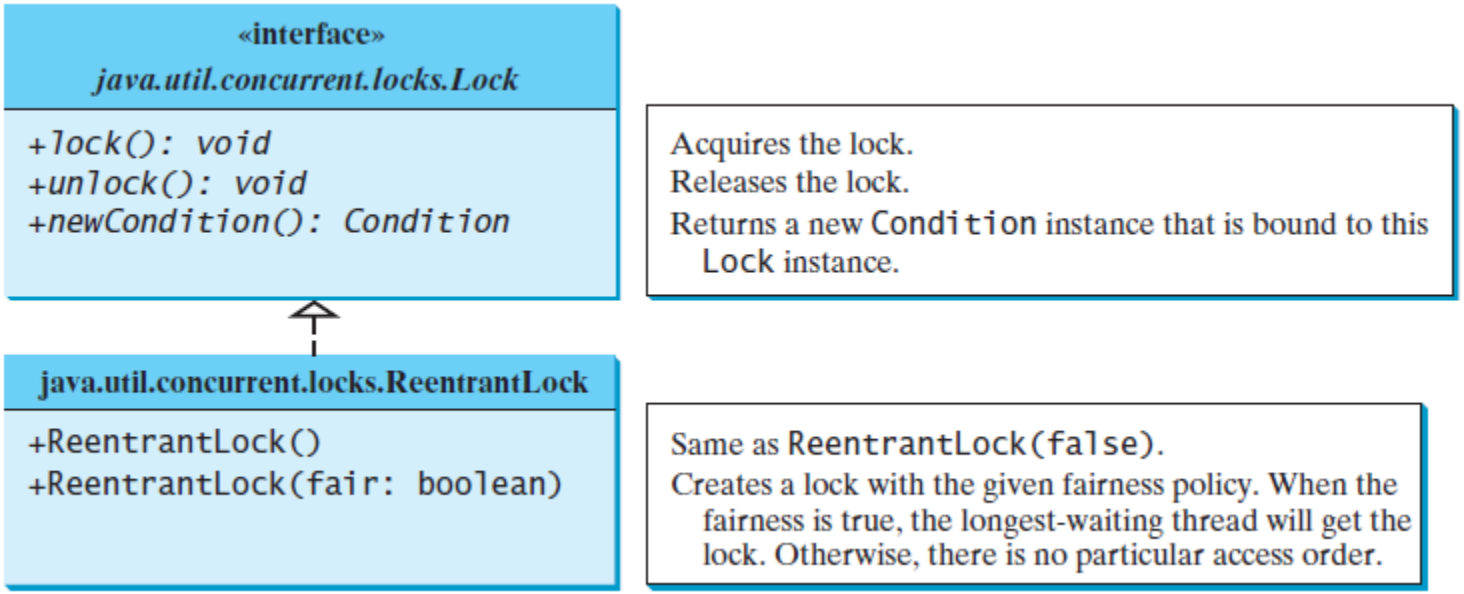
`synchronized` Keyword

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



Explicitly Acquiring Locks

- A `synchronized` method or block of code *implicitly* acquires a lock on the instance before executing
- Java gives us the ability to *explicitly* acquire locks using the `java.util.concurrent.locks.Lock` interface and `java.util.concurrent.locks.ReentrantLock` class



Reentrant Locks

- A reentrant lock is a lock that does not need to be acquired more than once if entering a second critical section on the same lock
  - Monitors use reentrant locks

```
class Reentrant {
  private synchronized void foo() {
    System.out.println("1");
    bar();
    System.out.println("3");
  }
  private synchronized void bar() {
    System.out.println("2");
  }
}
```

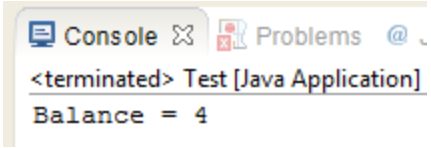
Once a thread enters `foo()`, it has obtained the lock on the object

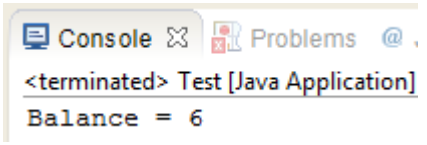
Since the thread already has the lock, it does not need to acquire it again to get into `bar()`

