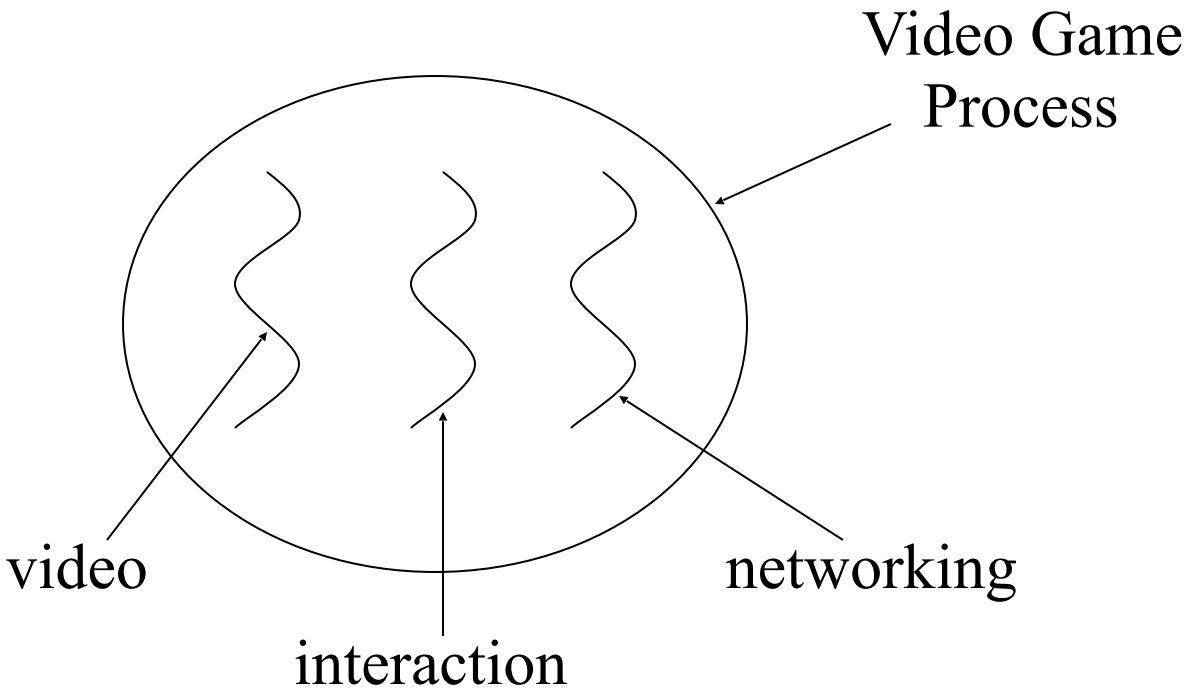
Java Threads

What is a Thread?

- Individual and separate unit of execution that is part of a process
  - multiple threads can work together to accomplish a common goal
- Video Game example
  - one thread for graphics
  - one thread for user interaction
  - one thread for networking

What is a Thread?



Advantages

- easier to program
  - 1 thread per task
- can provide better performance
  - thread only runs when needed
  - no polling to decide what to do
- multiple threads can share resources
- utilize multiple processors if available

Disadvantage

- multiple threads can lead to deadlock
  - much more on this later
- overhead of switching between threads

Creating Threads (method 1)

- extending the Thread class
  - must implement the *run()* method
  - thread ends when *run()* method finishes
  - call *.start()* to get the thread ready to run

Creating Threads Example 1

```
class Output extends Thread {
    private String toSay;
    public Output(String st) {
        toSay = st;
    }
    public void run() {
        try {
            for(;;) {
                System.out.println(toSay);
                sleep(1000);
            }
        } catch(InterruptedException e) {
            System.out.println(e);
        }
    }
}
```

Example 1 (continued)

```
class Program {
    public static void main(String [] args) {
        Output thr1 = new Output("Hello");
        Output thr2 = new Output("There");
        thr1.start();
        thr2.start();
    }
}
```

- main thread is just another thread (happens to start first)
- main thread can end before the others do
- any thread can spawn more threads

Creating Threads (method 2)

- implementing Runnable interface
  - virtually identical to extending Thread class
  - must still define the *run()*method
  - setting up the threads is slightly different

