hi-tech college + JOHN BRYCE Leading in IT Education a matrix company

Java Threads

What is a Thread ?

- A thread is a single sequential flow of control within a program
- A simple thread(like a sequential program) has:
  - a beginning
  - a sequence
  - an end
  - at any given time during the runtime of the thread, there is a single point of execution.
- Also known as: *lightweight process*, *execution context*
- Are not like processes – but does appear in the OS thread list

Multiple Threads in a Single Program

- running at the same time(concurrently) and performing different tasks
- Example: Web browser
  - scroll page
  - downloading applet
  - play animation, sound
  - print page
  - downloading a new page

Techniques for Using of Threads

- Subclassing Thread and Overriding run
- Implementing the Runnable interface

Subclassing Thread and Overriding run

```
public class SimpleThread extends Thread {
        public SimpleThread(String str) {
            super(str);
        }
        public void run() {
            for (int i = 0; i < 10; i++) {
                System.out.println(i + " " + getName());
                try {
                                sleep((long)(Math.random() * 1000));
                } catch (InterruptedException e) {}
            }
            System.out.println("DONE! " + getName());
        }
}
```

Main method:

```
public class TwoThreadsTest {
        public static void main (String[] args) {
                new SimpleThread("Jamaica").start();
                new SimpleThread("Fiji").start();
        }
}
```

Output:

```
0 Jamaica
0 Fiji
1 Fiji
1 Jamaica
2 Jamaica
2 Fiji
3 Fiji
…
6 Jamaica
7 Jamaica
7 Fiji
8 Fiji
9 Fiji
8 Jamaica
DONE! Fiji
9 Jamaica
DONE! Jamaica
```

Implementing Runnable

```
public class SimpleRunnable implements Runnable{

        public void run() {
            for (int i = 0; i < 10; i++) {
                System.out.println(i + " " + Thread.getCurrentThread().getName());
                try {
                                sleep((long)(Math.random() * 1000));
                } catch (InterruptedException e) {}
            }
            System.out.println("DONE! " + Thread.getCurrentThread(). getName());
        }
}
```

Main method:

```
public class TwoThreadsTest {
        public static void main (String[] args) {
                        SimpleRunnable  runner=new SimpleRunnable ();
                        Thread t1=new Thread(runner,"Jamaica");
                        Thread t2=new Thread(runner,"Fiji");
                        t1.start();
                        t2.start();
        }
}
```

Output:

```
0 Jamaica
0 Fiji
1 Fiji
1 Jamaica
2 Jamaica
2 Fiji
3 Fiji
…
6 Jamaica
7 Jamaica
7 Fiji
8 Fiji
9 Fiji
8 Jamaica
DONE! Fiji
9 Jamaica
DONE! Jamaica
```

Which way is better ?

