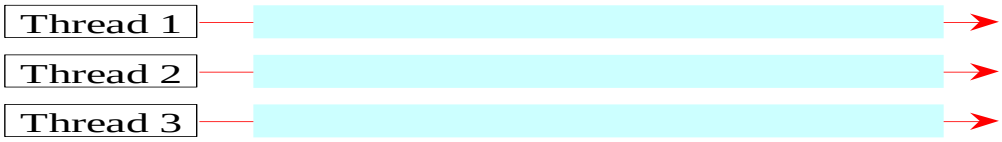
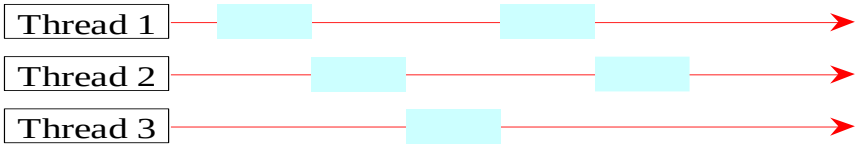
Summary on Java Multithreading and Synchronization Techniques

1. Thread Concept

- A program may consist of many tasks that can run concurrently. A thread is the flow of execution of a task.
- In Java, you can launch multiple threads from a program concurrently
- When your program executes an application, the JRE starts a thread for the main method. When your program executes an applet, the Web browser (or Applet container) starts a thread to run the applet.
- You can create additional threads to run concurrent tasks in the program.
- In Java, each task is an instance of the Runnable interface, also called runnable object. A thread is essentially an object that facilitates the execution of a task

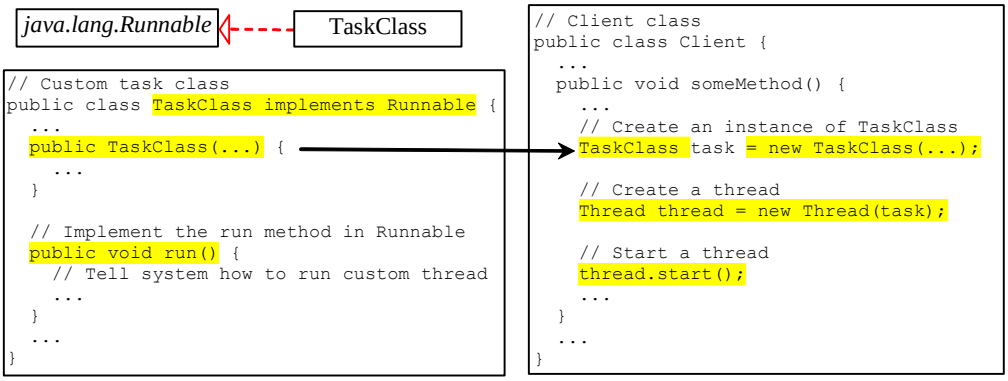


Multiple threads on multiple CPUs



Multiple threads on a single CPU

2. Creating Tasks and Threads



Example:

Create and run three threads:

- The first thread prints the letter *a* 100 times.
- The second thread prints the letter *b* 100 times.
- The third thread prints the integers 1 through 100.

printChar.java

```
// The task for printing a specified character in specified times
public class PrintChar implements Runnable {
  private char charToPrint; // The character to print
  private int times; // The times to repeat

  /** Construct a task with specified character and number of
   *  times to print the character
   */
  public PrintChar(char c, int t) {
    charToPrint = c;
    times = t;
  }

  /** Override the run() method to tell the system
   *  what the task to perform
   */
  public void run() {
    for (int i = 0; i < times; i++) {
      System.out.print(charToPrint);
    }
  }
}
```

printNum.java

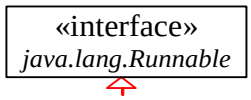
```
//The task class for printing number from 1 to n for a given n
class PrintNum implements Runnable {
  private int lastNum;

  /** Construct a task for printing 1, 2, ... i */
  public PrintNum(int n) {
    lastNum = n;
  }

  /** Tell the thread how to run */
  public void run() {
    for (int i = 1; i <= lastNum; i++) {
      System.out.print(" " + i);
    }
  }
}
```

TaskThreadDemo.java

```
public class TaskThreadDemo {
        public static void main(String[] args) {
          // Create tasks
```

```
    Runnable printA = new PrintChar('a', 100);
    Runnable printB = new PrintChar('b', 100);
    Runnable print100 = new PrintNum(100);

    // Create threads
    Thread thread1 = new Thread(printA);
    Thread thread2 = new Thread(printB);
    Thread thread3 = new Thread(print100);

    // Start threads
    thread1.start();
    thread2.start();
    thread3.start();
  }
}
```

3. The Thread Class



java.lang.Thread	
+Thread()	Creates a default thread.
+Thread(task: Runnable)	Creates a thread for a specified task.
+start(): void	Starts the thread that causes the run() method to be invoked by the JVM.
+isAlive(): boolean	Tests whether the thread is currently running.
+setPriority(p: int): void	Sets priority p (ranging from 1 to 10) for this thread.
+join(): void	Waits for this thread to finish.
+sleep(millis: long): void	Puts the runnable object to sleep for a specified time in milliseconds.
+yield(): void	Causes this thread to temporarily pause and allow other threads to execute.
+interrupt(): void	Interrupts this thread.

- The Static yield() Method

  You can use the yield() method to temporarily release time for other threads. For example, suppose you modify the code in PrintNum.java as follows:

```
public void run() {
  for (int i = 1; i <= lastNum; i++) {
    System.out.print(" " + i);
     Thread.yield();
  }
}
```

  Every time a number is printed, the print100 thread is yielded. So, the numbers are printed after the characters.