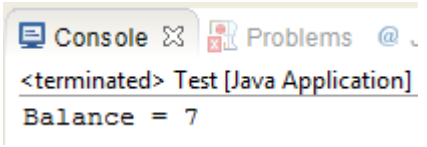
AddAPenny Example Revisited

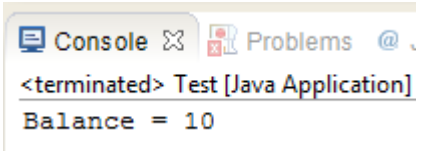
```
1   import java.util.concurrent.ExecutorService;
2   import java.util.concurrent.Executors;
3
4   public class AddAPenny implements Runnable {
5     private static PiggyBank piggy = new PiggyBank();
6
7     public void run() {
8       piggy.deposit(1);
9     }
10    public static void main(String [] args) {
11      ExecutorService executor = Executors.newCachedThreadPool();
12      for (int i=0; i < 100; i++) {
13        executor.execute(new AddAPenny());
14      }
15      executor.shutdown();
16      // wait until all tasks are finished
17      while(!executor.isTerminated()) {
18        Thread.yield();
19      }
20      System.out.println("Balance = " + piggy.getBalance());
21    }
22  }
23
24  class PiggyBank {
25    private int balance = 0;
26    public int getBalance() {
27      return balance;
28    }
29    public void deposit(int amount) {
30      int newBalance = balance + amount;
31      Thread.yield();
32      balance = newBalance;
33    }
34  }
```

4 Executions





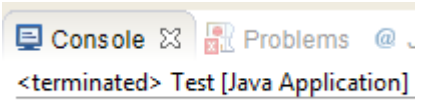


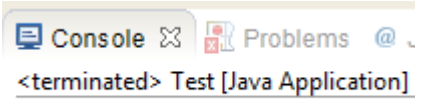


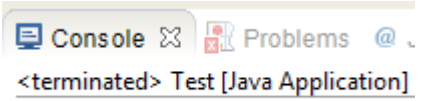
AddAPenny with Synchronization

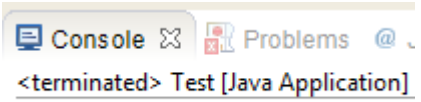
```
1   import java.util.concurrent.ExecutorService;
2   import java.util.concurrent.Executors;
3
4   public class AddAPenny implements Runnable {
5     private static PiggyBank piggy = new PiggyBank();
6
7     public void run() {
8       piggy.deposit(1);
9     }
10    public static void main(String [] args) {
11      ExecutorService executor = Executors.newCachedThreadPool();
12      for (int i=0; i < 100; i++) {
13        executor.execute(new AddAPenny());
14      }
15      executor.shutdown();
16      // wait until all tasks are finished
17      while(!executor.isTerminated()) {
18        Thread.yield();
19      }
20      System.out.println("Balance = " + piggy.getBalance());
21    }
22  }
23
24  class PiggyBank {
25    private int balance = 0;
26    public int getBalance() {
27      return balance;
28    }
29    public synchronized void deposit(int amount) {
30      int newBalance = balance + amount;
31      Thread.yield();
32      balance = newBalance;
33    }
34  }
```