Creating Threads Example 2

```
class Output implements Runnable {
    private String toSay;
    public Output(String st) {
        toSay = st;
    }
    public void run() {
        try {
            for(;;) {
                System.out.println(toSay);
                Thread.sleep(1000);
            }
        } catch(InterruptedException e) {
            System.out.println(e);
        }
    }
}
```

Example 2 (continued)

```
class Program {
    public static void main(String [] args) {
        Output out1 = new Output(“Hello”);
        Output out2 = new Output(“There”);
        Thread thr1 = new Thread(out1);
        Thread thr2 = new Thread(out2);
        thr1.start();
        thr2.start();
    }
}
```

- main is a bit more complex
- everything else identical for the most part

Advantage of Using Runnable

- remember - can only extend one class
- implementing runnable allows class to extend something else

Controlling Java Threads

- _.*start()*: begins a thread running
- *wait()* and *notify()*: for synchronization
  – more on this later
- _.*stop()*: kills a specific thread (deprecated)
- _.*suspend()* and *resume()*: deprecated
- _.*join()*: wait for specific thread to finish
- _.*setPriority()*: 0 to 10 (MIN_PRIORITY to MAX_PRIORITY); 5 is default (NORM_PRIORITY)

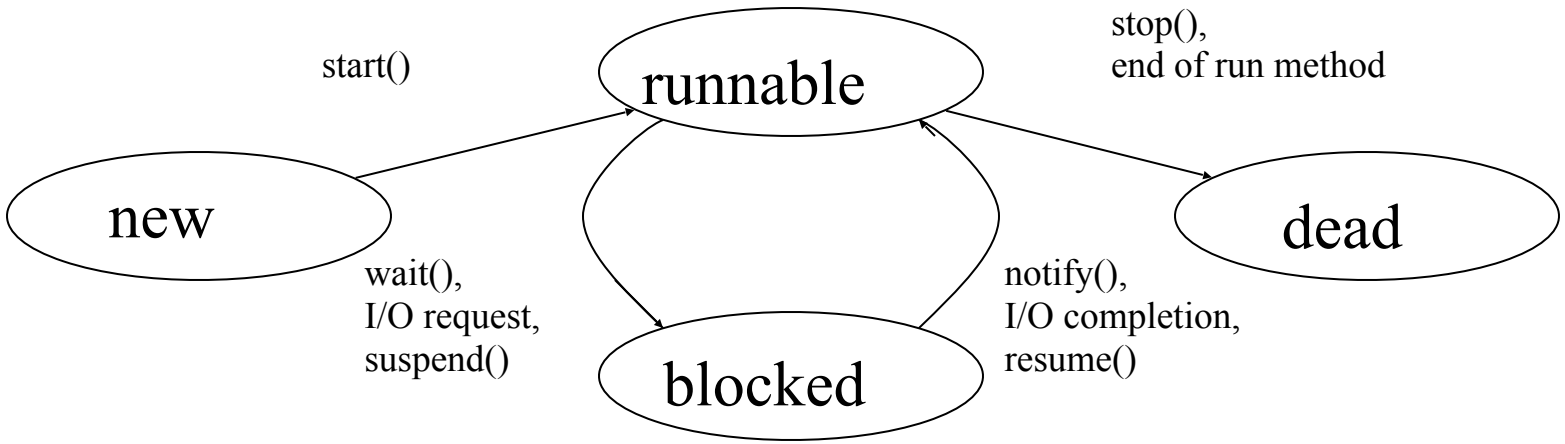
Java Thread Scheduling

- highest priority thread runs
  - if more than one, arbitrary
- *yield()*: current thread gives up processor so another of equal priority can run
  - if none of equal priority, it runs again
- *sleep(msec)*: stop executing for set time
  - lower priority thread can run

States of Java Threads

- 4 separate states
  - new: just created but not started
  - runnable: created, started, and able to run
  - blocked: created and started but unable to run because it is waiting for some event to occur
  - dead: thread has finished or been stopped

States of Java Threads



Java Thread Example 1

```
class Job implements Runnable {
    private static Thread [] jobs = new Thread[4];
    private int threadID;
    public Job(int ID) {
        threadID = ID;
    }
    public void run() { do something }
    public static void main(String [] args) {
        for(int i=0; i<jobs.length; i++) {
            jobs[i] = new Thread(new Job(i));
            jobs[i].start();
        }
        try {
            for(int i=0; i<jobs.length; i++) {
                jobs[i].join();
            }
        } catch(InterruptedException e) { System.out.println(e); }
    }
}
```

