Module 12

Threads

Huang Hua

School of Computer & Information Technology
Beijing Jiaotong University
hhua@bjtu.edu.cn

Objectives

- Define a thread
- Create separate threads in a Java technology program, controlling the code and data that are used by that thread
- Control the execution of a thread and write platform-independent code with threads
- Describe the difficulties that might arise when multiple threads share data
- Use **`wait`** and **`notify`** to communicate between threads
- Use **`synchronized`** to protect data from corruption

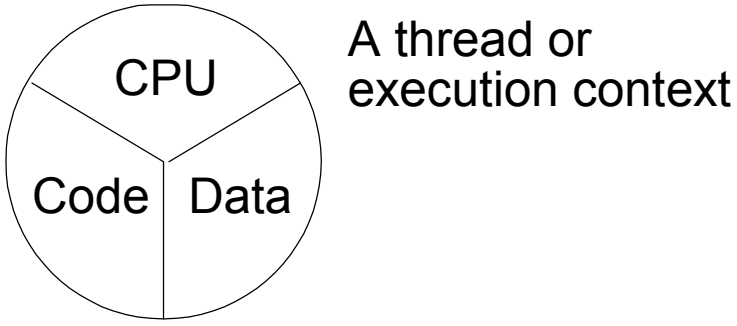
Relevance

How do you get programs to perform multiple tasks concurrently?

Threads

- What are threads?

  Threads are a virtual CPU.
- The three parts of at thread are:
  - CPU
  - Code
  - Data



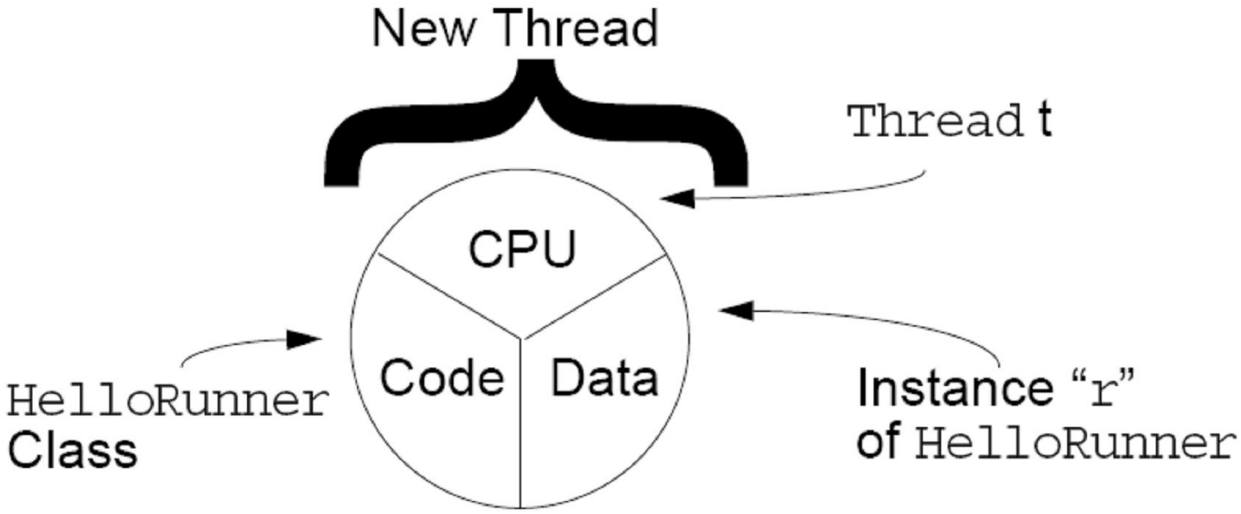
Creating the Thread

```
1  public class ThreadTester {
2    public static void main(String args[]) {
3      HelloRunner r = new HelloRunner();
4      Thread t = new Thread(r);
5      t.start();
6    }
7  }
8  class HelloRunner implements Runnable {
9    int i;
10   public void run() {
11     i = 0;
12     while (true) {
13       System.out.println("Hello " + i++);
14       if ( i == 50 ) {
15         break;
16       }
17     }
18   }
19 }
```

Creating the Thread

- Multithreaded programming has these characteristics:
  - Multiple threads are from one **`Runnable`** instance.
  - Threads share the same data and code.
- For example:

```
Thread t1 = new Thread(r);
Thread t2 = new Thread(r);
```