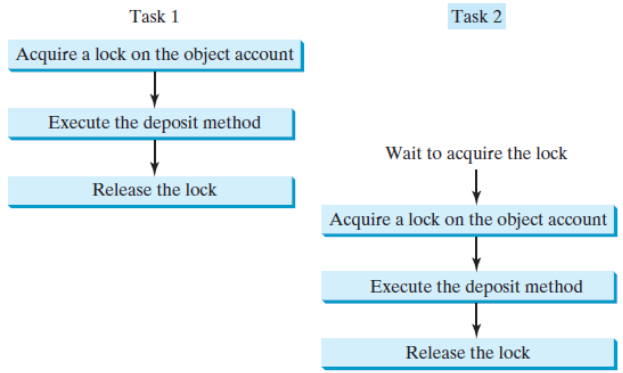
4 Executions







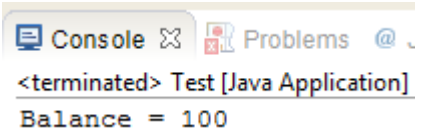


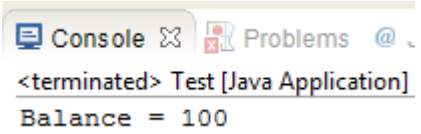


Explicitly Acquiring Locks Example

```
1   import java.util.concurrent.*;
2   import java.util.concurrent.locks.*;
3
4   public class AddAPenny implements Runnable {
5     private static PiggyBank piggy = new PiggyBank();
6
7     public void run() {
8       piggy.deposit(1);
9     }
10    public static void main(String [] args) {
11      ExecutorService executor = Executors.newCachedThreadPool();
12      for (int i=0; i < 100; i++) {
13        executor.execute(new AddAPenny());
14      }
15      executor.shutdown();
16      // wait until all tasks are finished
17      while(!executor.isTerminated()) {
18        Thread.yield();
19      }
20      System.out.println("Balance = " + piggy.getBalance());
21    }
22  }
23
24  class PiggyBank {
25    private int balance = 0;
26    public int getBalance() {
27      return balance;
28    }
29    public synchronized void deposit(int amount) {
31      int newBalance = balance + amount;
32      Thread.yield();
33      balance = newBalance;
34    }
35  }
```

2 Executions






```
24  class PiggyBank {
25    private int balance = 0;
26    private Lock lock = new ReentrantLock();
27    public int getBalance() {
28      return balance;
29    }
30    public void deposit(int amount) {
31      lock.lock();
32      try {
33        int newBalance = balance + amount;
34        Thread.yield();
35        balance = newBalance;
36      } finally {
37        lock.unlock();
38      }
39    }
40  }
```

AddAndRemoveAPenny Description

- Let's modify our `AddAPenny` program to also allow a thread to remove a penny
- Create a `withdraw(int)` method in the `PiggyBank` class
- Have half of the threads deposit a penny and half of the threads withdraw a penny

AddAndRemoveAPenny

```
1  import java.util.concurrent.ExecutorService;
2  import java.util.concurrent.Executors;
3  import java.util.concurrent.locks.Lock;
4  import java.util.concurrent.locks.ReentrantLock;
5
6  public class AddAndRemoveAPenny implements Runnable{
7    private static PiggyBank piggy = new PiggyBank();
8    private boolean isWithdrawal;
9    public void run() {
10     if (isWithdrawal) {
11       piggy.withdraw(1);
12     }
13     else {
14       piggy.deposit(1);
15     }
16   }
17   public static void main(String [] args) {
18     ExecutorService executor = Executors.newCachedThreadPool();
19     for (int i=0; i < 100; i++) {
20       AddAndRemoveAPenny penny = new AddAndRemoveAPenny();
21       if (i < 50) { // exactly 50 threads will deposit
22         penny.isWithdrawal = false;
23       }
24       else { // and exactly 50 threads will withdraw
25         penny.isWithdrawal = true;
26       }
27       executor.execute(penny);
28     }
29     executor.shutdown();
30     while(!executor.isTerminated()) {
31       Thread.yield();
32     }
33     System.out.println("balance = " + piggy.getBalance());
34   }
35 }
```

```
36 class PiggyBank {
37   private int balance = 0;
38   private Lock lock = new ReentrantLock();
39   public int getBalance() {
40     return balance;
41   }
42   public void withdraw(int amount) {
43     lock.lock();
44     try {
45       while (balance < amount) {
46         System.out.print("Waiting for deposit...");
47         System.out.print(" to withdraw $" + amount);
48         System.out.println(" from balance of $" + balance);
49       }
50       balance -= amount;
51       System.out.print("$" + amount + " withdrawn,");
52       System.out.println(" leaving balance of $" + balance);
53     } finally {
54       lock.unlock();
55     }
56   }
57   public void deposit(int amount) {
58     lock.lock();
59     try {
60       System.out.print("Depositing $" + amount + ", ");
61       int newBalance = balance + amount;
62       Thread.yield();
63       balance = newBalance;
64       System.out.println("giving balance of $" + balance);
65     } finally {
65       lock.unlock();
66     }
67   }
68 }
```

AddAndRemoveAPenny Explanation

- In the previous example, sometimes everything works properly

```
$1 withdrawn, leaving balance of $9
$1 withdrawn, leaving balance of $8
$1 withdrawn, leaving balance of $7
$1 withdrawn, leaving balance of $6
$1 withdrawn, leaving balance of $5
$1 withdrawn, leaving balance of $4
$1 withdrawn, leaving balance of $3
$1 withdrawn, leaving balance of $2
$1 withdrawn, leaving balance of $1
$1 withdrawn, leaving balance of $0
balance = 0
```