Java Thread Example 2

```
class Schedule implements Runnable {
        private static Thread [] jobs = new Thread[4];
        private int threadID;
        public Schedule(int ID) {
                threadID = ID;
        }
        public void run() { do something }
        public static void  main(String [] args) {
                int nextThread = 0;
                setPriority(Thread.MAX_PRIORITY);
                for(int i=0; i<jobs.length; i++) {
                   jobs[i] = new Thread(new Job(i));
                   jobs[i].setPriority(Thread.MIN_PRIORITY);
                   jobs[i].start();
                }
                try {
                   for(;;) {
                      jobs[nextThread].setPriority(Thread.NORM_PRIORITY);
                      Thread.sleep(1000);
                      jobs[nextThread].setPriority(Thread.MIN_PRIORITY);
                      nextThread = (nextThread + 1) % jobs.length;
                   }
                } catch(InterruptedException e) { System.out.println(e); }
        }
}
```

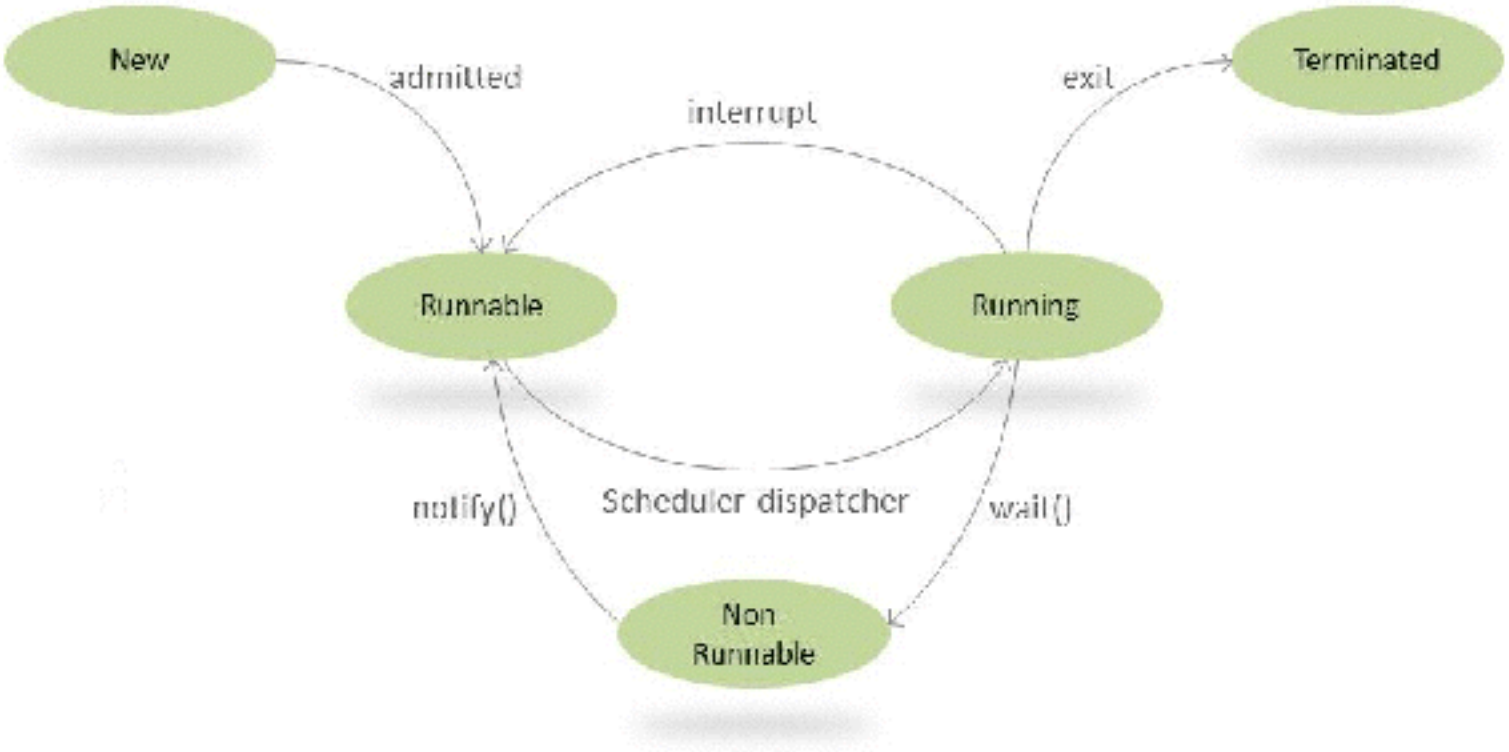
Java8 Concurrency Runnable Lambda function

```
Runnable task = () -> {
    String threadName = Thread.currentThread().getName();
    System.out.println("Hello " + threadName);
};

task.run();

Thread thread = new Thread(task);
thread.start();

System.out.println(“Done!");
```