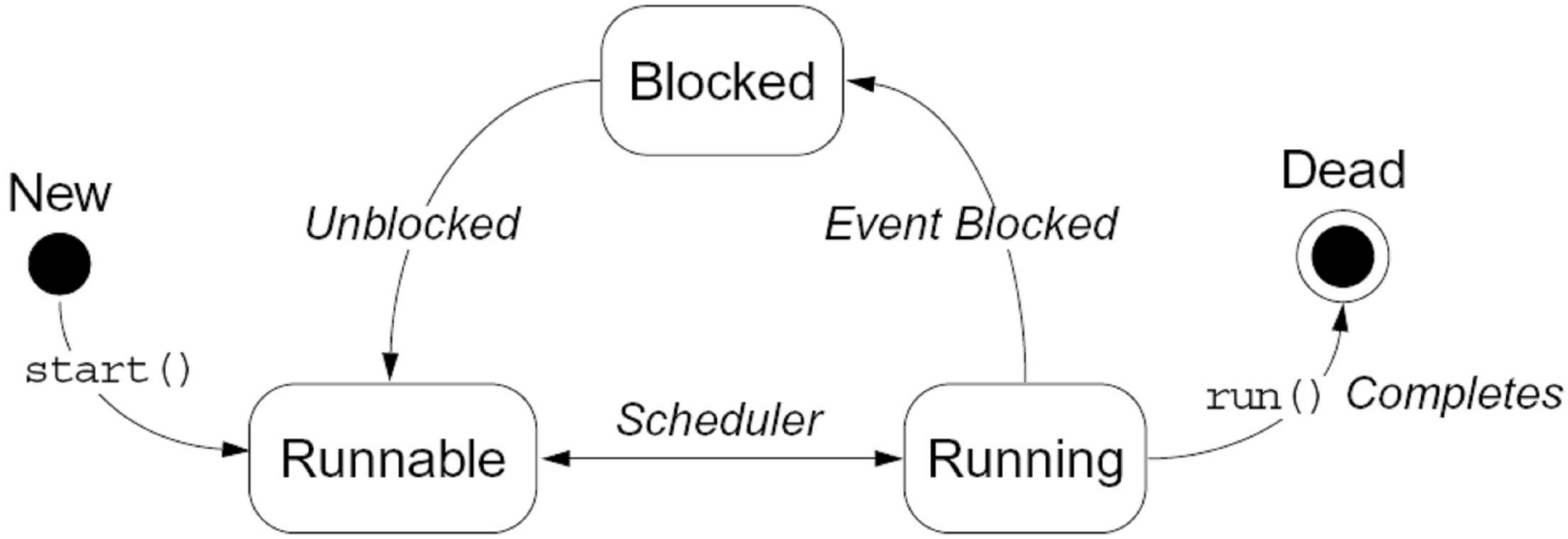


Starting the Thread

- Use the **`start`** method.
- Place the thread in a **runnable state**(**do not mean that the thread runs immediately**).

Thread Scheduling

- Preemptive Schedule



Thread Scheduling Example

```
1   public class Runner implements Runnable {
2     public void run() {
3       while (true) {
4         // do lots of interesting stuff
5         // ...
6         // Give other threads a chance
7         try {
8         Thread.sleep(10);
9       } catch (InterruptedException e) {
10        // This thread's sleep was interrupted
11        // by another thread
12      }
13    }
14  }
15 }
```

Terminating a Thread

```
1   public class Runner implements Runnable{
2     private boolean timeToQuit=false;
3
4     public void run() {
5       while ( ! timeToQuit ) {
6         // continue doing work
7       }
8       // clean up before run() ends
9     }
10
11    public void stopRunning() {
12      timeToQuit=true;
13    }
14  }
```

Terminating a Thread(Cont'd)

```
1  public class ThreadController {
2    private Runner r = new Runner();
3    private Thread t = new Thread(r);
4
5    public void startThread() {
6      t.start();
7    }
8
9    public void stopThread() {
10     // use specific instance of Runner
11     r.stopRunning();
12   }
13 }


// Thread.currentThread()
```

Basic Control of Threads

- Test threads:
  `isAlive()`
- Access thread priority:
  `getPriority()`
  `setPriority()`

  Contants:
  `Thread.MIN_PRIORITY = 1`
  `Thread.NORM_PRIORITY = 5`
  `Thread.MAX_PRIORITY = 10`
- Put threads on hold:

```
Thread.sleep()  // static
method join()
Thread.yield()  // static method
```

