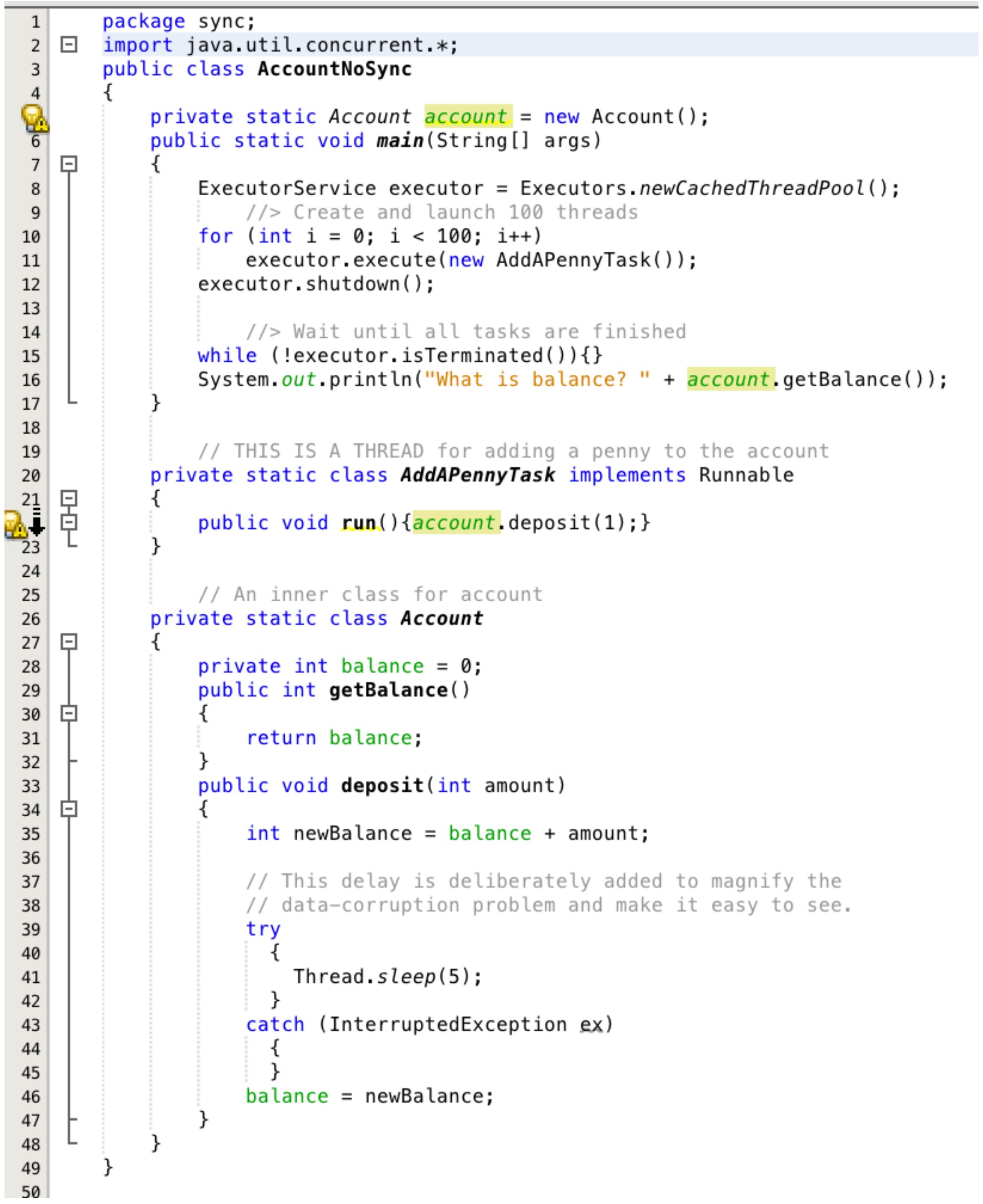
ASDV 2520, Data Structures and Algorithms
Lab, Threads II

1. Add a new package sync new project threads.

2. We have a PROBLEM when threads access common resources. In the example below, a common account, is accessed by multiple threads and the balance of the account loses integrity. Create class AccountNoSync and 2 static inner classes AddPennyTask and Account. Add the static variable account as shown in line 5 and create 10o threads in a pool that access the common static variable account. All threads add 1 penny to the account. So in the end the balance of the account should be 100 since we have 100 threads(users), but we don't.
Then,



