

## Threads

- A thread is a flow of control in a program.
- The Java Virtual Machine allows an application to have multiple threads of execution running concurrently.
- When a Java Virtual Machine starts up, there is usually a single thread (which typically calls the method named main of some designated class).
- Threads are given priorities. A high priority thread has preference over a low priority thread.

1/4/01

ICSS235 - Threads

1

---

---

---

---

---

---

---

---

## Understanding Threads

- You must be able to answer the following questions
  - What code does a thread execute?
  - What states can a thread be in?
  - How does a thread change its state?
  - How does synchronization work?

1/4/01

ICSS235 - Threads

2

---

---

---

---

---

---

---

---

## Thread Objects

- As is everything else, threads in Java are represented as objects.
- The code that a thread executes is contained in its `run()` method.
  - There is nothing special about run, anyone can call it.
- To make a thread eligible for running you call its `start()` method

1/4/01

ICSS235 - Threads

3

---

---

---

---

---

---

---

---

## Example

```
public class CounterThread extends Thread {
    public void run() {
        for ( int i=0; i<10; i++)
            System.out.println("Count:  " + i);
    }

    public static void main(String args[]) {
        CounterThread ct = new CounterThread();
        ct.start();
    }
}
```

1/4/01

ICSS235 - Threads

4

---

---

---

---

---

---

---

---

## Interface Runnable

- Classes that implement `Runnable` can also be run as separate threads
- `Runnable` classes have a `run()` method
- In this case you create a thread specifying the `Runnable` object as the constructor argument

1/4/01

ICSS235 - Threads

5

---

---

---

---

---

---

---

---

## Example

```
public class DownCounter implements Runnable {
    public void run() {
        for (int i=10; i>0; i--)
            System.out.println("Down:  "+ i);
    }

    public static void main(String args[]) {
        DownCounter ct = new DownCounter();
        Thread t = new Thread(ct);

        t.start();
    }
}
```

1/4/01

ICSS235 - Threads

6

---

---

---

---

---

---

---

---

## Many

```
public class Many extends Thread {
    private int retry; private String info;

    public Many (int retry, String info) {
        this.retry = retry; this.info = info;
    }

    public void run () {
        for (int n = 0; n < retry; ++ n) work();
        quit();
    }

    protected void work () { System.out.print(info); }
    protected void quit () { System.out.print('\n'); }

    public static void main (String args []) {
        if (args != null)
            for (int n = 0; n < args.length; ++n)
                new Many(args.length, args[n]).start();
    }
}
```

1/4/01

ICSS235 - Threads

7

---

---

---

---

---

---

---

---

## When Execution Ends

- The Java Virtual Machine continues to execute threads until either of the following occurs:
  - The exit method of class `Runtime` has been called
  - All threads that are not *daemon* threads have died, either by returning from the call to the `run()` or by throwing an exception that propagates beyond `run()`.
- You cannot restart a dead thread, but you can access its state and behavior.

1/4/01

ICSS235 - Threads

8

---

---

---

---

---

---

---

---

## Thread Scheduling

- Threads are scheduled like processes
- Thread states
  - Running
  - Waiting, Sleeping, Suspended, Blocked
  - Ready
  - Dead
- When you invoke `start()` the Thread is marked ready and placed in the thread queue

1/4/01

ICSS235 - Threads

9

---

---

---

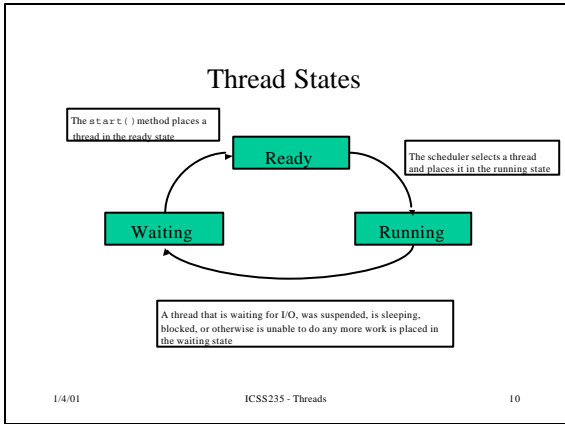
---

---

---

---

---




---

---

---

---

---

---

---

---

---

---

- ### Scheduling Implementations
- Scheduling is typically either:
    - non-preemptive
    - preemptive
  - Most Java implementations use preemptive scheduling.
    - the type of scheduler will depend on the JVM that you use.
    - In a non-preemptive scheduler a thread leaves the running state only when it is ready to do so.
- 1/4/01 ICSS235 - Threads 11

---

---

---

---

---

---

---

---

---

---

- ### Thread Priorities
- Threads can have priorities from 1 to 10 (10 is the highest)
  - The default priority is 5
    - The constants `Thread.MAX_PRIORITY`, `Thread.MIN_PRIORITY`, and `Thread.NORM_PRIORITY` give the actual values
  - Priorities can be changed via `setPriority()` (there is also a `getPriority()`)
- 1/4/01 ICSS235 - Threads 12