The `join` Method

```
1  public static void main(String[] args) {
2    Thread t = new Thread(new Runner());
3    t.start();
4    ...
5    // Do stuff in parallel with the other thread for a while
6    ...
7    // Wait here for the other thread to finish
8    try {
9      t.join();
10   } catch (InterruptedException e) {
11     // the other thread came back early
12   }
13   ...
14   // Now continue in this thread
15   ...
16 }
// void join(long millisec);
// void join(long millisec, int nanosec);
```

Other Ways to Create Threads

```
1  public class MyThread extends Thread {
2    public void run() {
3      while ( true ) {
4        // do lots of interesting stuff
5        try {
6          Thread.sleep(100);
7        } catch (InterruptedException e) {
8          // sleep interrupted
9        }
10     }
11   }
12
13   public static void main(String args[]) {
14     Thread t = new MyThread();
15     t.start();
16   }
17 }
```

Selecting a Way to Create Threads

- Implement **`Runnable`**:
  - Better object-oriented design
  - Single inheritance
  - Consistency
- Extend **`Thread`**:

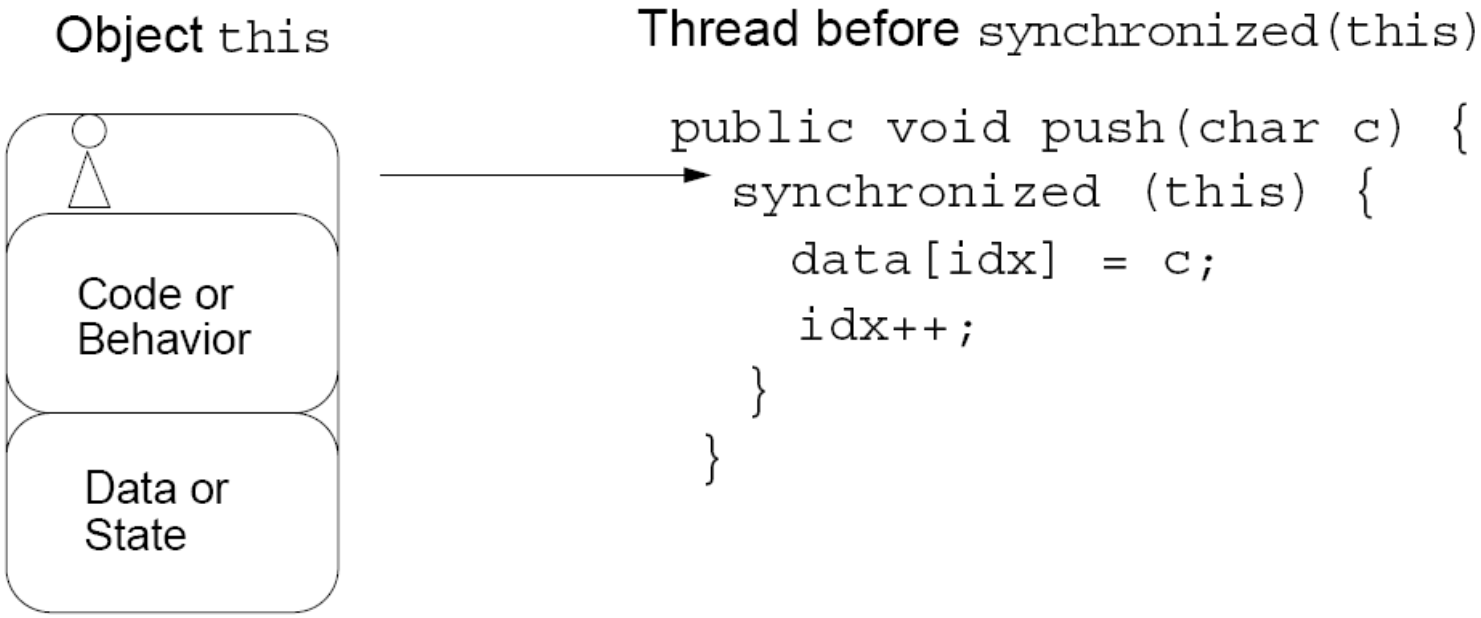
  Simpler code

Using the `synchronized` Keyword