- The Static sleep(milliseconds) Method

  The sleep(long mills) method puts the thread to sleep for the specified time in milliseconds. For example, suppose you modify the code in PrintNum.java as follows:

```
public void run() {
```

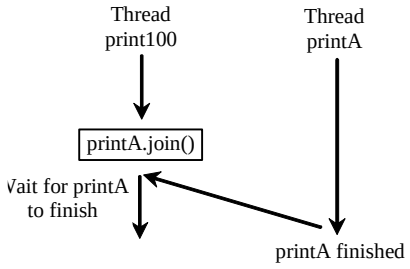
```
  for (int i = 1; i <= lastNum; i++) {
    System.out.print(" " + i);
    try {
      if (i >= 50) Thread.sleep(1);
    }
    catch (InterruptedException ex) {
    }
  }
}
```

Every time a number (>= 50) is printed, the print100 thread is put to sleep for 1 millisecond.

- The join() Method

  You can use the join() method to force one thread to wait for another thread to finish. For example, suppose you modify the code in PrintNum.java as follows:

```
public void run() {
  Thread thread4 = new Thread(
    new PrintChar('c', 40));
  thread4.start();
  try {
    for (int i = 1; i <= lastNum; i++) {
      System.out.print(" " + i);
      if (i == 50) thread4.join();
    }
  }
  catch (InterruptedException ex) {
  }
```



  The numbers after 50 are printed after thread printA is finished.

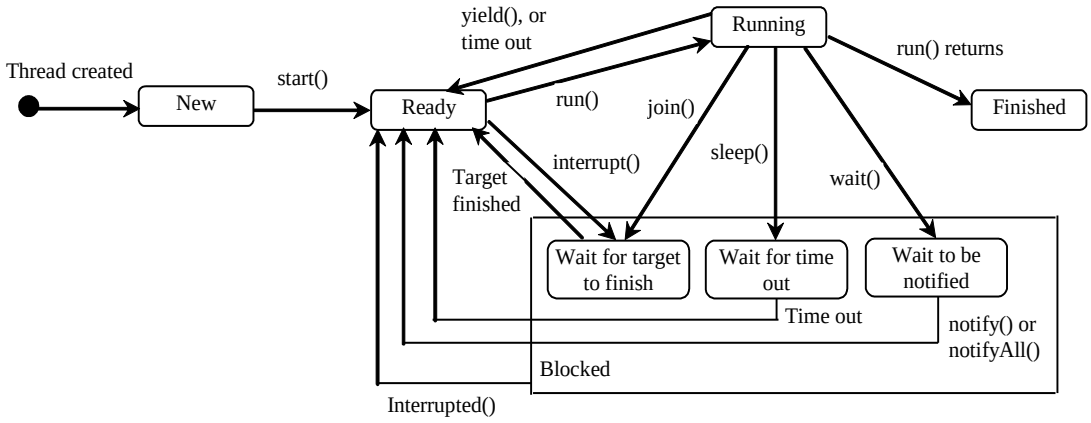
- isAlive(), interrupt(), and isInterrupted()

  The isAlive() method is used to find out the state of a thread. It returns true if a thread is in the Ready, Blocked, or Running state; it returns false if a thread is new and has not started or if it is finished.
  The interrupt() method interrupts a thread in the following way: If a thread is currently in the Ready or Running state, its interrupted flag is set; if a thread is currently blocked, it is awakened and enters the Ready state, and an java.io.InterruptedException is thrown.
  The isInterrupt() method tests whether the thread is interrupted.

- The deprecated stop(), suspend(), and resume() Methods

  NOTE: The Thread class also contains the stop(), suspend(), and resume() methods. As of Java 2, these methods are *deprecated* (or *outdated*) because they are known to be inherently unsafe. You should assign null to a Thread variable to indicate that it is stopped rather than use the stop() method.

4. The Life Cycle of a Thread



- Thread Priority

  Each thread is assigned a default priority of Thread.NORM_PRIORITY. You can reset the priority using setPriority(int priority).
  Some constants for priorities include Thread.MIN_PRIORITY Thread.MAX_PRIORITY Thread.NORM_PRIORITY

Example: Applet/Application animation

FlashingText.java

```
import javax.swing.*;

public class FlashingText extends JApplet implements Runnable {
  public static final long serialVersionUID = 1L;
  private JLabel jlblText = new JLabel("Welcome", JLabel.CENTER);

  public FlashingText() {
    add(jlblText);
    new Thread(this).start();
  }

  /** Set the text on/off every 200 milliseconds */
  public void run() {
    try {
      while (true) {
        if (jlblText.getText() == null)
          jlblText.setText("Welcome");
        else
          jlblText.setText(null);

        Thread.sleep(200);
      }
    }
```

```
    catch (InterruptedException ex) {
    }
  }

  /** Main method */
  public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        JFrame frame = new JFrame("FlashingText");
        frame.add(new FlashingText());
        frame.setLocationRelativeTo(null); // Center the frame
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(200, 200);
        frame.setVisible(true);
      }
    });
  }
}
```

5. GUI Event Dispatcher Thread

GUI event handling and painting code executes in a single thread, called the *event dispatcher thread*. This is necessary because most of Swing methods are not thread-safe. Invoking them from multiple threads may cause conflicts. In certain situations with multithreading, you need to run the code in the event dispatch thread to avoid possible conflicts.

- invokeLater and invokeAndWait

  You can use the static methods, invokeLater and invokeAndWait, in the javax.swing.SwingUtilities class to run the code in the event dispatcher thread. You must put this code in the run method of a Runnable object and specify the Runnable object as the argument to invokeLater and invokeAndWait. The invokeLater method returns immediately, without waiting for the event dispatcher thread to execute the code. The invokeAndWait method is just like invokeLater, except that invokeAndWait doesn't return until the event-dispatching thread has executed the specified code.