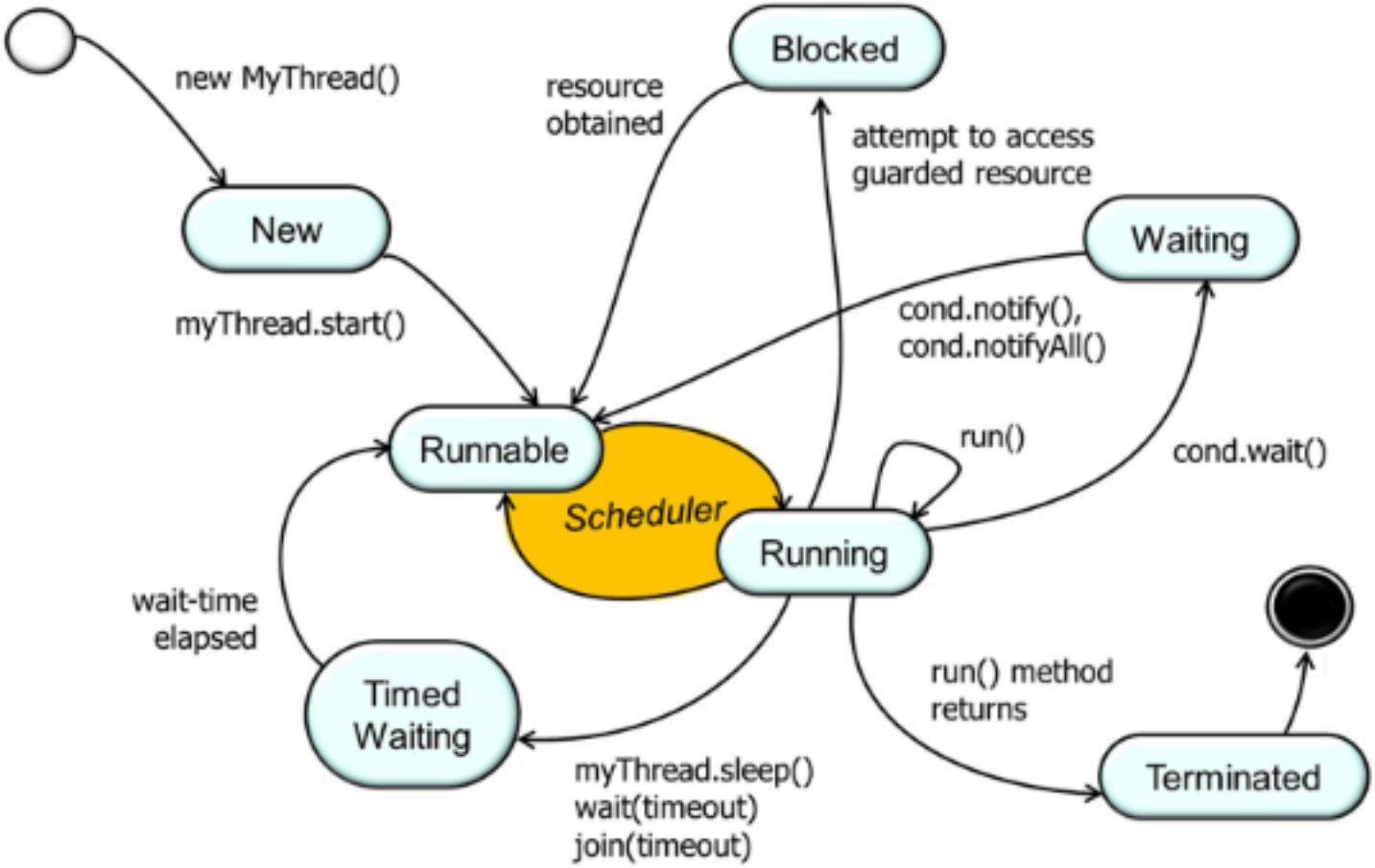
```
Hello main
Hello Thread-0
Done!
```

Thread State Diagram

- *Object.wait()* – to suspend a thread
- *Object.notify()* – to wake a thread up



