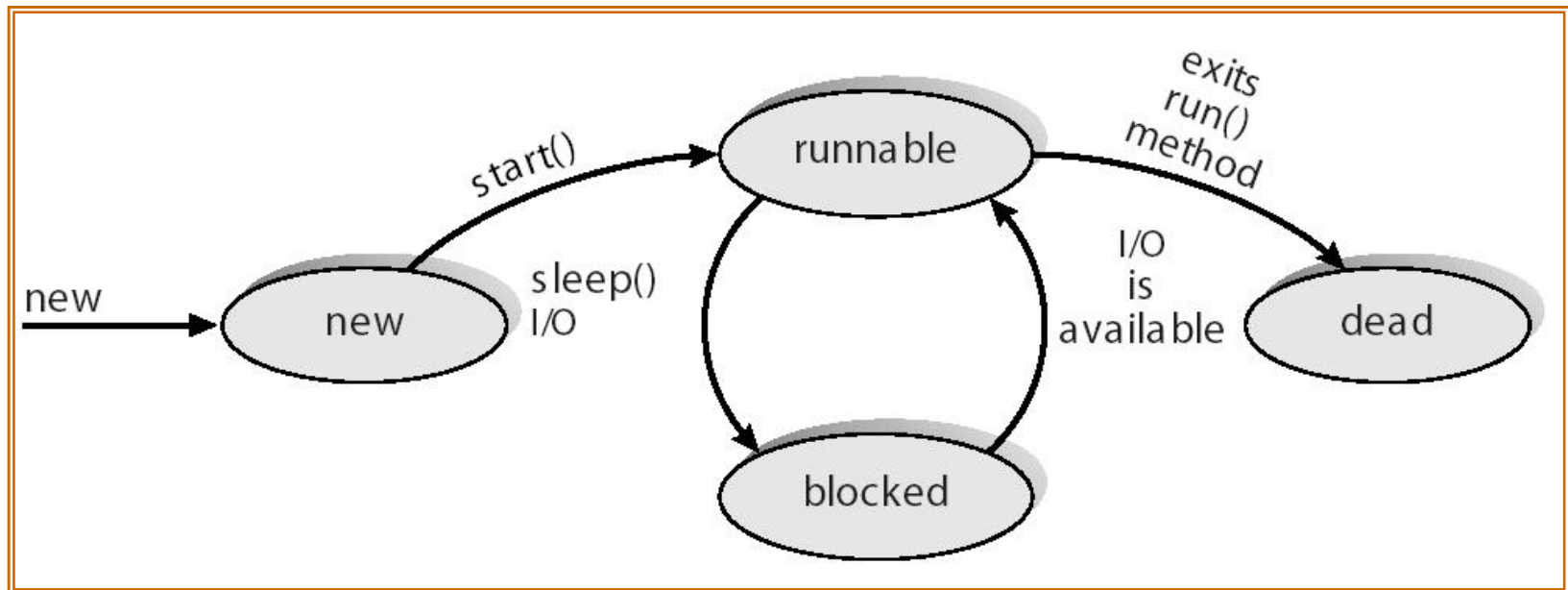


Java Threads

Java threads are managed by the JVM



Creating a Thread

- By extending the Thread class

```
// Custom thread class
public class CustomThread extends Thread
{
    ...
    public CustomThread(...)
    {
        ...
    }

    // Override the run method in Thread
    public void run()
    {
        // Tell system how to run custom thread
        ...
    }
    ...
}
```

```
// Client class
public class Client
{
    ...
    public someMethod()
    {
        ...
        // Create a thread
        CustomThread thread = new CustomThread(...);

        // Start a thread
        thread.start();
        ...
    }
    ...
}
```