- Launch Application from Main Method

  So far, you have launched your GUI application from the main method by creating a frame and making it visible. This works fine for most applications. In certain situations, however, it could cause problems. To avoid possible conflicts (e.g., thread deadlock), you should launch GUI creation from the event dispatcher thread as follows:

```
public static void main(String[] args) {
  SwingUtilities.invokeLater(new Runnable() {
    public void run() {
      // Place the code for creating a frame and setting it properties
    }
  });
}
```

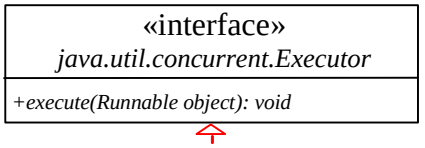
Example:

EventDispatcherThreadDemo.java

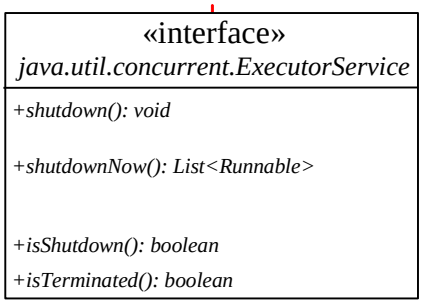
```
import javax.swing.*;

public class EventDispatcherThreadDemo extends JApplet {
public static final long serialVersionUID = 1L;
  public EventDispatcherThreadDemo() {
    add(new JLabel("Hi, it runs from an event dispatcher thread"));
  }

  /** Main method */
  public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        JFrame frame = new JFrame("EventDispatcherThreadDemo");
        frame.add(new EventDispatcherThreadDemo());
        frame.setLocationRelativeTo(null); // Center the frame
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(200, 200);
        frame.setVisible(true);
      }
    });
  }
}
```

6. Creating and Executing Threads with Executor Framework

Starting a new thread for each task could limit throughput and cause poor performance. A thread pool is ideal to manage the number of tasks executing concurrently. Since JDK 1.5 uses the Executor interface for executing tasks in a thread pool and the ExecutorService interface for managing and controlling tasks. ExecutorService is a subinterface of Executor.



Executes the runnable task.



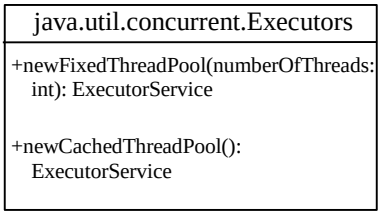
Shuts down the executor, but allows the tasks in the executor to complete. Once shutdown, it cannot accept new tasks.

Shuts down the executor immediately even though there are unfinished threads in the pool. Returns a list of unfinished tasks.

Returns true if the executor has been shutdown.

Returns true if all tasks in the pool are terminated.

- To create an Executor object, use the static methods in the Executors class.



Creates a thread pool with a fixed number of threads executing concurrently. A thread may be reused to execute another task after its current task is finished.

Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available.

Example:

ExecutorDemo.java

```
import java.util.concurrent.*;

public class ExecutorDemo {
  public static void main(String[] args) {
    // Create a fixed thread pool with maximum three threads
    ExecutorService executor = Executors.newFixedThreadPool(3);

    // Submit runnable tasks to the executor
    executor.execute(new PrintChar('a', 100));
    executor.execute(new PrintChar('b', 100));
    executor.execute(new PrintNum(100));

    // Shut down the executor
    executor.shutdown();
  }
}
```

Example:

PrintTask.java

```
import java.util.Random;

public class PrintTask implements Runnable
{
  private final int sleepTime; // random sleep time for thread
  private final String taskName; // name of task
  private final static Random generator = new Random();

  // constructor
  public PrintTask( String name )
  {
    taskName = name; // set task name

    // pick random sleep time between 0 and 5 seconds
    sleepTime = generator.nextInt( 5000 ); // milliseconds
  } // end PrintTask constructor

  // method run contains the code that a thread will execute
  public void run()
  {
    try // put thread to sleep for sleepTime amount of time
    {
```

```
      System.out.printf( "%s going to sleep for %d milliseconds.\n",
        taskName, sleepTime );
      Thread.sleep( sleepTime ); // put thread to sleep
    } // end try
    catch ( InterruptedException exception )
    {
      System.out.printf( "%s %s\n", taskName,
        "terminated prematurely due to interruption" );
    } // end catch

    // print task name
    System.out.printf( "%s done sleeping\n", taskName );
  } // end method run
} // end class PrintTask
```

TaskExecutor.java

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

public class TaskExecutor
{
  public static void main( String[] args )
  {
    // create and name each runnable
    PrintTask task1 = new PrintTask( "task1" );
    PrintTask task2 = new PrintTask( "task2" );
    PrintTask task3 = new PrintTask( "task3" );

    System.out.println( "Starting Executor" );

    // create ExecutorService to manage threads
    ExecutorService threadExecutor = Executors.newCachedThreadPool();

    // start threads and place in runnable state
    threadExecutor.execute( task1 ); // start task1
    threadExecutor.execute( task2 ); // start task2
    threadExecutor.execute( task3 ); // start task3

    // shut down worker threads when their tasks complete
    threadExecutor.shutdown();