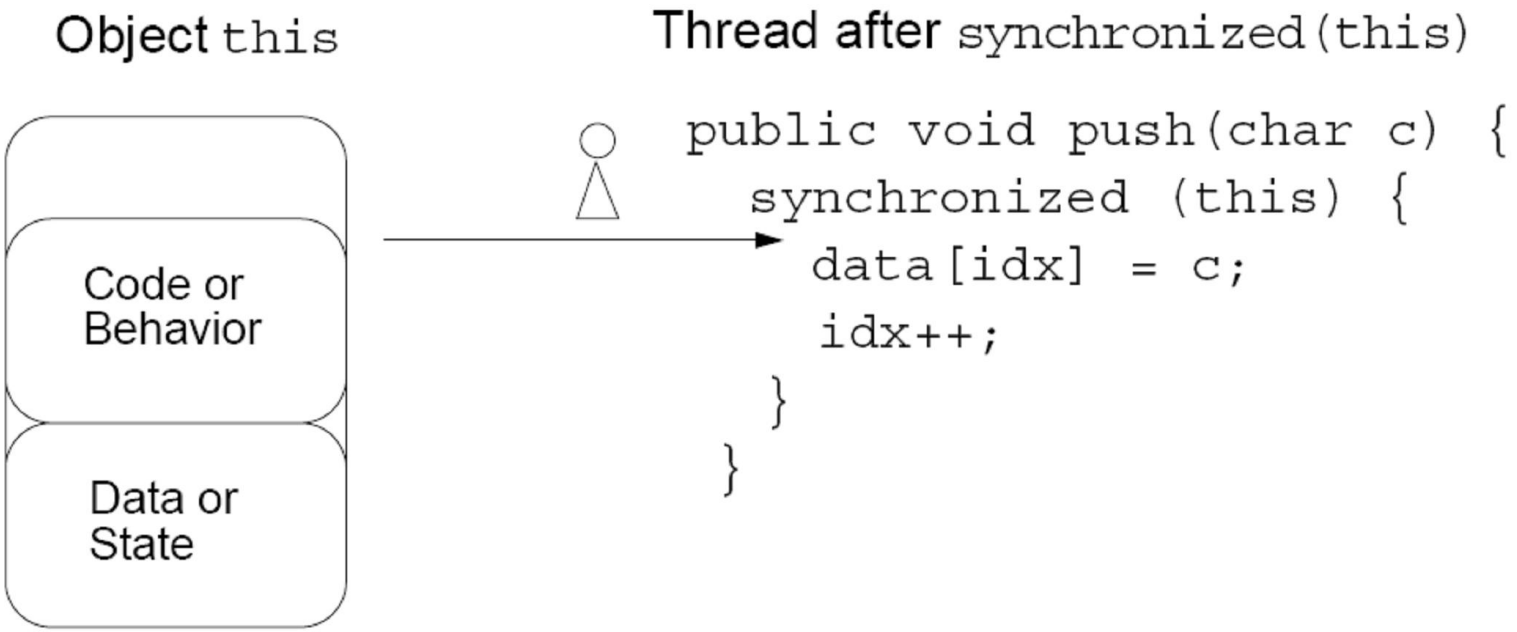
```
1  public class MyStack {
2
3    int idx = 0;
4    char [] data = new char[6];
5
6    public void push(char c) {
7      data[idx] = c;
8      idx++;
9    }
10
11   public char pop() {
12     idx--;
13     return data[idx];
14   }
15 }
```

The Object Lock Flag

- Every object has a flag that is a type of ***lock flag***.
- The **`synchronized`** enables interaction with the lock flag.

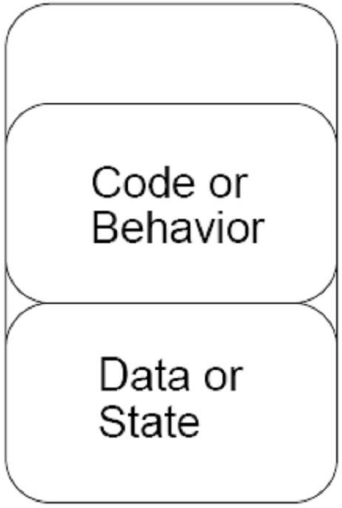


The Object Lock Flag



The Object Lock Flag

Object `this`
lock flag missing



Another thread, trying to
execute `synchronized(this)`

Waiting for object lock

```
public char pop() {
  synchronized (this) {
    idx--;
    return data[idx];
  }
}
```

Releasing the Lock Flag

The lock flag is released in the following events:

- Released when the thread passes the end of the **`synchronized`** code block
- Released automatically when a break, return, or exception is thrown by the **`synchronized`** code block

Using `synchronized` – Putting It Together

- ***All* access** to delicate data should be **`synchronized`**.
- Delicate data protected by **`synchronized`** should be **`private`**.