- And other times we end up in an endless loop because we are stuck in the `withdraw` method's `while` loop, holding onto the lock, which does not allow any deposits

```
$1 withdrawn, leaving balance of $5
$1 withdrawn, leaving balance of $4
$1 withdrawn, leaving balance of $3
$1 withdrawn, leaving balance of $2
$1 withdrawn, leaving balance of $1
$1 withdrawn, leaving balance of $0
Waiting for deposit... to withdraw $1 from balance of $0
Waiting for deposit... to withdraw $1 from balance of $0
Waiting for deposit... to withdraw $1 from balance of $0
Waiting for deposit... to withdraw $1 from balance of $0
Waiting for deposit... to withdraw $1 from balance of $0
```

Thread Conditions

- Threads are able to communicate with each other based on specific conditions
- Threads can perform the following actions using the `java.util.concurrent.Condition` interface
  - `await()`
    - Wait for the condition to be signaled
  - `signal()`
    - Wake up one thread waiting on the condition
  - `signalAll()`
    - Wake up all of the threads waiting on the condition

- A `Condition` is created from a `Lock` object by calling the `newCondition()` method

AddAndRemoveAPenny with Conditions

```
1  import java.util.concurrent.ExecutorService;
2  import java.util.concurrent.Executors;
3  import java.util.concurrent.locks.Condition;
3  import java.util.concurrent.locks.Lock;
4  import java.util.concurrent.locks.ReentrantLock;
5
6  public class AddAndRemoveAPenny implements Runnable{
7    private static PiggyBank piggy = new PiggyBank();
8    private boolean isWithdrawal;
9    public void run() {
10     if (isWithdrawal) {
11       piggy.withdraw(1);
12     }
13     else {
14       piggy.deposit(1);
15     }
16   }
17   public static void main(String [] args) {
18     ExecutorService executor = Executors.newCachedThreadPool();
19     for (int i=0; i < 100; i++) {
20       AddAndRemoveAPenny penny = new AddAndRemoveAPenny();
21       if (i < 50) { // exactly 50 threads will deposit
22         penny.isWithdrawal = false;
23       }
24       else { // and exactly 50 threads will withdraw
25         penny.isWithdrawal = true;
26       }
27       executor.execute(penny);
28     }
29     executor.shutdown();
30     while(!executor.isTerminated()) {
31       Thread.yield();
32     }
33     System.out.println("balance = " + piggy.getBalance());
34   }
35 }
```

```
36 class PiggyBank {
37   private int balance = 0;
38   private Lock lock = new ReentrantLock();
40   private Condition depositMade = lock.newCondition();
39   public int getBalance() {
40     return balance;
41   }
42   public void withdraw(int amount) {
43     lock.lock();
44     try {
45       while (balance < amount) {
46         System.out.print("Waiting for deposit...");
47         System.out.print(" to withdraw $" + amount);
48         System.out.println(" from balance of $" + balance);
51         depositMade.await();
49       }
50       balance -= amount;
51       System.out.print("$" + amount + " withdrawn,");
52       System.out.println(" leaving balance of $" + balance);
56     } catch (InterruptedException ie) {
57       System.out.println("IE: " + ie.getMessage());
58     } finally {
54       lock.unlock();
55     }
56   }
57   public void deposit(int amount) {
58     lock.lock();
59     try {
60       System.out.print("Depositing $" + amount + ", ");
61       int newBalance = balance + amount;
62       Thread.yield();
63       balance = newBalance;
64       System.out.println("giving balance of $" + balance);
65       depositMade.signalAll();
65     } finally {
65       lock.unlock();
66     }
67   }
68 }
```

Thread Condition Issues

- Once a thread invokes `await()` on a condition, the thread will move to the waiting state until it is signaled
  - If `signal()` or `signalAll()` is never called on the condition, the thread will wait forever
- You must first obtain the lock on the object before you are able to invoke a method on its `Condition`
  - When a thread calls `await()`, it will release its lock
    - It will not be able to start executing again, even after it is signaled, if it is not able to obtain the lock again