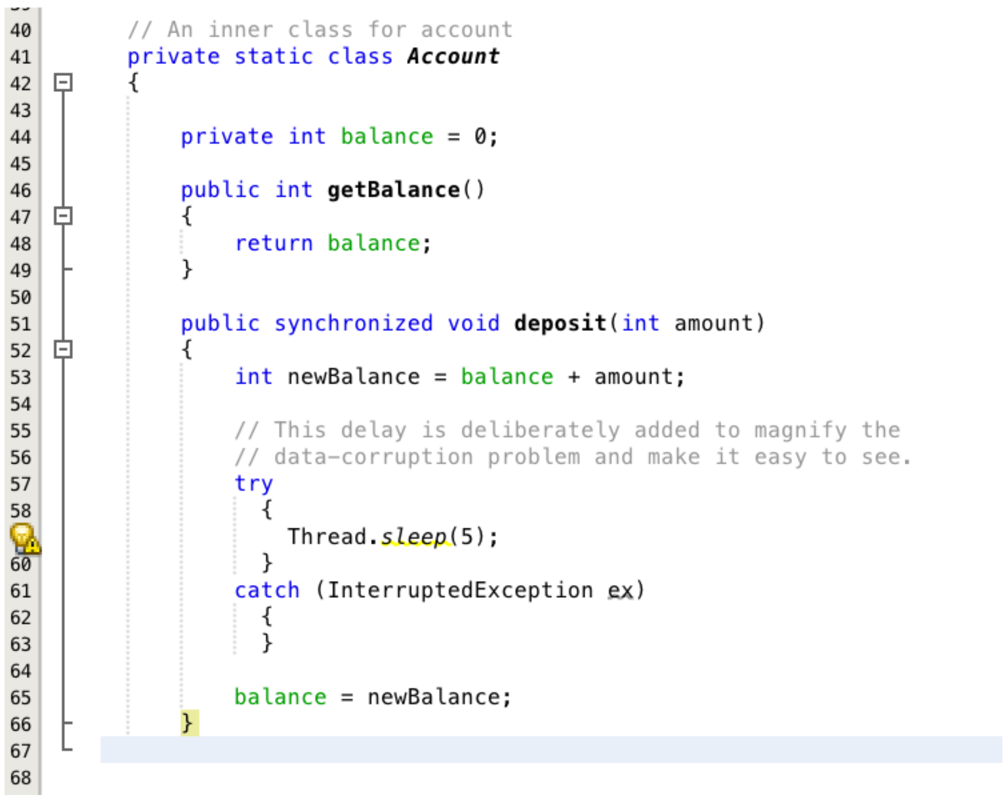
```
 1  package sync;
 2  import java.util.concurrent.*;
 3  public class AccountNoSync
 4  {
 5      private static Account account = new Account();
 6      public static void main(String[] args)
 7      {
 8          ExecutorService executor = Executors.newCachedThreadPool();
 9              //> Create and launch 100 threads
10          for (int i = 0; i < 100; i++)
11              executor.execute(new AddAPennyTask());
12          executor.shutdown();
13
14              //> Wait until all tasks are finished
15          while (!executor.isTerminated()){}
16          System.out.println("What is balance? " + account.getBalance());
17      }
18
19          // THIS IS A THREAD for adding a penny to the account
20      private static class AddAPennyTask implements Runnable
21      {
22          public void run(){account.deposit(1);}
23      }
24
25          // An inner class for account
26      private static class Account
27      {
28          private int balance = 0;
29          public int getBalance()
30          {
31              return balance;
32          }
33          public void deposit(int amount)
34          {
35              int newBalance = balance + amount;
36
37              // This delay is deliberately added to magnify the
38              // data-corruption problem and make it easy to see.
39              try
40                {
41                  Thread.sleep(5);
42                }
43              catch (InterruptedException ex)
44                {
45                }
46              balance = newBalance;
47          }
48      }
49  }
50
```

The Race Condition

Step	balance	Task 1	Task 2
1	0	newBalance = balance + 1;	
2	0		newBalance = balance + 1;
3	1	balance = newBalance;	
4	1		balance = newBalance;