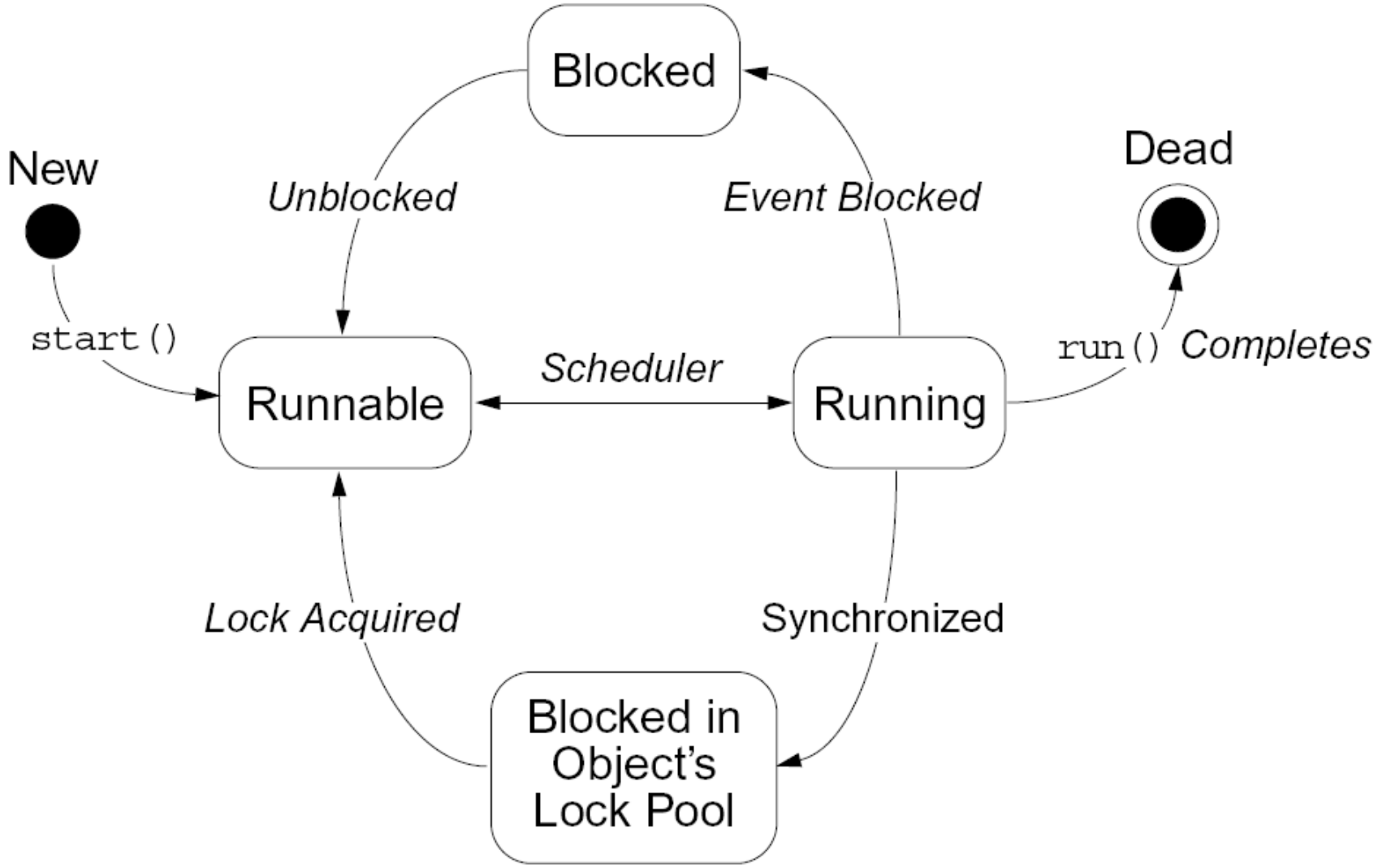
Using `synchronized` – Putting It Together