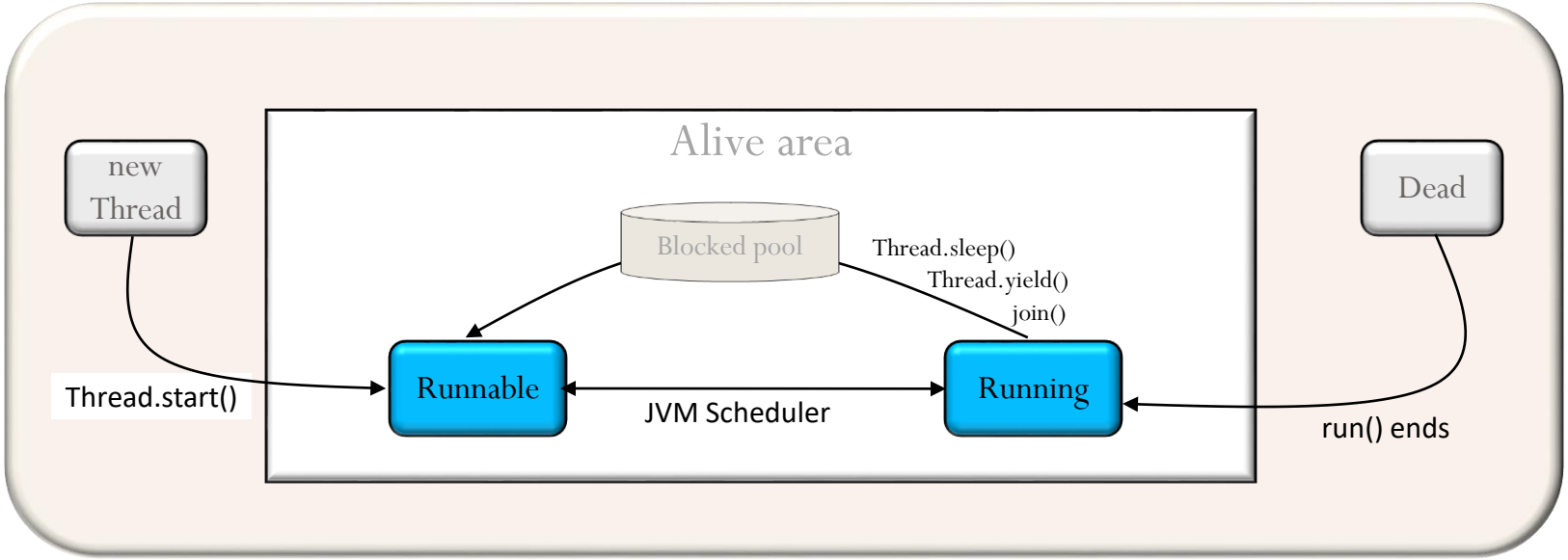
Runnable

- When your class must subclass some other class
- When you would like to have a light-weight runnable objects
- That's what interfaces are for
- Problems – might be less convenient for coding

Thread

- When your runnable uses some of the thread attributes directly
- Less coding
- Problems – single inheritance in Java

The Life Cycle of a Thread



Basic states of the Thread:

- New Thread
- Running
- Not Runnable
- Dead

States of the Thread – New Thread

Main method:

```
public class TwoThreadsTest {
        public static void main (String[] args) {
                SimpleRunnable  runner=new SimpleRunnable ();
                Thread t1=new Thread(runner,"Jamaica");
                Thread t2=new Thread(runner,"Fiji");
                t1.start();
                t2.start();
        }
}
```

- no system resource have been allocated for it yet
- can only be started
- calling any method besides causes an *IllegalThreadStateException*

States of the Thread - Running

Main method:

```
public class TwoThreadsTest {
        public static void main (String[] args) {
                SimpleRunnable  runner=new SimpleRunnable ();
                Thread t1=new Thread(runner,"Jamaica");
                Thread t2=new Thread(runner,"Fiji");
                t1.start();
                t2.start();
        }
}
```

- creates the system resources necessary to run the thread
- schedules the thread to run
- calls the thread's run method

States of the Thread – Not Runnable

A thread becomes Not Runnable when one of these events occurs:

-- Its sleep(), yield() method is invoked

- One thread uses *join()* on another and becomes blocked
- The thread calls the *wait()* method to wait for a specific condition to be satisfied.
- The thread is blocked on I/O

```
public class SimpleRunnable implements Runnable{

        public void run() {
            for (int i = 0; i < 10; i++) {
                System.out.println(i + " " + getName());
                try {
                                    sleep((long)(Math.random() * 1000));
                } catch (InterruptedException e) {}
            }
            System.out.println("DONE! " + getName());
        }
}
```

States of the Thread – Not Runnable

Basic control:

- Sleep – move the thread to a non-Runnable state for a period of time (ms)
  - Usually the simplest way to delay threads or main
  - Note: blocks the thread at least to the specified time – not exactly
- Yield – move the Running thread to the Rannable pool (Equals to *sleep(0)*)
  - Usually for giving other low priority thread a chance to run
- Join – move the running thread to a non-Runnable state until a specific thread ends
  - Delays the caller until the referenced thread ends
  - Is absolute – not like priority
- All methods throws *InterruptedException*
  - When thread are out of the blocking state before time
  - Might happen due to OS activity
- Blocked threads returns to runnable state
  - never to running (!)

```
try {
        Thread.sleep(3000)
} catch (InterruptedException e) {}
```

```
try {
        Thread.yield()
} catch (InterruptedException e) {}
```

```
…
Thread t=new Thread(runner);
t.start();
try {
        t.join()
} catch (InterruptedException e) {}
// all the work here happens after t ends
…
```

States of the Thread – Not Runnable

The following list describes the escape route for every entrance into the Not Runnable state:

- If a thread has been put to sleep, then the specified number of milliseconds must elapse.
- If a thread is waiting for a condition, then another object must notify the waiting thread of a change in condition by calling *notify()* or *notifyAll()* - later
- If a thread is blocked on I/O, then the I/O must complete.