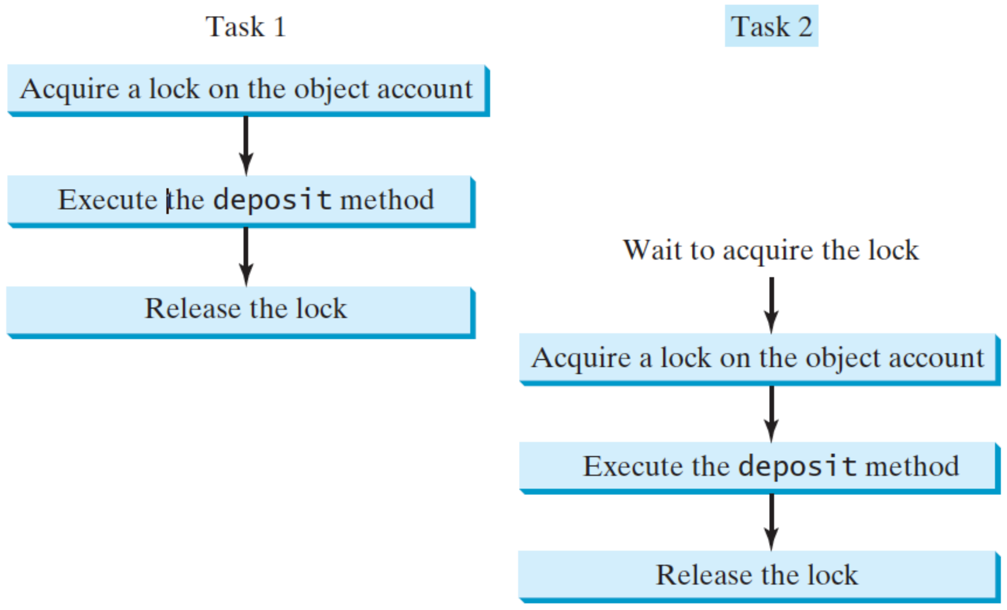
The problem is that Task 1 and Task 2 are accessing the common resource <u>balance</u> in a way that causes conflict. Task 1 did nothing, because in Step 4 **Task 2 overrides Task 1's** result, or **Task 2 UNDOES Task 1's** result and puts its own result. This is a common problem known as a race condition in multithreaded programs. A class is said to be thread-safe if an object of the class **does not cause a race condition** in the presence of multiple threads.

3. Fix the problem by adding synchronization. Copy <u>AcccountNoSync</u> and paste refactor into <u>AcccountSync1.</u> Add the keyword synchronize to method <u>deposit</u> in *line 51* as shown below. We avoid race conditions, and prevent more than one thread from simultaneously entering a certain part of the program, known as the **critical region**. The critical region is the entire <u>deposit</u> method. Run the program. The balance should be 100.

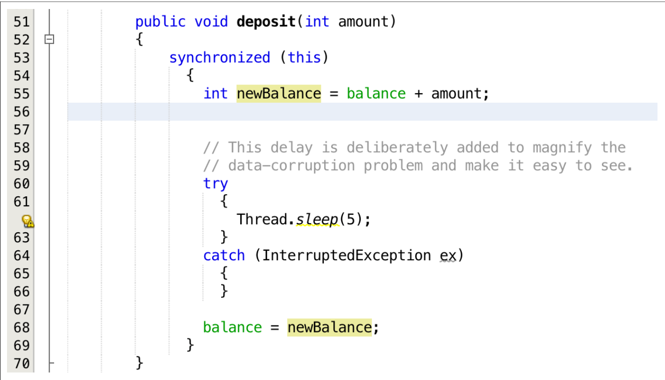


```
40      // An inner class for account
41      private static class Account
42      {
43
44          private int balance = 0;
45
46          public int getBalance()
47          {
48              return balance;
49          }
50
51          public synchronized void deposit(int amount)
52          {
53              int newBalance = balance + amount;
54
55              // This delay is deliberately added to magnify the
56              // data-corruption problem and make it easy to see.
57              try
58                {
59                  Thread.sleep(5);
60                }
61              catch (InterruptedException ex)
62                {
63                }
64
65              balance = newBalance;
66          }
67
68
```

When Task 1 enters the method deposit, Task 2 is **blocked** until Task 1 finishes the method's code.