The following two code segments are equivalent:

```
public void push(char c) {
  synchronized(this) {
    // The push method code
  }
}
```

```
public synchronized void push(char c) {
  // The push method code
}
```

Thread State Diagram With Synchronization



Deadlock

A **deadlock** has the following characteristics:

- It is two threads, each waiting for a lock from the other.
- It is not detected or avoided.
- Deadlock can be avoided by:
  - Deciding on the order to obtain locks
  - Adhering to this order throughout
  - Releasing locks in reverse order

Thread Interaction – `wait` and `notify`

- Scenario:

  Consider yourself and a cab driver as two threads.
- The problem:

  How do you determine when you are at your destination?
- The solution:
  - You notify the cab driver of your destination and relax.
  - The driver drives and notifies you upon arrival at your destination.

Thread Interaction

Thread interactions include:

- The **`wait`** and **`notify`** methods

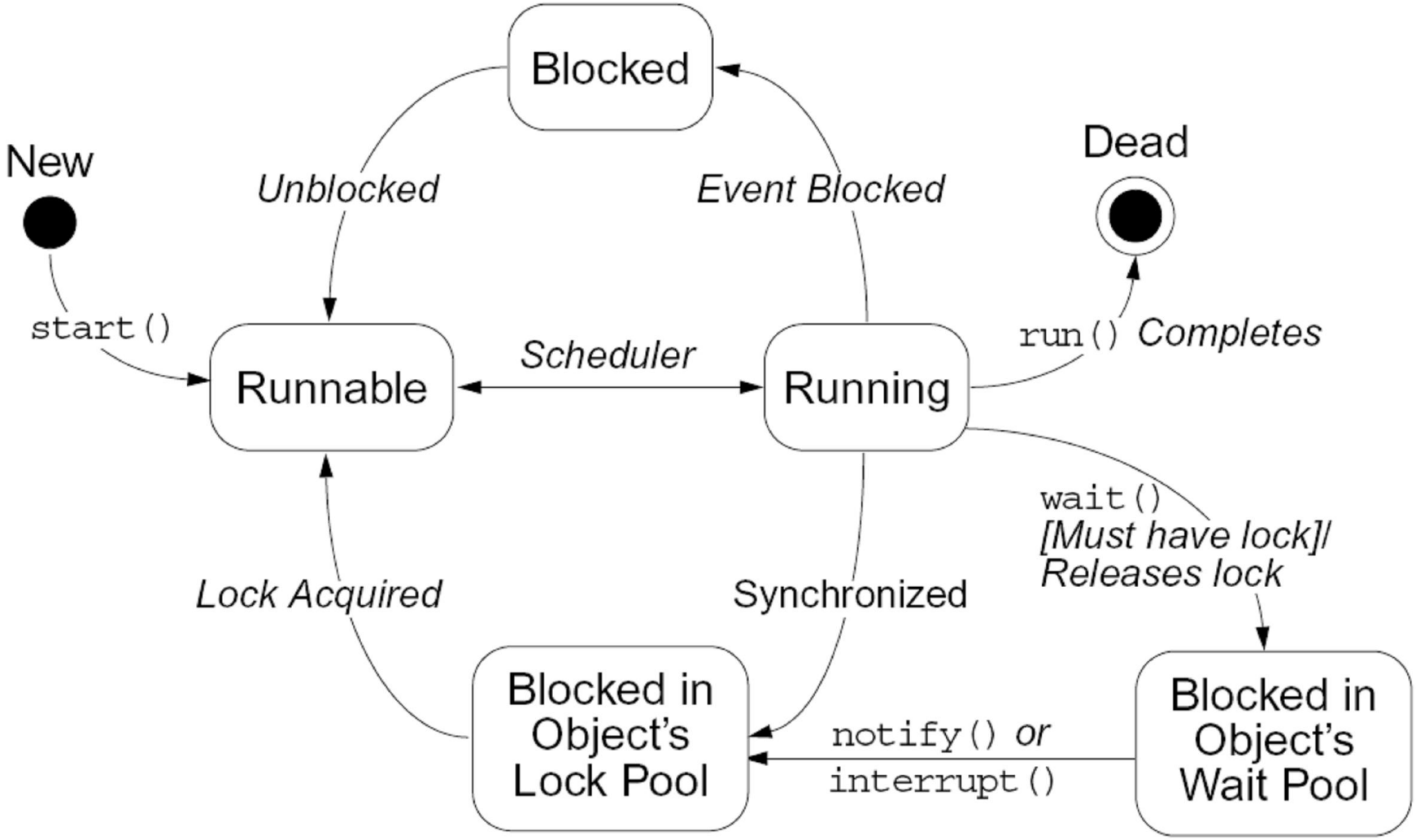
  For a thread to call either **`wait`** or **`notify`** on an object, the thread **must have the lock** for that particular object.