Creating a Thread

- By implementing the Runnable interface

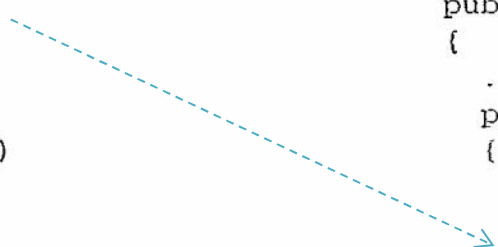
```
// Custom thread class
public class CustomThread
    implements Runnable
{
    ...
    public CustomThread(...)
    {
        ...
    }

    // Implement the run method in Runnable
    public void run()
    {
        // Tell system how to run custom thread
        ...
    }
    ...
}
```

```
// Client class
public class Client
{
    ...
    public someMethod()
    {
        ...
        // Create an instance of CustomThread
        CustomThread customThread
            = new CustomThread(...);

        // Create a thread
        Thread thread = new Thread(customThread);

        // Start a thread
        thread.start();
        ...
    }
    ...
}
```



Thread Sleep

- You can make a thread sleep (**block**) for a number of milliseconds
- Notice that these methods are static
- They throw an InterruptedException

```
public static void sleep(long ms) throws InterruptedException
```

```
...  
    try{  
        sleep(1000); // sleep 1 sec  
    } catch (InterruptedException e) {}  
...
```

Thread Join

- If you create several threads you can wait for the threads to complete before putting together the results from these threads

```
public final void join() throws InterruptedException
public final void join(long millis) throws InterruptedException
public final void join(long millis, int nanos) throws InterruptedException
```

```
Thread t1 = new Thread(new Task1());
Thread t2 = new Thread(new Task2());
Thread t3 = new Thread(new Task3());
```

```
// this will call run() function
t1.start(); t2.start(); t3.start();
/
/ waits for this thread to die
t1.join(); t2.join(); t3.join();
```

Interrupting Threads

- In Java, you have no way to force a Thread to stop
- But you can interrupt a Thread with the `interrupt()` method (software interrupt)
 - If the thread is sleeping or joining another Thread, an `InterruptedException` is thrown and the interrupted status of the thread is cleared

```
public void interrupt()
```

Intrinsic Locks – synchronized methods

- Any piece of code that can be simultaneously modified by several threads must be made **Thread Safe**
- Consider the following simple piece of code:

```
public int getNextCount() {  
    return ++counter;  
}
```

- An increment like this, is not an **atomic** action, it involves:
 - **Reading** the current value of counter
 - **Adding** one to its current value
 - **Storing** the result back to memory

Intrinsic Locks – synchronized methods

- We must use a lock on access to the “value”
- You can add such lock to a method by simply using the keyword: **synchronized**

```
public synchronized int getNextCount(){  
    return ++counter;  
}
```
- This guarantees that only one thread executes the method
- If you have several methods with the synchronized keyword, only one method can be executed at a time
 - This is called an **Intrinsic Lock**

Intrinsic Locks – synchronized block

Each Java object has an intrinsic lock associated with it (monitor)

- we can use that lock to synch access to a method
- instead, we can elect to synch access to a block of code

```
public int getNextValue() {  
    synchronized (this) {return value++;}  
}
```

- alternatively, use the lock of another object, which is useful since it allows you to use several locks for thread safety in a single class

```
public int getNextValue() {  
    synchronized (lock) {return value++;}  
}
```

Intrinsic Locks - synchronized static methods

- So we mentioned that each Java object has an intrinsic lock associated with it, e.g. static methods that are not associated with a particular object?
 - There is also an intrinsic lock associated with the class
 - Only used for synchronized class (static) methods

Java Monitors

- There are several mechanisms to create monitors
 - The simplest one, uses the knowledge that we have already gathered regarding the **intrinsic locks**
- The intrinsic locks can be effectively used for mutual exclusion (**competition synchronization**)
- We need a mechanism to implement **cooperation synchronization**
 - we need to allow threads to suspend themselves if a condition prevents their execution in a monitor
 - This is handled by the **wait()** and **notify()** methods

Wait Operations

`wait()`

Tells the calling thread to give up the monitor and wait until some other thread enters the same monitor and calls `notify()` or `notifyAll()`

`wait(long timeout)`

Causes the current thread to wait until another thread invokes the `notify()` or `notifyAll()` method, or the specified amount of time elapses

Notify Operations

notify()

Wakes up a single thread that is waiting on this object's monitor (intrinsic lock).

If more than a single thread is waiting, the choice is arbitrary (is this fair?)

The awakened thread will not be able to proceed until the current thread relinquishes the lock.

notifyAll()

Wakes up all threads that are waiting on this object's monitor.