    System.out.println( "Tasks started, main ends.\n" );
  } // end main
} // end class TaskExecutor
```

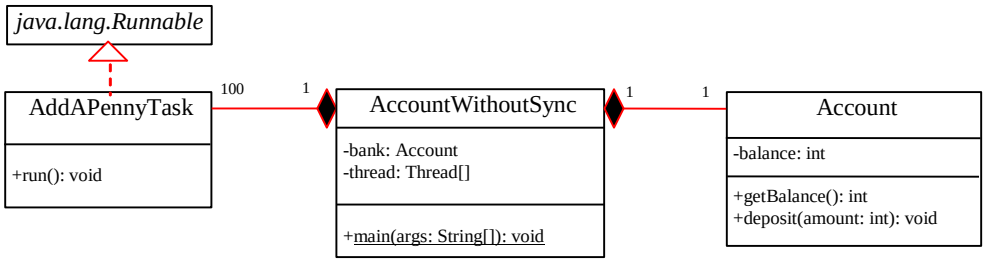
7. Thread Synchronization

- A shared resource may be corrupted if it is accessed simultaneously by multiple threads. For example, two unsynchronized threads accessing the same bank account may cause conflict.

Step	balance	thread[i]	thread[j]
1	0	newBalance = bank.getBalance() + 1;	
2	0		newBalance = bank.getBalance() + 1;
3	1	bank.setBalance(newBalance);	
4	1		bank.setBalance(newBalance);

Example:

Suppose that you create and launch one hundred threads, each of which adds a penny to an account. Assume that the account is initially empty.



AccountWithoutSync.java

```
import java.util.concurrent.*;

public class AccountWithoutSync {
  private static Account account = new Account();

  public static void main(String[] args) {
    ExecutorService executor = Executors.newCachedThreadPool();
   // ExecutorService executor = Executors.newFixedThreadPool(20);

    // Create and launch 100 threads
    for (int i = 0; i < 100; i++) {
      executor.execute(new AddAPennyTask());
    }

    executor.shutdown();

    // Wait until all tasks are finished
    while (!executor.isTerminated()) {
    }

    System.out.println("What is balance? " + account.getBalance());
  }

  // A thread for adding a penny to the account
  private static class AddAPennyTask implements Runnable {
```

```
  public void run() {
        System.out.println(Thread.currentThread());
        account.deposit(1);
  }
 }

 // An inner class for account
 private static class Account {
  private int balance = 0;

  public int getBalance() {
   return balance;
  }

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

   // This delay is deliberately added to magnify the
   // data-corruption problem and make it easy to see.
   try {
        Thread.sleep(100);
   }
   catch (InterruptedException ex) {
   }
 //   System.out.println("this thread is in the deposit() " + Thread.currentThread().getId());
   balance = newBalance;
  }
 }
}
```

- Race Condition

  What caused the error in the example?
  Here is a possible scenario:

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 effect of this scenario is that Task 1 did nothing, because in Step 4 Task 2 overrides Task 1's result. Obviously, the problem is that Task 1 and Task 2 are accessing a common resource in a way that causes conflict. 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. As demonstrated in the preceding example, the Account class is not thread-safe.

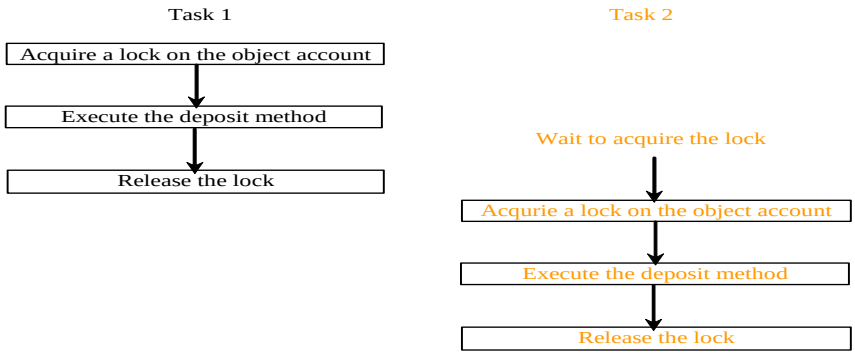
- The synchronized keyword

To avoid race conditions, more than one thread must be prevented from simultaneously entering certain part of the program, known as critical region. The critical region in the AccountWithoutSync.java is the entire deposit method. You can use the synchronized keyword to synchronize the method so that only one thread can access the method at a time. There are several ways to correct the problem in AccountWithoutSync.java. One approach is to make Account thread-safe by adding the synchronized keyword in the deposit method as follows:

public synchronized void deposit(double amount)

- Synchronizing Instance Methods and Static Methods
  A synchronized method acquires a lock before it executes. In the case of an instance method, the lock is on the object for which the method was invoked. In the case of a static method, the lock is on the class. If one thread invokes a synchronized instance method (respectively, static method) on an object, the lock of that object (respectively, class) is acquired first, then the method is executed, and finally the lock is released. Another thread invoking the same method of that object (respectively, class) is blocked until the lock is released.

  With the deposit method synchronized, the preceding scenario cannot happen. If Task 2 starts to enter the method, and Task 1 is already in the method, Task 2 is blocked until Task 1 finishes the method.



Example:

AccountWithSynch.java

```
import java.util.concurrent.*;

public class AccountWithSynch {
  private static Account account = new Account();

  public static void main(String[] args) {
    ExecutorService executor = Executors.newCachedThreadPool();
   // ExecutorService executor = Executors.newFixedThreadPool(20);

    // Create and launch 100 threads
    for (int i = 0; i < 100; i++) {
      executor.execute(new AddAPennyTask());
    }

    executor.shutdown();
```

```
    // Wait until all tasks are finished
    while (!executor.isTerminated()) {
    }

    System.out.println("What is balance? " + account.getBalance());
  }

  // A thread for adding a penny to the account
  private static class AddAPennyTask implements Runnable {
    public void run() {
      account.deposit(1);
      System.out.println(Thread.currentThread());
    }
  }

  // An inner class for account
  private static class Account {
    private int balance = 0;

    public int getBalance() {
      return balance;
    }

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

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

      balance = newBalance;
    }
  }
}
```

- Synchronizing Statements

  Invoking a synchronized instance method of an object acquires a lock on the object, and invoking a synchronized static method of a class acquires a lock on the class. A synchronized statement can be used to acquire a lock on any object, not just *this* object, when executing a block of the code in a method. This block is referred to as a *synchronized block*. The general form of a synchronized statement is as follows:

```
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.