  In other words, **`wait`** and **`notify`** are **called only from** within a **synchronized** block on the instance on which they are being called.

- The pools:
  - Wait pool
  - Lock pool

```
wait()
wait(long timeout)
wait(long timeout, int nanos)
```

Thread State Diagram With `wait` and `notify`



Monitor Model for Synchronization

- Leave shared data in a consistent state.
- Ensure programs can not deadlock.
- Do not put threads expecting different notifications in the same wait pool.

The `Producer` Class

```
1   package mod12;
2
3   public class Producer implements Runnable {
4     private SyncStack theStack;
5     private int num;
6     private static int counter = 1;
7
8     public Producer (SyncStack s) {
9       theStack = s;
10      num = counter++;
11    }
12
```

The `Producer` Class

```
13 public void run() {
14    char c;
15
16    for (int i = 0; i < 200; i++) {
17      c = (char)(Math.random() * 26 +'A');
18      theStack.push(c);
19      System.out.println("Producer" + num + ": " + c);
20      try {
21        Thread.sleep((int)(Math.random() * 300));
22      } catch (InterruptedException e) {
23        // ignore it
24      }
25    }
26  } // END run method
27
28 } // END Producer class
```

The `Consumer` Class

```
1   package mod12;
2
3   public class Consumer implements Runnable {
4     private SyncStack theStack;
5     private int num;
6     private static int counter = 1;
7
8     public Consumer (SyncStack s) {
9       theStack = s;
10      num = counter++;
11    }
12
```

The `Consumer` Class

```
13 public void run() {
14     char c;
15     for (int i = 0; i < 200; i++) {
16       c = theStack.pop();
17       System.out.println("Consumer" + num + ": " + c);
18
19       try {
20         Thread.sleep((int)(Math.random() * 300));
21       } catch (InterruptedException e) {
22         // ignore it
23       }
24     }
25   } // END run method
26
```

The `SyncStack` Class

This is a sketch of the `SyncStack` class:

```
public class SyncStack {

  private List<Character> buffer = new
  ArrayList<Character>(400);

  public synchronized char pop() {
     // pop code
        here
  }

  public synchronized void push(char c) {
    // push code here
  }
}
```

The pop Method

```
9   public synchronized char pop() {
10     char c;
11     while (buffer.size() == 0) {
12       try {
13         this.wait();
14       } catch (InterruptedException e) {
15         // ignore it...
16       }
17     }
18     c = buffer.remove(buffer.size()-1);
19     return c;
20   }
21
```

The `push` Method

```
22 public synchronized void push(char c) {
23     this.notify();
24     buffer.add(c);
25   }
```

The `SyncTest` Class

```
1  package mod12;
2  public class SyncTest {
3    public static void main(String[] args) {
4      SyncStack stack = new SyncStack();
5      Producer p1 = new Producer(stack);
6      Thread prodT1 = new Thread (p1);
7      prodT1.start();
8      Producer p2 = new Producer(stack);
9      Thread prodT2 = new Thread (p2);
10     prodT2.start();
11
12     Consumer c1 = new Consumer(stack);
13     Thread consT1 = new Thread (c1);
14     consT1.start();
15     Consumer c2 = new Consumer(stack);
16     Thread consT2 = new Thread (c2);
17     consT2.start();
18   }
19 }
```

The `SyncTest` Class

```
Producer2: F
Consumer1: F
Producer2: K
Consumer2: K
Producer2: T
Producer1: N
Producer1: V
Consumer2: V
Consumer1: N
Producer2: V
Producer2: U
Consumer2: U
Consumer2: V
Producer1: F
Consumer1: F
Producer2: M
Consumer2: M
Consumer2: T
```

Summary

- Process & Thread concept
- Create threads in Java program, control the code and data used by the thread
- Control the execution of a thread
- Dealing with the shared data
- Use **`wait`** and **`notify`** to for communication
- Use **`synchronized`** keywords to protect shared data
- Advanced topics about threads

Questions or Comments?