States of the Thread – Dead

- the run method must terminate naturally
- stop method – ***deprecated!!!***
  - *This method is inherently unsafe. Stopping a thread with Thread.stop causes it to unlock all of the monitors that it has locked (as a natural consequence of the unchecked ThreadDeath exception propagating up the stack)*

Terminating a Thread

```
public class Runner implements Runnable {
    private boolean timeToQuit=false;

    public void run() {
          while ( ! timeToQuit ) {
             ...
          }
         // clean up before run() ends
    }

    public void stopRunning() {
        timeToQuit=true;
    }
}
```

```
public class ThreadController {
    private Runner r = new Runner();
    private Thread t = new Thread(r);

    public void startThread() {
         t.start();
    }

    public void stopThread() {
        // use specific instance of Runner
         r.stopRunning();
    }
}
```

Daemon Threads

- Threads keep on running even after main thread ends
- Means that the VM still ‘on the air’ until the last thread dies
- In order to kill a thread when system exits it has to be a daemon
- Thread can be set to behave as daemon via *setDaemon(boolean)*
- Thread can be checked *via isDaemon()*

- *Garbage collection is a daemon thread*
  - therefore doesn’t last after system exit
  - That’s why sometimes object may never get the *finalize()* call

The isAlive Method

- Returns *true* if:
  - If the thread has been started and not stopped
  - the thread is Runnable or Not Runnable
- Returns *false* if:
  - the thread is New Thread or Dead

Understanding Thread Priority

- The execution of multiple threads on a single CPU, in some order, is called *scheduling*
- The Java runtime supports a very simple, deterministic scheduling algorithm known as *fixed priority scheduling*
- Each Java thread is given a numeric priority between *MIN_PRIORITY* and *MAX_PRIORITY* (constants defined in the Thread class)
- At any given time, when multiple threads are ready to be executed, the thread with the highest priority is chosen for execution
- A lower priority will start executing when the current thread
  - stops
  - yields
  - becomes not runnable
- If two threads of the same priority are waiting for the CPU, the scheduler chooses one of them to run in a round-robin fashion

Fixed Priority Scheduling

- The chosen thread will run until one of the following conditions is true:
  - A higher priority thread becomes runnable
  - It yields, or its run method exits
  - On systems that support *time-slicing*, its time allotment has expired

- Regarding Time-slicing
  - The Java runtime does not implement (and therefore does not guarantee) time-slicing
  - Your Java programs **should not rely** on time-slicing as it may produce different results on different systems

Synchronizing Threads

- Separate, concurrently running threads do share data and must consider the state and activities of other threads
- One such set of programming situations are known as **producer/consumer** scenarios where the producer generates a stream of data which then is consumed by a consumer
- The code segments within a program that access the same object from separate, concurrent threads are called *critical sections*
- In the Java language, a critical section can be a block or a method and are identified with the synchronized keyword
- The Java platform then associates a lock with every object that has synchronized code.

Locking an Object - Example

```
public class Car{
        ...
        public synchronized void drive(){
            //this Car is locked by a Driver
            ...
            //this Car is unlocked by a Driver
        }
}
```

- Only one Driver can drive the Car at a time
- Only one thread at a time can own an object's monitor