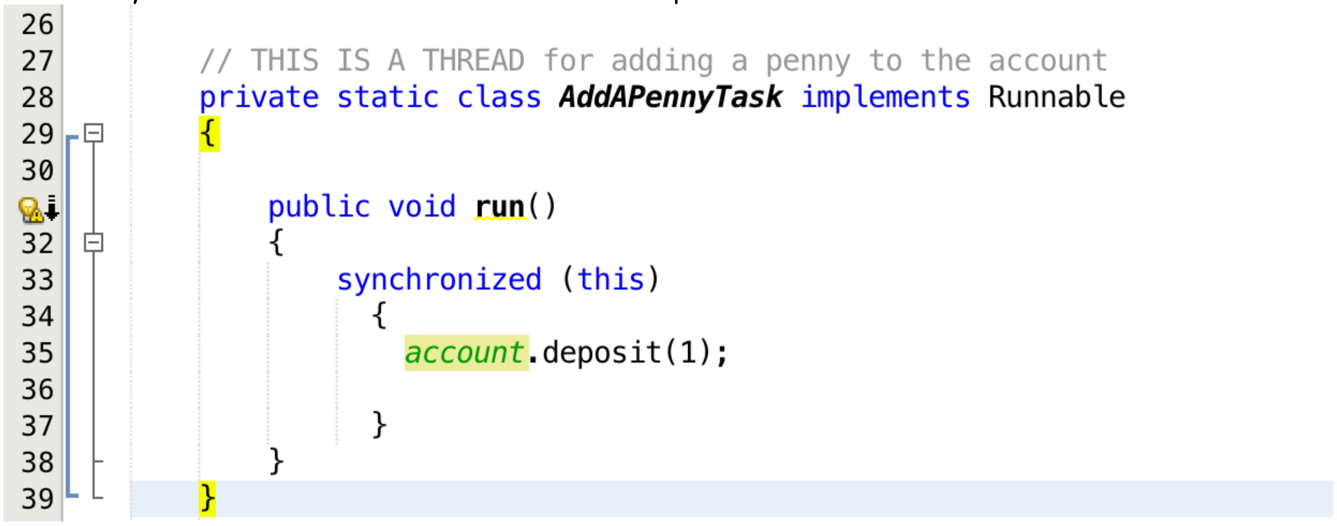

Task 1
Acquire a lock on the object account
Execute the deposit method
Release the lock
Task 2
Wait to acquire the lock
Acquire a lock on the object account
Execute the deposit method
Release the lock

4. Fix the problem again differently. Copy AcccountNoSync and paste refactor into AcccountSync2.

	Synchronized block: **synchronized (expr)**
	{
	statements;
	} The expression **expr** must evaluate to an object **reference**. If the object is already locked by another thread, the thread is blocked until the lock is released. When a lock is obtained on the object, the statements in the synchronized block are executed and then the lock is released.

```
        public void deposit(int amount)
        {
            synchronized (this)
            {
                int newBalance = balance + amount;


                // This delay is deliberately added to magnify the
                // data-corruption problem and make it easy to see.
                try
                {
                    Thread.sleep(5);
                }
                catch (InterruptedException ex)
                {
                }

                balance = newBalance;
            }
        }
```

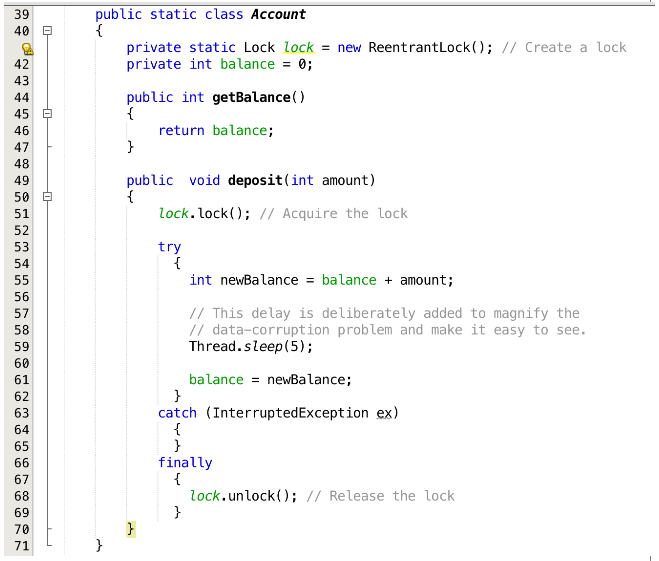
5. Try the code below instead of the code of step 4. Observe. Does it work? Who is “this” here?



```

    // THIS IS A THREAD for adding a penny to the account
    private static class AddAPennyTask implements Runnable
    {

        public void run()
        {
            synchronized (this)
              {
                account.deposit(1);

              }
        }
    }
```

EXPLICIT LOCKS

6. Copy AcccountNoSync and paste refactor into AcccountSync3. Modify the inner static class account by introducing a Lock in line 41. , lock in line 51 and unlock in line 68.
7.



```
39  public static class Account
40  {
        private static Lock lock = new ReentrantLock(); // Create a lock
42      private int balance = 0;
43
44      public int getBalance()
45      {
46          return balance;
47      }
48
49      public  void deposit(int amount)
50      {
51          lock.lock(); // Acquire the lock
52
53          try
54            {
55              int newBalance = balance + amount;
56
57              // This delay is deliberately added to magnify the
58              // data-corruption problem and make it easy to see.
59              Thread.sleep(5);
60
61              balance = newBalance;
62            }
63          catch (InterruptedException ex)
64            {
65            }
66          finally
67            {
68              lock.unlock(); // Release the lock
69            }
70      }
71  }
```

Instead of using implicit locking via the synchronized keyword use explicit locks specified by the Lock interface.

ALWAYS your code into a try/finally block to ensure unlocking in case of exceptions.