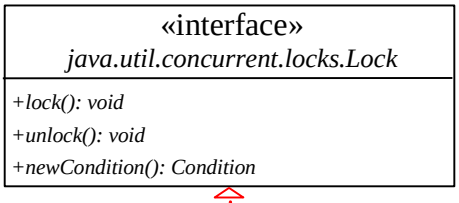
- Synchronizing Statements vs. Methods

  Any synchronized instance method can be converted into a synchronized statement. Suppose that the following is a synchronized instance method:

```
public synchronized void xMethod() {
  // method body
}

This method is equivalent to
public void xMethod() {
  synchronized (this) {
    // method body
  }
}
```

- Synchronization Using Locks

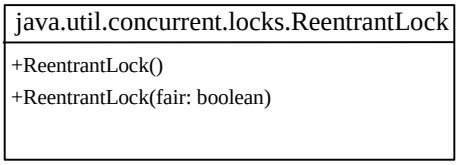
  A synchronized instance method implicitly acquires a lock on the instance before it executes the method. Since JDK 1.5 enables you to use locks explicitly. The new locking features are flexible and give you more control for coordinating threads. A lock is an instance of the Lock interface, which declares the methods for acquiring and releasing locks, as shown below. A lock may also use the newCondition() method to create any number of Condition objects, which can be used for thread communications.



Acquires the lock.

Releases the lock.

Returns a new Condition instance that is bound to this Lock instance.



Same as ReentrantLock(false).

Creates a lock with the given fairness policy. When the fairness is true, the longest-waiting thread will get the lock. Otherwise, there is no particular access order.

- Fairness Policy

  Reentrant Lock is a concrete implementation of Lock for creating mutual exclusive locks. You can create a lock with the specified fairness policy. True fairness policies guarantee the longest-wait thread to obtain the lock first. False fairness policies grant a lock to a waiting thread without any access order. Programs using fair locks accessed by many threads may have poor overall performance than those using the default setting, but have smaller variances in times to obtain locks and guarantee lack of starvation.

Example:

AccountWithSyncUsingLock.java

```
import java.util.concurrent.*;
import java.util.concurrent.locks.*;

public class AccountWithSyncUsingLock {
  private static Account account = new Account();

  public static void main(String[] args) {
    ExecutorService executor = Executors.newCachedThreadPool();

    // Create and launch 100 threads
    for (int i = 0; i < 100; i++) {
      executor.execute(new AddAPennyTask());
    }

    executor.shutdown();

    // Wait until all tasks are finished
    while (!executor.isTerminated()) {
    }

    System.out.println("What is balance ? " + account.getBalance());
  }

  // A thread for adding a penny to the account
  public static class AddAPennyTask implements Runnable {
    public void run() {
      account.deposit(1);
    }
  }

  // An inner class for account
  public static class Account {
    private static Lock lock = new ReentrantLock(); // Create a lock
    private int balance = 0;

    public int getBalance() {
      return balance;
    }

    public void deposit(int amount) {
      lock.lock(); // Acquire the lock

      try {
        int newBalance = balance + amount;

        // This delay is deliberately added to magnify the
        // data-corruption problem and make it easy to see.
        Thread.sleep(5);
```

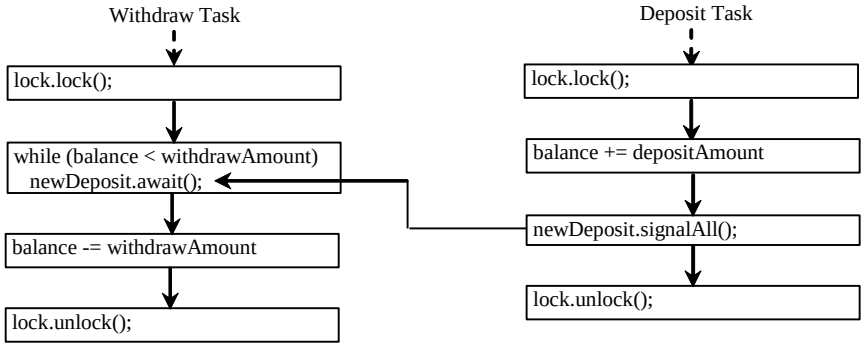
```
      balance = newBalance;
    }
    catch (InterruptedException ex) {
    }
    finally {
      lock.unlock(); // Release the lock
    }
  }
 }
}
```

8. Cooperation among Threads

The conditions can be used to facilitate communications among threads. A thread can specify what to do under a certain condition. 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, as shown below. The await() method causes the current thread to wait until the condition is signaled. The signal() method wakes up one waiting thread, and the signalAll() method wakes all waiting threads.

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

To synchronize the operations, use a lock with a condition: newDeposit (i.e., new deposit added to the account). If the balance is less than the amount to be withdrawn, the withdraw task will wait for the newDeposit condition. When the deposit task adds money to the account, the task signals the waiting withdraw task to try again. The interaction between the two tasks is shown below.



Example:

Suppose that you create and launch two threads, one deposits to an account, and the other withdraws from the same account. The second thread has to wait if the amount to be withdrawn is more than the current balance in the account. Whenever new fund is deposited to the account, the first thread notifies the second thread to resume. If the amount is

still not enough for a withdrawal, the second thread has to continue to wait for more funds in the account. Assume the initial balance is 0 and the amount to deposit and to withdraw is randomly generated.