---

---

---

---

---

---

---

---

---

---

## isAlive()

- The method `isAlive()` determines if a thread is considered to be alive
  - A thread is alive if it has been started and has not yet died.
- This method can be used to determine if a thread has actually been started and has not yet terminated

1/4/01

ICSS235 - Threads

13

---

---

---

---

---

---

---

---

## isAlive()

```
public class WorkerThread extends Thread {
    private int result = 0;

    public void run() {
        // Perform a complicated time consuming calculation
        // and store the answer in the variable result
    }

    public static void main(String args[]) {
        WorkerThread t = new WorkerThread();
        t.start();

        while ( t.isAlive() ); // What is wrong with this?
        System.out.println( result );
    }
}
```

1/4/01

ICSS235 - Threads

14

---

---

---

---

---

---

---

---

## sleep()

- Puts the currently executing thread to sleep for the specified number of milliseconds
  - `sleep(int milliseconds)`
  - `sleep(int millisecs , int nanosecs)`
- Sleep can throw an `InterruptedException`
- The method is static and can be accessed through the `Thread` class name

1/4/01

ICSS235 - Threads

15

---

---

---

---

---

---

---

---

## sleep()

```
public class WorkerThread extends Thread {
    private int result = 0;

    public void run() {
        // Perform a complicated time consuming calculation
        // and store the answer in the variable result
    }

    public static void main(String args[]) {
        WorkerThread t = new WorkerThread();
        t.start();

        while ( t.isAlive() )
            try {
                sleep( 100 );
            } catch ( InterruptedException ex ) {}

        System.out.println( result );
    }
}
```

1/4/01

ICSS235 - Threads

16

---

---

---

---

---

---

---

---

## Timer

```
import java.util.Date;

class Timer implements Runnable {
    public void run() {
        while ( true ) {
            System.out.println( new Date() );

            try {
                Thread.currentThread().sleep(1000);
            }
            catch ( InterruptedException e ) {}
        }

        public static void main( String args[] ) {
            Thread t = new Thread( new Timer() );

            t.start();
            System.out.println( "Main done" );
        }
    }
}
```

1/4/01

ICSS235 - Threads

17

---

---

---

---

---

---

---

---

## yield()

- A call to the `yield()` method causes the currently executing thread to go to the ready state (this is done by the thread itself)

1/4/01

ICSS235 - Threads

18

---

---

---

---

---

---

---

---

## yield()

```
public class WorkerThread extends Thread {
    private int result = 0;

    public void run() {
        // Perform a complicated time consuming calculation
        // and store the answer in the variable result
    }

    public static void main(String args[]) {
        WorkerThread t = new WorkerThread();
        t.start();

        while ( t.isAlive() )
            yield();

        System.out.println( result );
    }
}
```

1/4/01

ICSS235 - Threads

19

---

---

---

---

---

---

---

---

## Joining Threads

- Calling `isAlive()` to determine when a thread has terminated is probably not the best way to accomplish this
- What would be better is to have a method that once invoked would wait until a specified thread has terminated
- `join()` does exactly that
  - `join()`
  - `join(long timeout)`
  - `join(long timeout, int nanos)`
- Like `sleep()`, `join()` is static and can throw an `InterruptedException`

1/4/01

ICSS235 - Threads

20

---

---

---

---

---

---

---

---

## join()

```
public class WorkerThread extends Thread {
    private int result = 0;

    public void run() {
        // Perform a complicated time consuming calculation
        // and store the answer in the variable result
    }

    public static void main(String args[]) {
        WorkerThread t = new WorkerThread();
        t.start();

        try {
            t.join();
        } catch ( InterruptedException ex ) {}

        System.out.println( result );
    }
}
```

1/4/01

ICSS235 - Threads

21

---

---

---

---

---

---

---

---

## Problems!!

```
import java.util.*;

public class Sync extends Thread {
    private static int common = 0;
    private int id;

    public Sync( int id ) { this.id = id; }

    public void run() {
        for ( int i = 0; i < 10; i++ ) {
            int tmp = common; tmp = tmp + 1;

            try {
                Thread.currentThread().sleep( 10 );
            } catch ( InterruptedException e ) {};

            common = tmp;
        }
    }
}
```

1/4/01

ICSS235 - Threads

22

---

---

---

---

---

---

---

---

## Problems!!

```
public static void main( String args[] ) {
    int numThreads = 0;
    try {
        numThreads = Integer.parseInt( args[ 0 ] );
    } catch ( NumberFormatException e ) { System.exit( 1 ); }

    List threads = new ArrayList();
    for ( int i = 0; i < numThreads; i++ ) {
        threads.add( new Sync( i ) );
        ( ( Thread ) threads.get( i ) ).start(); }

    Iterator i = threads.iterator();
    while ( i.hasNext() )
        try {
            ( Thread ) i.next().join();
        } catch( InterruptedException e ) {};

    System.out.println( common );
}
}
```

1/4/01

ICSS235 - Threads

23

---

---

---

---

---

---

---

---