Communication Between Threads

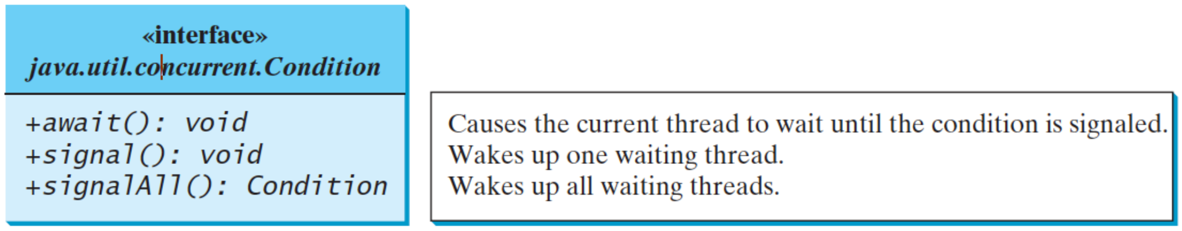
a) Threads can cooperate.

b) Conditions on locks can be used to coordinate thread interactions.

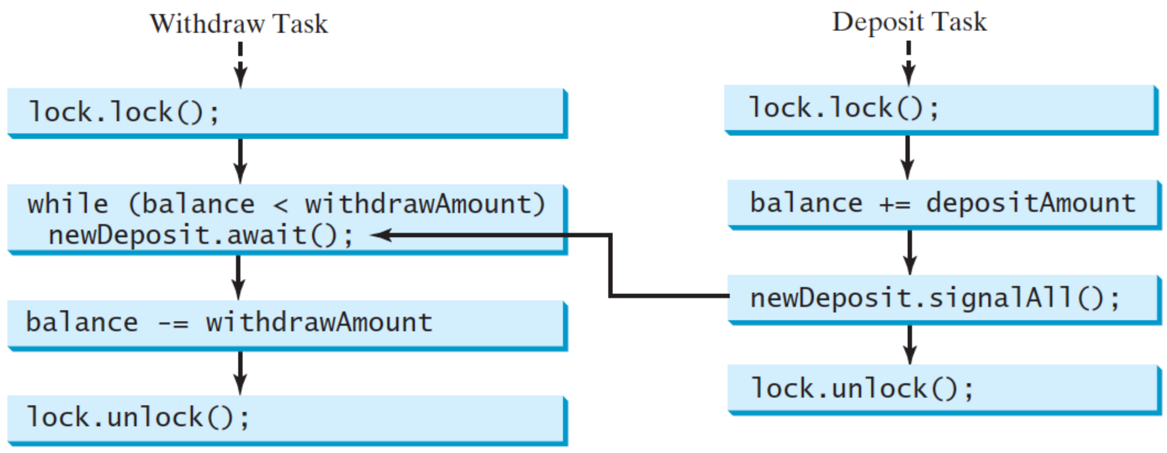
c) Thread synchronization avoids race conditions and ensures the mutual exclusion of multiple threads in the critical region.

d) Threads can cooperate.

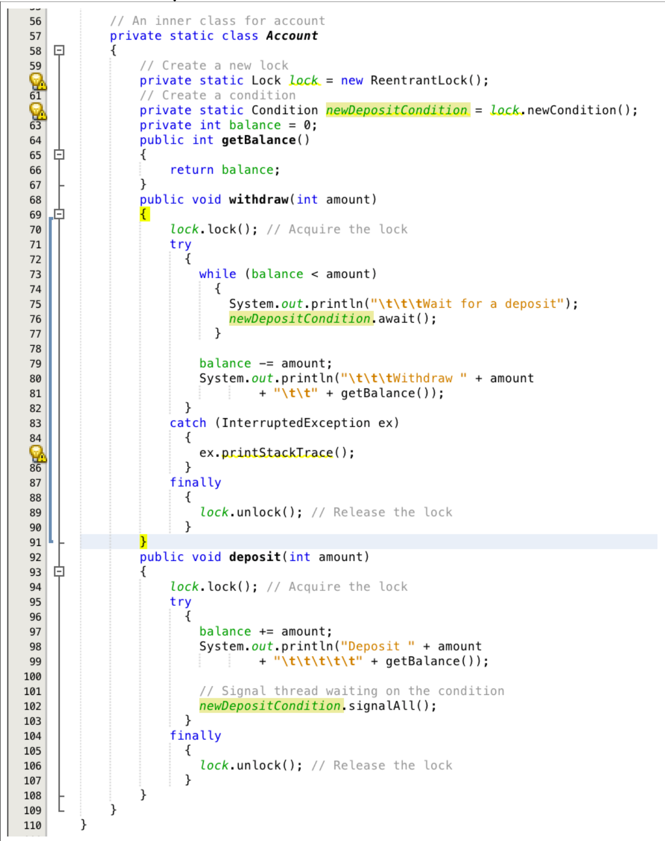
1. Conditions can be used for communications among threads. A thread can specify what to do under a certain condition.
2. Conditions are objects created by invoking the newCondition() method on a Lock object. Once a condition is created, you can use its await(), signal(), and signalAll() methods for thread communications.