ThreadCooperation.java

```
import java.util.concurrent.*;
import java.util.concurrent.locks.*;

public class ThreadCooperation {
  private static Account account = new Account();

  public static void main(String[] args) {
    // Create a thread pool with two threads
    ExecutorService executor = Executors.newFixedThreadPool(2);
    executor.execute(new DepositTask());
    executor.execute(new WithdrawTask());
    executor.shutdown();

    System.out.println("Thread 1\t\tThread 2\t\tBalance");
  }

  // A task for adding an amount to the account
  public static class DepositTask implements Runnable {
    public void run() {
      try { // Purposely delay it to let the withdraw method proceed
        while (true) {
          account.deposit((int)(Math.random() * 10) + 1);
          Thread.sleep(1000);
        }
      }
      catch (InterruptedException ex) {
        ex.printStackTrace();
      }
    }
  }

  // A task for subtracting an amount from the account
  public static class WithdrawTask implements Runnable {
    public void run() {
      while (true) {
        account.withdraw((int)(Math.random() * 10) + 1);
      }
    }
  }

  // An inner class for account
  private static class Account {
    // Create a new lock
    private static Lock lock = new ReentrantLock();
```

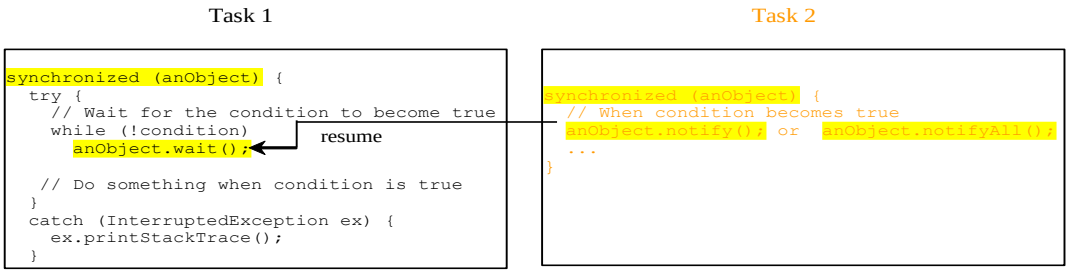
```
  // Create a condition
  private static Condition newDeposit = lock.newCondition();

  private int balance = 0;

  public int getBalance() {
    return balance;
  }

  public void withdraw(int amount) {
    lock.lock(); // Acquire the lock
    try {
      while (balance < amount)
        newDeposit.await();

      balance -= amount;
      System.out.println("\t\t\tWithdraw " + amount +
        "\t\t" + getBalance());
    }
    catch (InterruptedException ex) {
      ex.printStackTrace();
    }
    finally {
      lock.unlock(); // Release the lock
    }
  }

  public void deposit(int amount) {
    lock.lock(); // Acquire the lock
    try {
      balance += amount;
      System.out.println("Deposit " + amount +
        "\t\t\t\t\t" + getBalance());

      // Signal thread waiting on the condition
      newDeposit.signalAll();
    }
    finally {
      lock.unlock(); // Release the lock
    }
  }
 }
}
```

10. Java's Built-in Monitors

Locks and conditions are new in Java 5. Prior to Java 5, thread communications are programmed using object's built-in monitors. A *monitor* is an object with mutual exclusion and synchronization capabilities. Only one thread can execute a method at a time in the monitor. A thread enters the monitor by acquiring a lock on the monitor and exits by releasing the lock. *Any object can be a monitor*. An object becomes a monitor once a thread locks it. Locking is implemented using the synchronized keyword on a method or a block. A thread must acquire a lock

before executing a synchronized method or block. A thread can wait in a monitor if the condition is not right for it to continue executing in the monitor.

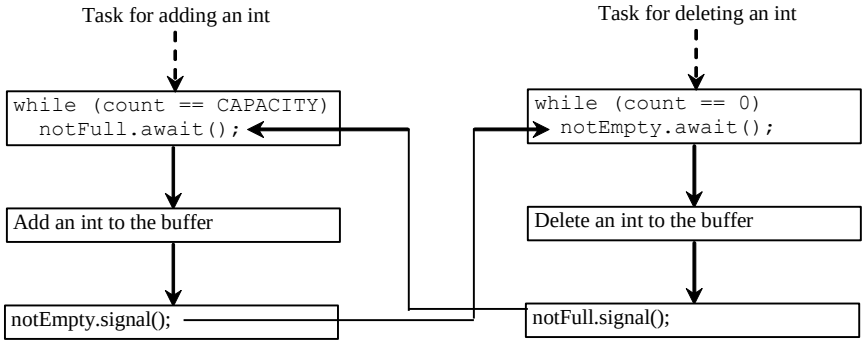
- wait(), notify(), and notifyAll()
  Use the wait(), notify(), and notifyAll() methods to facilitate communication among threads.
  The wait(), notify(), and notifyAll() methods must be called in a synchronized method or a synchronized block on the calling object of these methods. Otherwise, an IllegalMonitorStateException would occur.
  The wait() method lets the thread wait until some condition occurs. When it occurs, you can use the notify() or notifyAll() methods to notify the waiting threads to resume normal execution. The notifyAll() method wakes up all waiting threads, while notify() picks up only one thread from a waiting queue.



  The wait(), notify(), and notifyAll() methods on an object are analogous to the await(), signal(), and signalAll() methods on a condition.

Example:
Producer/Consumer

Consider the classic Consumer/Producer example. Suppose you use a buffer to store integers. The buffer size is limited. The buffer provides the method write(int) to add an int value to the buffer and the method read() to read and delete an int value from the buffer. To synchronize the operations, use a lock with two conditions: notEmpty (i.e., buffer is not empty) and notFull (i.e., buffer is not full). When a task adds an int to the buffer, if the buffer is full, the task will wait for the notFull condition. When a task deletes an int from the buffer, if the buffer is empty, the task will wait for the notEmpty condition. The interaction between the two tasks is shown below.



