Java Threads

- The class java.lang.Thread
  - Implements the basic threading abstraction
  - Can extend this class to create your own threads
- The interface java.langRunnable
  - Can create a thread by passing it a class that implements this interface
  - Favors composition over inheritance; more flexible

Extending class Thread

- Can build a thread class by extending java.lang.Thread
- Must supply a public void run() method
- Start a thread by invoking the start() method
- When a thread starts, executes run()
- When run() returns, thread is finished/dead
Example: Synchronous alarms

```java
while (true) {
    System.out.print("Alarm> ");

    // read user input
    String line = b.readLine();
    parseInput(line);

    // wait (in secs)
    try {
        Thread.sleep(timeout * 1000);
    } catch (InterruptedException e) {
    }
    System.out.println("("+timeout+") "+msg);
}
```

Making it Threaded (1)

```java
public class AlarmThread extends Thread {
    private String msg = null;
    private int timeout = 0;

    public AlarmThread(String msg, int time) {
        this.msg = msg;
        this.timeout = time;
    }

    public void run() {
        try {
            Thread.sleep(timeout * 1000);
        } catch (InterruptedException e) {
        }
        System.out.println("("+timeout+") "+msg);
    }
}
```
while (true) {
    System.out.print("Alarm> ");

    // read user input
    String line = b.readLine();
    Thread t = parseInput(line);

    // wait (in secs) asynchronously
    if (t != null)
        t.start();
}

Runnable interface

- Extending Thread means can’t extend any other class
- Instead implement **Runnable**
  - declares that the class has a void run() method
- Can construct a new Thread
  - and give it an object of type Runnable as an argument to the constructor
  - Thread(Runnable target)
  - Thread(Runnable target, String name)
Thread example revisited

```java
public class AlarmRunnable implements Runnable {
    private String msg = null;
    private int timeout = 0;

    public AlarmRunnable(String msg, int time) {
        this.msg = msg;
        this.timeout = time;
    }

    public void run() {
        try {
            Thread.sleep(timeout * 1000);
        } catch (InterruptedException e) { }
        System.out.println("("+timeout") "+msg);
    }
}
```

Change is in parseInput

- Old parseInput does
  – return new AlarmThread(m,t);
- New parseInput does
  – return new Thread(new AlarmRunnable(m,t));
- Code in while loop doesn’t change
Another example

```java
public class ThreadDemo implements Runnable {
    public void run() {
        for (int i = 5; i > 0; i--) {
            System.out.println(i);
            try { Thread.sleep(1000); } catch(InterruptedException e) { }
        }
        System.out.println("exiting " + Thread.currentThread());
    }
    public static void main(String [] args) {
        Thread t = new Thread(new ThreadDemo(), "Demo Thread");
        System.out.println("main thread: " + Thread.currentThread());
        System.out.println("Thread created: " + t);
        t.start();
        try { Thread.sleep(3000); } catch (InterruptedException e) {}
        System.out.println("exiting " + Thread.currentThread());
    }
}
```

InterruptedException

- A number of thread methods throw it
  - really means: “wake-up call!”
- `interrupt()` tries to wake up a thread
- Won’t disturb the thread if it is working
- Will wake up the thread if it is sleeping, or otherwise waiting (or will do so when such a state is entered)
  - Thrown by `sleep()`, `join()`, `wait()`
Thread scheduling

• When multiple threads share a CPU, must decide:
  – When the current thread should stop running
  – What thread to run next
• A thread can voluntarily yield() the CPU
• Preemptive schedulers can de-schedule the current thread at any time
  – But not all JVM implementations use preemptive scheduling; so a thread stuck in a loop may never yield by itself. Therefore, put yield() into loops
• Threads are descheduled whenever they block (e.g. on a lock or on I/O) or go to sleep

Which thread to run next?

• The scheduler looks at all of the runnable threads; these will include threads that were unblocked because
  – A lock was released
  – I/O became available
  – They finished sleeping, etc.
• Of these threads, it considers the thread’s priority. This can be set with setPriority(). Higher priority threads get preference.
  – Oftentimes, threads waiting for I/O are also preferred.
Simple thread methods

- void start()
- boolean isAlive()
- void setPriority(int newPriority)
  - thread scheduler might respect priority
- void join() throws InterruptedException
  - waits for a thread to die/finish

Example: threaded, sync alarm

```java
while (true) {
    System.out.print("Alarm> ");

    // read user input
    String line = b.readLine();
    Thread t = parseInput(line);

    // wait (in secs) asynchronously
    if (t != null)
        t.start();
    // wait for the thread to complete
    t.join();
}
```
Simple static thread methods