«interface»
java.util.concurrent.Condition
+await(): void
+signal(): void
+signalAll(): Condition
Causes the current thread to wait until the condition is signaled.
Wakes up one waiting thread.
Wakes up all waiting threads.

The communication between two Threads where one deposits money in account while the other withdraws money from account.



7. We will implement the communication between threads using conditions on deposting and withdrawing money from account. Copy <u>AcccountSync3</u> and paste refactor into <u>AcccountSync4.</u> Modify the inner class account as shown below:



```
 56      // An inner class for account
 57      private static class Account
 58      {
 59          // Create a new lock
             private static Lock lock = new ReentrantLock();
 61          // Create a condition
             private static Condition newDepositCondition = lock.newCondition();
 63          private int balance = 0;
 64          public int getBalance()
 65          {
 66              return balance;
 67          }
 68          public void withdraw(int amount)
 69          {
 70              lock.lock(); // Acquire the lock
 71              try
 72              {
 73                  while (balance < amount)
 74                  {
 75                      System.out.println("\t\t\tWait for a deposit");
 76                      newDepositCondition.await();
 77                  }
 78
 79                  balance -= amount;
 80                  System.out.println("\t\t\tWithdraw " + amount
 81                          + "\t\t" + getBalance());
 82              }
 83              catch (InterruptedException ex)
 84              {
                     ex.printStackTrace();
 86              }
 87              finally
 88              {
 89                  lock.unlock(); // Release the lock
 90              }
 91          }
 92          public void deposit(int amount)
 93          {
 94              lock.lock(); // Acquire the lock
 95              try
 96              {
 97                  balance += amount;
 98                  System.out.println("Deposit " + amount
 99                          + "\t\t\t\t\t" + getBalance());
100
101                  // Signal thread waiting on the condition
102                  newDepositCondition.signalAll();
103              }
104              finally
105              {
106                  lock.unlock(); // Release the lock
107              }
108          }
109      }
110  }
```

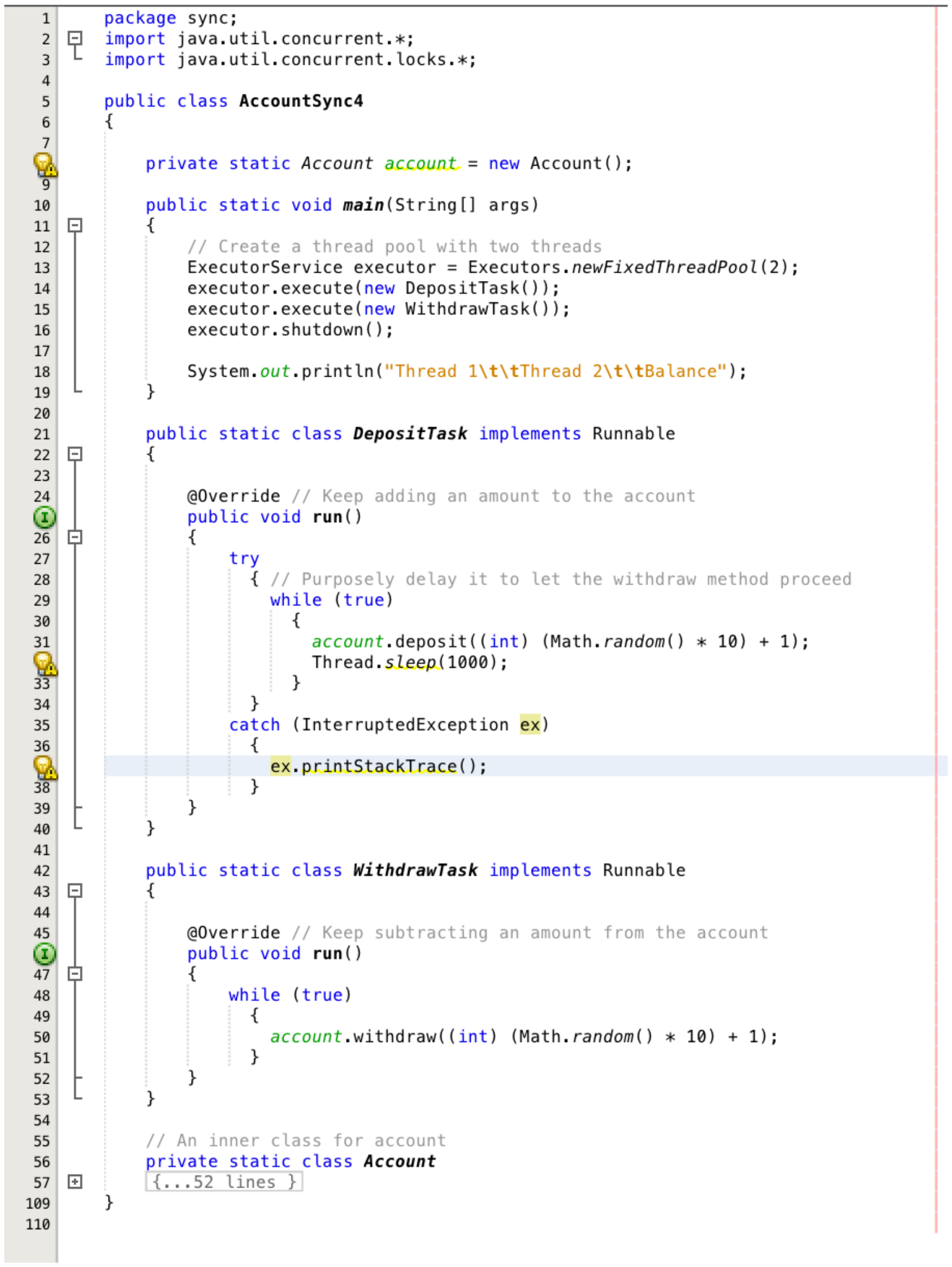
Line 62. Create a condition object.