Thread State Diagram Example



Thread State Diagram Description

1. When a java process initialize a new thread, its in the New state.
2. After the program calls the start method of the thread, the thread state transition to the Runnable state
3. When the thread scheduler selects this thread for execution, the thread state is changed to RUNNING state in which the java virtual machine invokes the Thread run hook method
4. If there is any wait operation called from the thread (sleep, wait or join method), the thread state transition to Timed Waiting state
5. When this waiting time elapses, the thread is back to RUNNABLE state and is eligible for the Thread scheduler selection .
6. After the thread start executing its run method, it may access a guarded resources like a synchronized method or a state protected by a monitor lock which will transition the thread to BLOCKED state if the resource is not yet available
7. Once the resource is available, the thread can access the guarded resource and will transition to RUNNABLE state
8. Once the thread start running again, it might wait for a monitor condition like wait() method changing the thread state to WAITING state.
9. When the other threads notifies the wait condition with notify or notifyAll method, the thread again becomes RUNNABLE
10. Finally, when a RUNNING thread completes its execution, it gets TERMINATED.

THREAD Send/Receive EXAMPLE WIAT/NOTIFY

```
public class Data {
    private String packet;

    // True if receiver should wait
    // False if sender should wait
    private boolean transfer = true;

    public synchronized void send(String packet) {
        while (!transfer) {
            try {
                wait();
            } catch (InterruptedException e)  {
                Thread.currentThread().interrupt();
                Log.error("Thread interrupted", e);
            }
        }
        transfer = false;

        this.packet = packet;
        notifyAll();
    }
    public synchronized String receive() {
        while (transfer) {
            try {
                wait();
            } catch (InterruptedException e)  {
                Thread.currentThread().interrupt();
                Log.error("Thread interrupted", e);
            }
        }
        transfer = true;

        notifyAll();
        return packet;
    } }
```

Steps:

- The packet variable denotes the data that is being transferred over the network
- We have a boolean variable transfer – which the Sender and Receiver will use for synchronization:
  - If this variable is true, then the Receiver should wait for Sender to send the message
  - If it's false, then Sender should wait for Receiver to receive the message
- The Sender uses send() method to send data to the Receiver:
  - If transfer is false, we'll wait by calling wait() on this thread
  - But when it is true, we toggle the status, set our message and call notifyAll() to wake up other threads to specify that a significant event has occurred and they can check if they can continue execution
- Similarly, the Receiver will use receive() method:
  - If the transfer was set to false by Sender, then only it will proceed, otherwise we'll call wait() on this thread
  - When the condition is met, we toggle the status, notify all waiting threads to wake up and return the data packet that was Receiver

THREAD Array Data WIAT/NOTIFY EXAMPLE

```
public class Input {
    int index;
    int[] input = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};

    public Input(){
        index = 0;
    }

    public void print(int index){
        System.out.println(input[index]);
    }

    synchronized public int getIndex(){
        if(index == 15)
            return -1;
        return index++;
    }
}

public class MyThread implements Runnable{

    Input ip;
    Object lock;

    public MyThread(Input ip, Object lock){
        this.ip = ip;
        this.lock = lock;
    }

    @Override
    public void run() {
        int index = -1;
        while((index=ip.getIndex())!=-1){
            synchronized(lock) {
                System.out.println(Thread.currentThread().getName());
                ip.print(index);
            }
        }
    }
}
public class Caller {

    public static void main(String[] args) throws InterruptedException {
        Input ip = new Input();
        Object lock = new Object();
        Thread t1 = new Thread(new MyThread(ip, lock), "Thread1");
        Thread t2 = new Thread(new MyThread(ip, lock), "Thread2");
        t1.start();
        t2.start();
        t1.join();
        t2.join();
    }
}
```

```
Runnable runnableTask = () -> {
    try {
        TimeUnit.MILLISECONDS.sleep(300);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
};

Callable<String> callableTask = () -> {
    TimeUnit.MILLISECONDS.sleep(300);
    return "Task's execution";
};

List<Callable<String>> callableTasks = new ArrayList<>();
callableTasks.add(callableTask);
callableTasks.add(callableTask);
callableTasks.add(callableTask);
```