ProducerConsumer.java presents the complete program. The write(int) method in Buffer class adds an integer to the buffer. The read() method in Buffer class deletes and returns an integer from the buffer. The buffer is implemented using a linked list. Two conditions notEmpty and notFull on the lock are created in the Buffer class. The conditions are bound to a lock. A lock must be acquired before a condition can be applied.

Buffer.java

```
import java.util.concurrent.locks.*;
```

```
public class Buffer {
  private static final int CAPACITY = 5; // buffer size
  private java.util.LinkedList<Integer> queue =
    new java.util.LinkedList<Integer>();

  // Create a new lock
  private static Lock lock = new ReentrantLock();

  // Create two conditions
  private static Condition notEmpty = lock.newCondition();
  private static Condition notFull = lock.newCondition();

  public void write(int value) {
    lock.lock(); // Acquire the lock
    try {
      while (queue.size() == CAPACITY) {
        System.out.println("Wait for notFull condition");
        notFull.await();
      }

      queue.offer(value);
      notEmpty.signal(); // Signal notEmpty condition
    } catch (InterruptedException ex) {
      ex.printStackTrace();
    } finally {
      lock.unlock(); // Release the lock
    }
  }

  public int read() {
    int value = 0;
    lock.lock(); // Acquire the lock
    try {
      while (queue.isEmpty()) {
        System.out.println("\t\t\tWait for notEmpty condition");
        notEmpty.await();
      }

      value = queue.remove();
      notFull.signal(); // Signal notFull condition
    } catch (InterruptedException ex) {
      ex.printStackTrace();
    } finally {
      lock.unlock(); // Release the lock
      return value;
    }
  }
}
```

ProducerTask.java

```
// A task for adding an int to the buffer
 public class ProducerTask implements Runnable {

          private final Buffer buffer; // reference to shared object

           // constructor
           public ProducerTask( Buffer sharedBuffer )
           {
              buffer = sharedBuffer;
           } // end Producer constructor

           public void run() {
    try {
     int i = 1;
     while (true) {
       System.out.println("Producer writes " + i);
       buffer.write(i++); // Add a value to the buffer
       // Put the thread into sleep
       Thread.sleep((int)(Math.random() * 1000));
     }
    } catch (InterruptedException ex) {
     ex.printStackTrace();
    }
  }
 }
```

ConsumerTask.java

```
// A task for reading and deleting an int from the buffer
public class ConsumerTask implements Runnable {

          private final Buffer buffer; // reference to shared object

           // constructor
           public ConsumerTask( Buffer sharedBuffer )
           {
              buffer = sharedBuffer;
           } // end Producer constructor

 public void run() {
  try {
   while (true) {
     System.out.println("\t\t\tConsumer reads " + buffer.read());
     // Put the thread into sleep
     Thread.sleep((int)(Math.random() * 1000));
   }
  } catch (InterruptedException ex) {
```

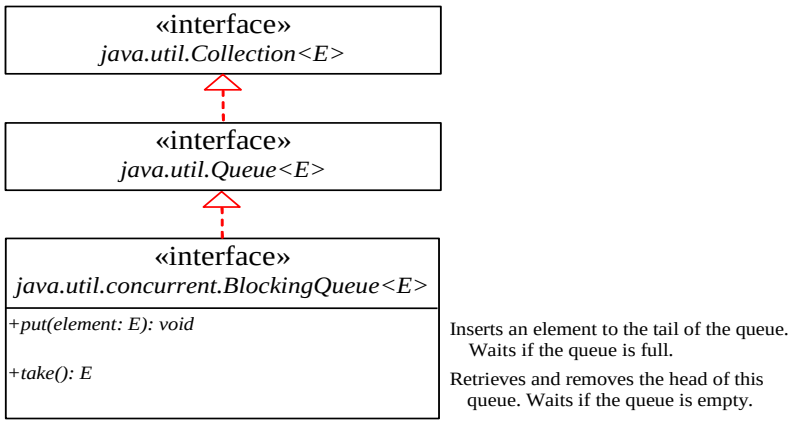
```
    ex.printStackTrace();
   }
  }
}
```

ProducerConsumer.java

```
import java.util.concurrent.*;

public class ProducerConsumer {
  private static Buffer buffer = new Buffer();

  public static void main(String[] args) {
    // Create a thread pool with two threads
    ExecutorService executor = Executors.newFixedThreadPool(2);
    executor.execute(new ProducerTask(buffer));
    executor.execute(new ConsumerTask(buffer));
    executor.shutdown();
  }
}
```

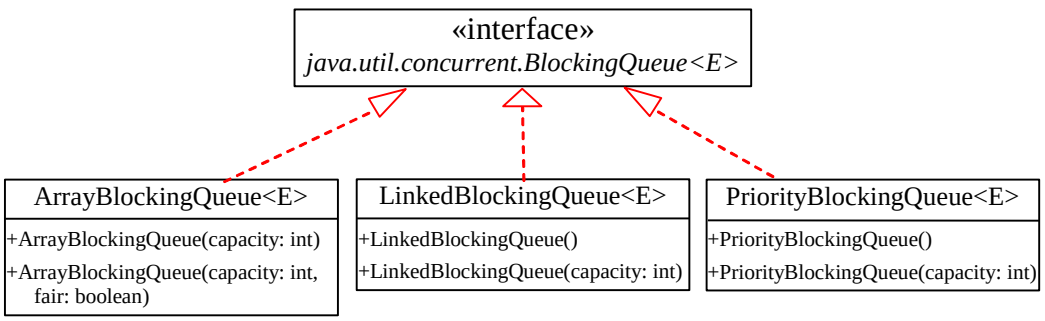
11. Blocking Queues

A *blocking queue* causes a thread to block when you try to add an element to a full queue or to remove an element from an empty queue.