Line 76, Withdraw money `await()`
Causes the current thread to wait until it is signaled or interrupted.

Line 102, Deposit money `signalAll()`
Any threads waiting on this condition are all woken up. Each thread must re-acquire the lock before it can return from `await`.

9. Modify AccountSync4 with inner classes <u>DepositTask</u> and <u>WithrawTask</u> as shown and in main having only 2 threads, one that deposits and the other that withdraws.

10. What will happen if we replace the while loop in lines 73 with an if statement? Replace it to see the affect on the balance.



```
  1   package sync;
  2   import java.util.concurrent.*;
  3   import java.util.concurrent.locks.*;
  4
  5   public class AccountSync4
  6   {
  7
      private static Account account = new Account();
  9
 10       public static void main(String[] args)
 11       {
 12           // Create a thread pool with two threads
 13           ExecutorService executor = Executors.newFixedThreadPool(2);
 14           executor.execute(new DepositTask());
 15           executor.execute(new WithdrawTask());
 16           executor.shutdown();
 17
 18           System.out.println("Thread 1\t\tThread 2\t\tBalance");
 19       }
 20
 21       public static class DepositTask implements Runnable
 22       {
 23
 24           @Override // Keep adding an amount to the account
              public void run()
 26           {
 27               try
 28               { // Purposely delay it to let the withdraw method proceed
 29                 while (true)
 30                 {
 31                   account.deposit((int) (Math.random() * 10) + 1);
                      Thread.sleep(1000);
 33                 }
 34               }
 35               catch (InterruptedException ex)
 36               {
                    ex.printStackTrace();
 38               }
 39           }
 40       }
 41
 42       public static class WithdrawTask implements Runnable
 43       {
 44
 45           @Override // Keep subtracting an amount from the account
              public void run()
 47           {
 48               while (true)
 49               {
 50                 account.withdraw((int) (Math.random() * 10) + 1);
 51               }
 52           }
 53       }
 54
 55       // An inner class for account
 56       private static class Account
 57       {...52 lines }
109   }
110
```

Caution 1

Once a thread invokes await() on a condition, the thread waits for a signal to resume. If you forget to call signal() or signalAll() on the condition, **the thread will wait forever.**

Caution 2

A condition is created from a Lock object. To invoke its methods , await(), signal(), and signalAll() **you must first own the lock.** If you invoke these methods without acquiring the lock, **an IllegalMonitorStateException will be thrown.**