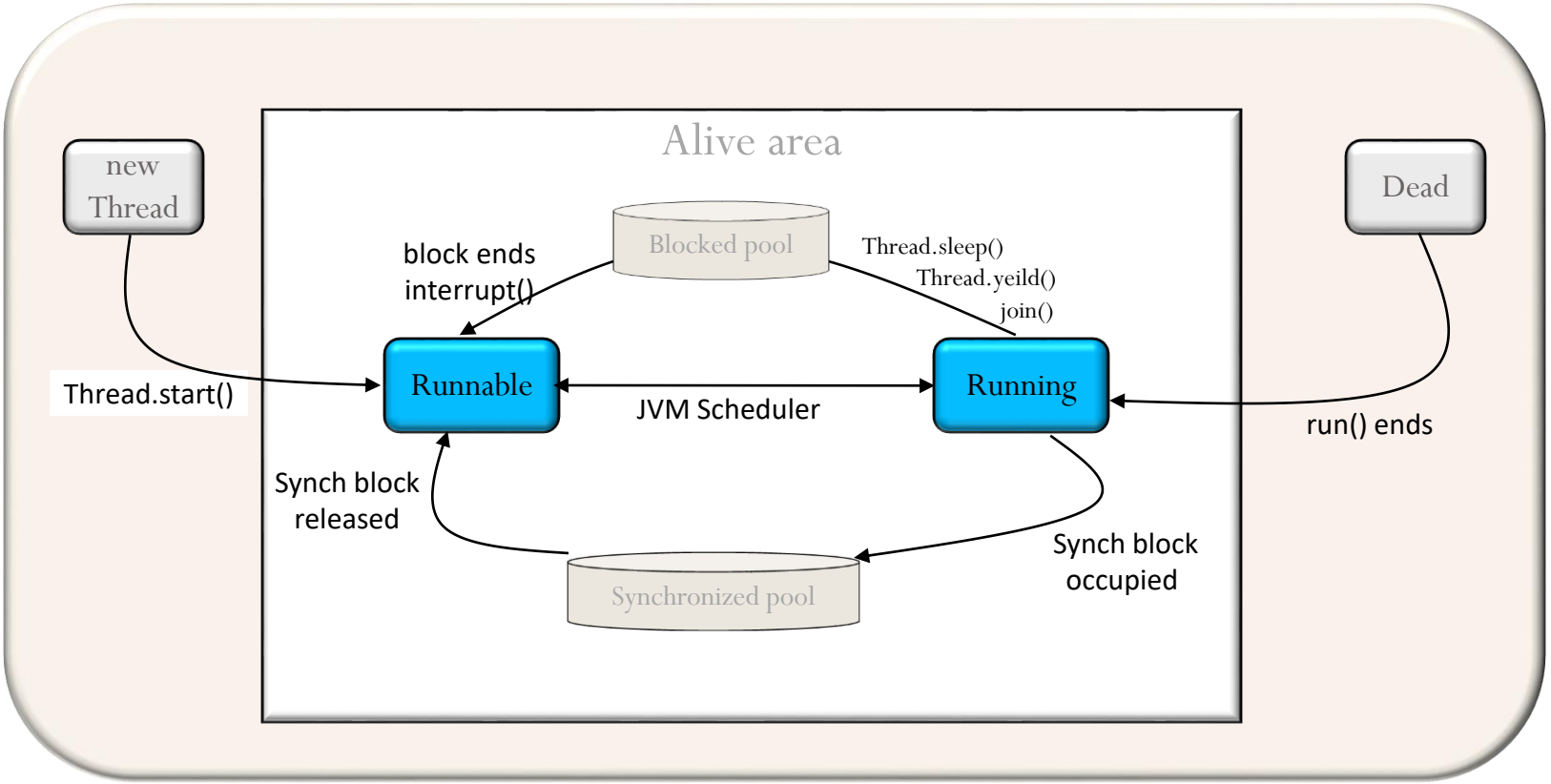
```
public class Car{
        ...
        //It's not synchronized now
        public void drive(){…}
}


public class Person{
        public void use(Car c){
            synchronized(c){
                ...
                c.drive();
                ...
            }
        }
}
```

- More than one Driver can drive the Car at a time

Object Lock Flag

- Every object has a flag that can be thought of as a "lock flag"
- synchronized allows interaction with the lock flag
- Released when :
  - the thread passes the end of the synchronized code block
  - Automatically released when a break or exception is thrown by the synchronized code block