- Concrete Blocking Queues

  Three concrete blocking queues ArrayBlockingQueue, LinkedBlockingQueue, and PriorityBlockingQueue are supported in Since JDK 1.5, as shown below. All are in the java.util.concurrent package. ArrayBlockingQueue implements a blocking queue using an array. You have to specify a capacity or an optional fairness to construct an ArrayBlockingQueue. LinkedBlockingQueue implements a blocking queue using a linked list. You may create an unbounded or bounded LinkedBlockingQueue. PriorityBlockingQueue is a priority queue. You may create an unbounded or bounded priority queue.



Example:

Producer.java

```
import java.util.concurrent.ArrayBlockingQueue;

// A task for adding an int to the buffer
public class Producer implements Runnable {

            private final ArrayBlockingQueue<Integer> buffer;

              // constructor
             public Producer( ArrayBlockingQueue<Integer> sharedBuffer )
             {
                buffer = sharedBuffer;
             } // end Producer constructor

            public void run() {
   try {
     int i = 1;
     while (true) {
       System.out.println("Producer writes " + i);
       buffer.put(i++); // Add any value to the buffer, say, 1
       // Put the thread into sleep
       Thread.sleep((int)(Math.random() * 1000));
     }
   } catch (InterruptedException ex) {
     ex.printStackTrace();
   }
 }
}
```

Consumer.java

```
import java.util.concurrent.ArrayBlockingQueue;

// A task for reading and deleting an int from the buffer
public class Consumer implements Runnable {

            private final ArrayBlockingQueue<Integer> buffer;

              // constructor
             public Consumer( ArrayBlockingQueue<Integer> sharedBuffer )
             {
                buffer = sharedBuffer;
             } // end Producer constructor

  public void run() {
    try {
```

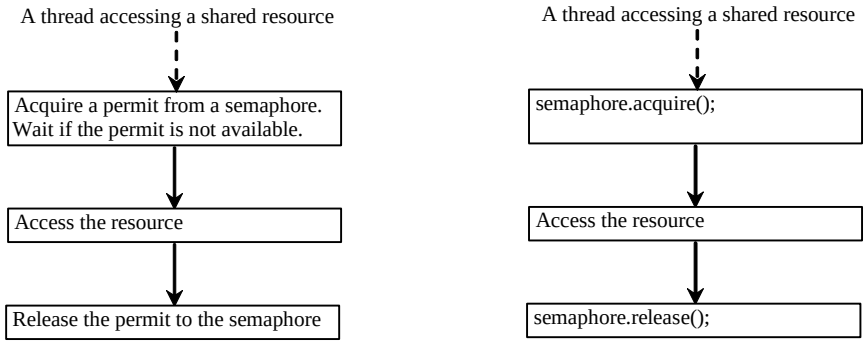
```
    while (true) {
      System.out.println("\t\t\tConsumer reads " + buffer.take());
      // Put the thread into sleep
      Thread.sleep((int)(Math.random() * 1000));
    }
   } catch (InterruptedException ex) {
    ex.printStackTrace();
   }
  }
}
```

ProducerConsumerUsingBQ.java

```
import java.util.concurrent.*;
public class ProducerConsumerUsingBQ {
        private static ArrayBlockingQueue<Integer> buffer =
          new ArrayBlockingQueue<Integer>(2);

        public static void main(String[] args) {
          // Create a thread pool with two threads
          ExecutorService executor = Executors.newFixedThreadPool(2);
          executor.execute(new Producer(buffer));
          executor.execute(new Consumer(buffer));
          executor.shutdown();
        }
}
```

12. Semaphore

Semaphores can be used to restrict the number of threads that access a shared resource. Before accessing the resource, a thread must acquire a permit from the semaphore. After finishing with the resource, the thread must return the permit back to the semaphore, as shown below.



To create a semaphore, you have to specify the number of permits with an optional fairness policy, as shown below. A task acquires a permit by invoking the semaphore's acquire() method and releases the permit by invoking the semaphore's release() method. Once a permit is acquired, the total number of available permits in a semaphore is reduced by 1. Once a permit is released, the total number of available permits in a semaphore is increased by 1.

java.util.concurrent.Semaphore	
+Semaphore(numberOfPermits: int)	Creates a semaphore with the specified number of permits. The fairness policy is false.
+Semaphore(numberOfPermits: int, fair: boolean)	Creates a semaphore with the specified number of permits and the fairness policy.
+acquire(): void	Acquires a permit from this semaphore. If no permit is available, the thread is blocked until one is available.
+release(): void	Releases a permit back to the semaphore.

Example:

AccountWithSemaphore.java

```
import java.util.concurrent.*;

public class AccountWithSemaphore {
  private static Account account = new Account();

  public static void main(String[] args) {
    ExecutorService executor = Executors.newCachedThreadPool();
   // ExecutorService executor = Executors.newFixedThreadPool(20);

    // Create and launch 100 threads
    for (int i = 0; i < 100; i++) {
      executor.execute(new AddAPennyTask());
    }

    executor.shutdown();

    // Wait until all tasks are finished
    while (!executor.isTerminated()) {
    }

    System.out.println("What is balance? " + account.getBalance());
  }

  // A thread for adding a penny to the account
  private static class AddAPennyTask implements Runnable {
    public void run() {
          System.out.println(Thread.currentThread());
          account.deposit(1);
    }
  }

  // An inner class for account
  private static class Account {
          private static Semaphore semaphore = new Semaphore(1);
    private int balance = 0;

    public int getBalance() {
      return balance;
    }

    public void deposit(int amount) {
     try {
      semaphore.acquire();
      int newBalance = balance + amount;
```

```
            Thread.sleep(100);
            balance = newBalance;
        }
        catch (InterruptedException ex) {
        }
        finally{
        semaphore.release();
      }
    }
  }
}
```