11. Synchronization and Monitors in Java

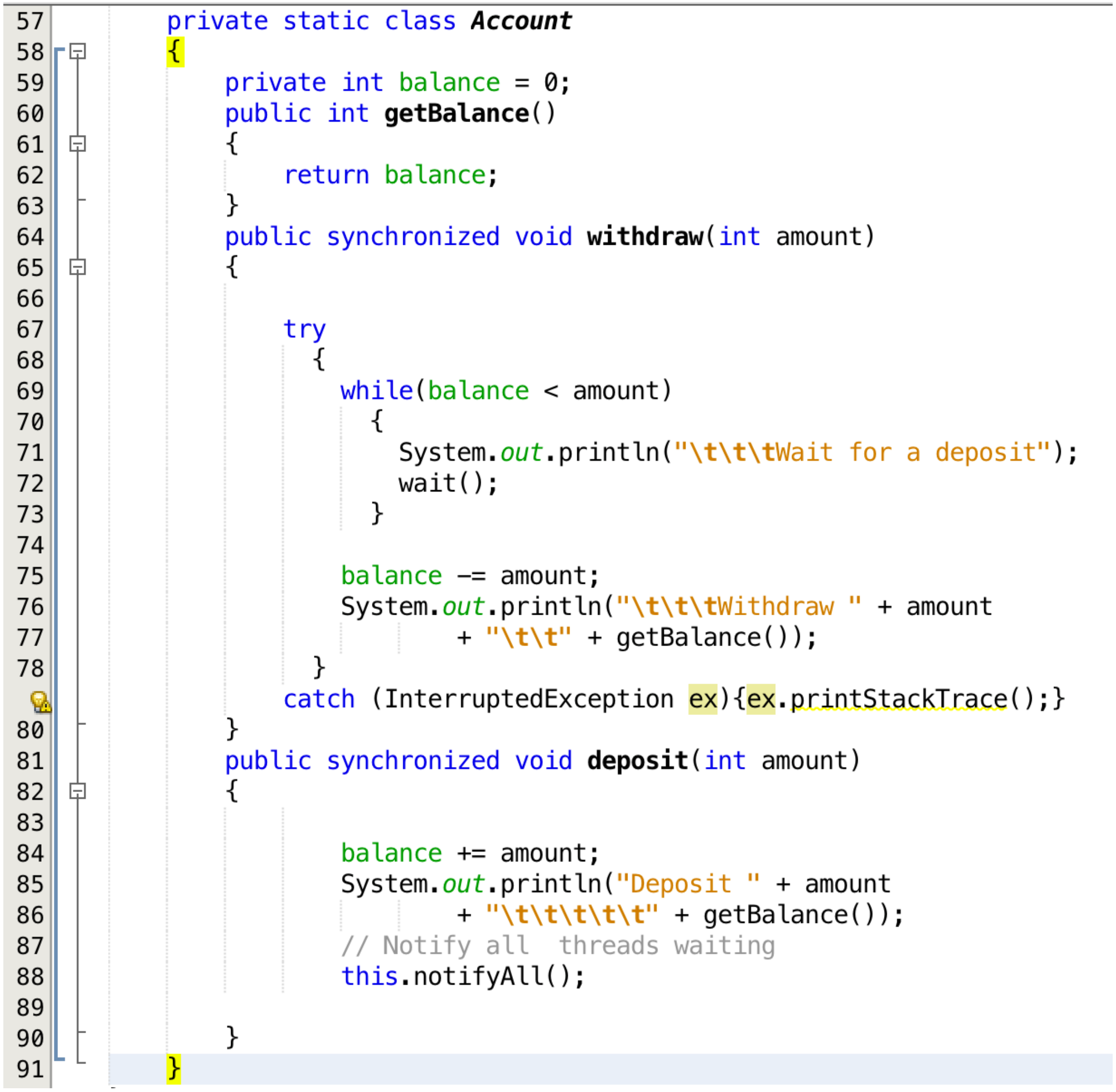
Synchronization in java is implemented using monitors. Each object in Java is associated with a **monitor, which a thread can lock or unlock. An object itself becomes a monitor once a thread locks it.** Only one thread at a time may hold a lock on a monitor. Any other threads attempting to lock that monitor are blocked until they can obtain a lock on that monitor. A thread may lock a particular monitor multiple times; each unlock reverses the effect of one lock operation.

The synchronized statement computes a reference to an object; it then attempts to perform a lock action on that object's monitor and does not proceed further until the lock action has successfully completed. After the lock action has been performed, the body of the synchronized statement is executed. If execution of the body is ever completed, either normally or abruptly, an unlock action is automatically performed on that same monitor.

A synchronized method automatically performs a lock action when it is invoked; its body is not executed until the lock action has successfully completed. **If the method is an instance method, it locks the monitor associated with the instance** for which it was invoked (that is, the object that will be known as *this* during execution of the body of the method). **If the method is static, it locks the monitor** associated with the Class object that represents the class in which the method is defined. If execution of the method's body is ever completed, either normally or abruptly, an unlock action is automatically performed on that same monitor.

12. Copy and paste/refactor AccountSync4 into AccountSync5.

13. Modify it by deleting Locks, adding synchronized and using Object's wait and notifyAll to achieve the same affect we achieved with locks.



```
    private static class Account
    {
        private int balance = 0;
        public int getBalance()
        {
            return balance;
        }
        public synchronized void withdraw(int amount)
        {

            try
             {
               while(balance < amount)
                {
                  System.out.println("\t\t\tWait for a deposit");
                  wait();
                }

               balance -= amount;
               System.out.println("\t\t\tWithdraw " + amount
                        + "\t\t" + getBalance());
             }
            catch (InterruptedException ex){ex.printStackTrace();}
        }
        public synchronized void deposit(int amount)
        {

               balance += amount;
               System.out.println("Deposit " + amount
                        + "\t\t\t\t\t" + getBalance());
               // Notify all  threads waiting
               this.notifyAll();

        }
    }
```