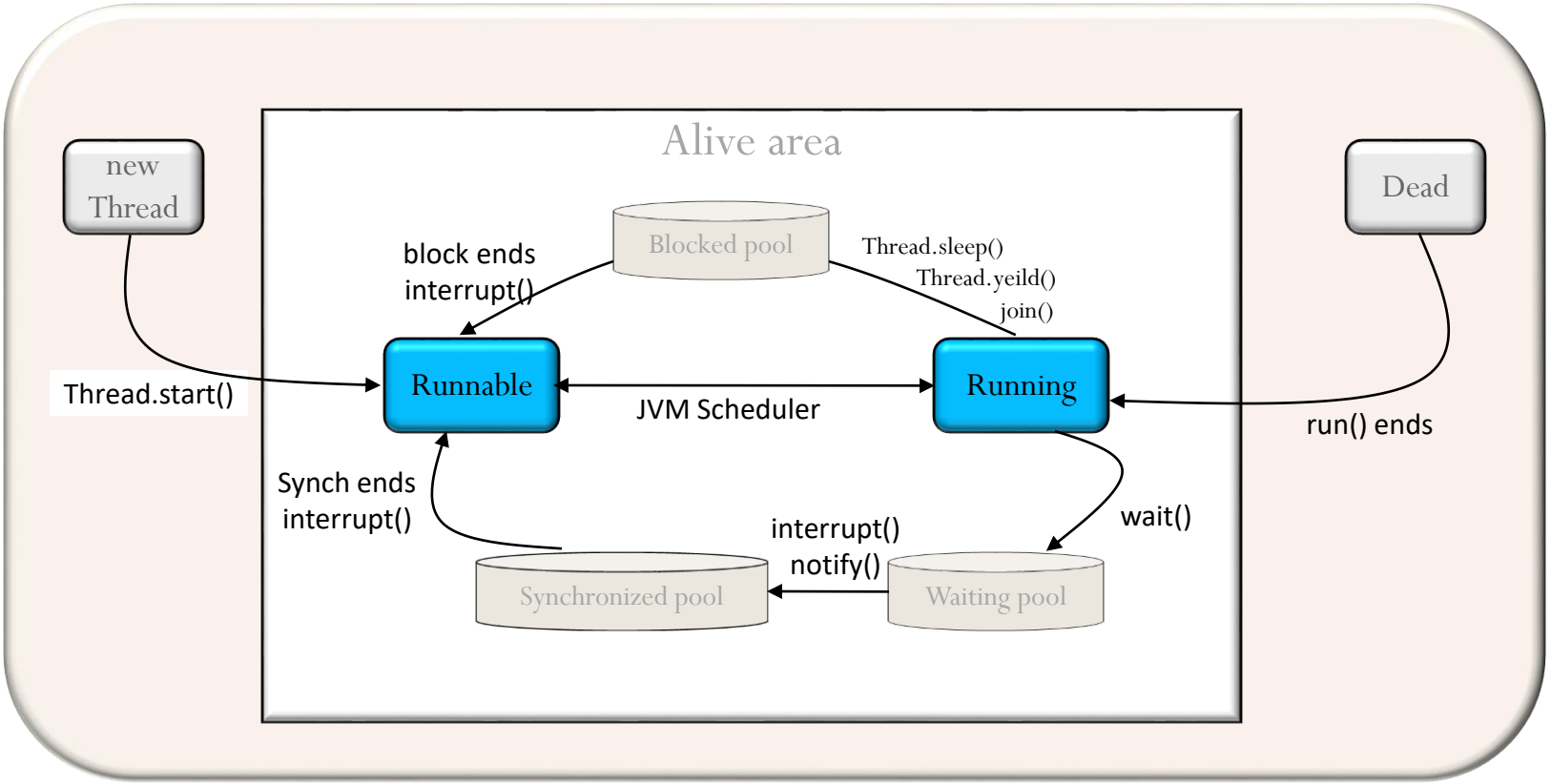
Using the notify and wait Methods



Using the notify and wait Methods

- *wait(), wait(long timeout)*
  - Causes current thread to wait until another thread invokes the *notify()* method or the *notifyAll()* method for this object, or a specified amount of time has elapsed.
- *notify()*
  - Wakes up a single thread that is waiting on this object's monitor
- *notifyAll()*
  - Wakes up all threads that are waiting on this object's monitor

Using the notify and wait Methods



Precondition to wait and notify Methods

- The current thread must own this object's monitor
- A thread becomes the owner of the object's monitor in one of three ways:
  - By executing a synchronized instance method
  - By executing the body of a synchronized
  - For objects of type Class, by executing a synchronized static method
- Only one thread at a time can own an object's monitor

Reentrant Locks

- The Java runtime system allows a thread to re-acquire a lock that it already holds because Java locks are reentrant

```
public class Reentrant {
        public synchronized void a() {
            b();
            System.out.println("here I am, in a()");
        }
        public synchronized void b() {
            System.out.println("here I am, in b()");
        }
}
```

Output:

```
here I am, in b()
here I am, in a()
```

The method interrupt

Interrupts the thread

- If the current thread is interrupted by another thread while it is waiting, then an *InterruptedException* is thrown
- This exception is not thrown until the lock status of this object has been restored as described above
  - This method should only be called by a thread that is the owner of this object's monitor

Deadlock

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