The awakened thread will not be able to proceed until the current thread relinquishes the lock.

The next thread to lock this monitor is also randomly chosen

Wait / Notify Example

```
public class BufferMonitor{
    int [] buffer = new int [5];
    int next_in = 0, next_out = 0, filled = 0;

    public synchronized void deposit (int item )
        throws InterruptedException{
        while (buffer.length == filled){
            wait(); // blocks thread
        }

        buffer[next_in] = item;
        next_in = (next_in + 1) % buffer.length;
        filled++;

        notifyAll(); // free a task that has been waiting on a condition
    }
}
```

Wait / Notify Example

```
public synchronized int fetch() throws InterruptedException{
    while (filled == 0){
        wait(); // block thread
    }

    int item = buffer[next_out];
    next_out = (next_out + 1) % buffer.length;
    filled--;

    notifyAll(); // free a task that has been waiting on a condition

    return item;
}
```

Java Lock interface

- You can create a monitor using the Java Lock interface
 - **ReentrantLock** is the most popular implementation of **Lock**
- ReentrantLock defines two constructors:
 - Default constructor
 - Constructor that takes a Boolean (is_fair)
- Fairness is a slightly heavy (in terms of processing), and therefore should be used only when needed
- To acquire the lock, you just have to use the method **lock**, and to release it, call **unlock**

<https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantLock.html>

Lock Example

```
public class SimpleMonitor {  
    private final Lock lock = new ReentrantLock();  
    public void testA() {  
        lock.lock();  
        try { //Some code }  
        finally {lock.unlock();}  
    }  
    public int testB() {  
        lock.lock();  
        try {return 1;}  
        finally {lock.unlock();}  
    }  
}
```

Adding a condition to a lock

- Without being able to wait on a condition, monitors are useless...
 - Cooperation is not possible
- There is a specific class that has been developed just to this end: `Condition` class
 - You create a **`Condition`** instance using the **`newCondition()`** method defined in the **`Lock`** interface

Adding a condition to a lock

```
public class BufferMonitor {  
  
    int [] buffer = new int [5];  
    int next_in = 0, next_out = 0, filled = 0;  
  
    private final Lock lock = new ReentrantLock(true);  
        private final Condition notFull =  
lock.newCondition();  
        private final Condition notEmpty =  
lock.newCondition();  
}
```

Adding a condition to a lock

```
public void deposit (int item ) throws InterruptedException{
    lock.lock(); // Lock to ensure mutually exclusive access
    try{
        while (buffer.length == filled){
            notFull.await(); // blocks thread (wait on condition)
        }
        buffer[next_in] = item;
        next_in = (next_in + 1) % buffer.length;
        filled++;
        notEmpty.signal(); // signal thread waiting on the empty condition
    }
    finally{
        lock.unlock(); // Whenever you lock, you must unlock
    }
}
```

Lock vs Synchronized

- Monitors implemented with `Lock` and `Condition` classes have some advantages over the intrinsic lock based implementation:
 1. Ability to have more than one condition variable per monitor (see previous example)
 2. Ability to make the lock fair (remember, synchronized blocks or methods do not guarantee fairness)
 3. Ability to check if the lock is currently being held (by calling the `isLocked()` method)
 - ✓ Alternatively, you can call `tryLock()` which acquires the lock only if it is not held by another thread
 4. Ability to get the list of threads waiting on the lock (by calling the method `getQueuedThreads()`)

Lock vs Synchronized

- Disadvantages of `Lock` and `Condition` :
 1. Need to add lock acquisition and release code
 2. Need to add try-finally block

Java Semaphores

- Java defines a semaphore class:
`java.util.concurrent.Semaphore`
- Creating a counting semaphore:
`Semaphore available = new Semaphore(100);`
- Creating a binary semaphore:
`Semaphore available = new Semaphore(1);`

Semaphore Example

```
public class Example {  
    private int counter= 0;  
    private final Semaphore mutex = new Semaphore(1)  
    public int getNextCount() throws InterruptedException {  
        mutex.acquire();  
        try {  
            return ++counter;  
        } finally {  
            mutex.release();  
        }  
    }  
}
```