- void yield()
  - Give up the CPU
- void sleep(long milliseconds)
  - Sleep for the given period
- Thread currentThread()
  - Thread object for currently executing thread
- All apply to thread invoking the method

Daemon threads

- void setDaemon(boolean on)
  - Marks thread as a daemon thread
- By default, thread acquires status of thread that spawned it
- Program execution terminates when no threads running except daemons
Example - why synchronization?

```java
class UnSyncTest extends Thread {
    String msg;
    public UnSyncTest(String s) {
        msg = s; start(); }
    public void run() {
        System.out.println("[" + msg);
        try { Thread.sleep(1000); } 
        catch(InterruptedException e) {} 
        System.out.println("]");
        System.out.println("\n");
    public static void main(String [] args) {
        new UnSyncTest("Hello");
        new UnSyncTest("UnSynchronized");
        new UnSyncTest("World");
    }
}
```

Synchronization Topics

- Locks
- `synchronized` statements and methods
- `wait` and `notify`
- Deadlock
Locks

- Any Object subclass can act as a lock
- Only one thread can hold the lock on an object
  - other threads block until they can acquire it
- If your thread already holds the lock on an object
  - can lock many times
  - Lock is released when object unlocked the corresponding number of times
- No way to only attempt to acquire a lock
  - Either succeeds, or blocks the thread

Synchronized methods

- A method can be synchronized
  - add synchronized modifier before return type
- Obtains the lock on object referenced by this, before executing method
  - releases lock when method completes
- For a static synchronized method
  - locks the class object
Synchronized statement

- `synchronized (obj) { statements }`
- Obtains the lock on `obj` before executing statements in block
- Releases the lock when the statements block completes
- Finer-grained than synchronized method

Synchronization example

```java
class SyncTest extends Thread {
    String msg;
    public SyncTest(String s) {
        msg = s;
        start();
    }
    public void run() {
        synchronized (SyncTest.class) {
            System.out.print("[" + msg);
            try { Thread.sleep(1000); } catch (InterruptedException e) {};
            System.out.println");
        }
    }
    public static void main(String [] args) {
        new SyncTest("Hello");
        new SyncTest("Synchronized");
        new SyncTest("World");
    }
}
```
Wait and Notify

- Must be called inside **synchronized** method or block of statements
- **a.wait()**
  - releases the lock on a
  - adds the thread to the *wait set for a*
  - blocks the thread
- **a.wait(int m)**
  - limits wait time to *m* milliseconds (but see below)

Wait and Notify (cont.)

- **a.notify()** resumes one thread from a’s wait list
  - and removes it from wait set
  - no control over which thread
- **a.notifyAll()** resumes *all* threads on a’s wait list
- resumed thread(s) must reacquire lock before continuing
- **wait** doesn’t give up locks on any other objects
  - e.g., acquired by methods that called this one
Producer/Consumer Example –
Too Much Synchronization

```java
public class ProducerConsumer {
    private boolean ready = false;
    private Object obj;
    public ProducerConsumer() { }
    public ProducerConsumer(Object o) {
        obj = o;  ready = true;  }
    synchronized void produce(Object o) {
        while (ready) wait();
        obj = o;  ready = true;
        notifyAll();
    }
    synchronized Object consume() {
        synchronized Object consume() {
            while (!ready) wait();
            ready = false;
            notifyAll();
            return obj;
        }
    }
}
```

Changed example –
Attempt to refine synch.

```java
synchronized void produce(Object o) {
    while (ready) {
        wait();
        if (ready) notify();
    }
    obj = o;  ready = true;
    notify();
}

Doesn't work well – no guarantee about who will get woken up
```
A Better Solution

synchronized void produce(Object o) {
    while (ready) {
        synchronized (empty) {
            try { empty.wait(); }
            catch (InterruptedException e) { }
        }
    }
    obj = o;  ready = true;
    synchronized (full) {
        full.notify();
    }
}

Use two objects, empty and full, to allow two different wait sets

class synchronized Object consume() {
    while (!ready) {
        synchronized (full) {
            try { full.wait(); }
            catch (InterruptedException e) { }
        }
    }
    Object o = obj;  ready = false;
    synchronized(empty){
        empty.notify();
    }
    return obj;
}

notify() vs. notifyAll()

• Very tricky to use notify() correctly
  – notifyAll() generally much safer
• To use correctly, should have:
  – all waiters be equal
  – each notify only needs to wake up 1 thread
  – handle InterruptedException correctly