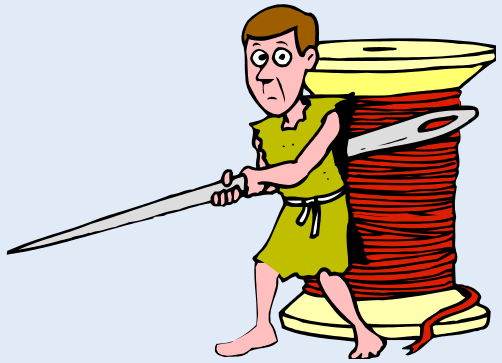
G52APR
Application programming

Java Threads

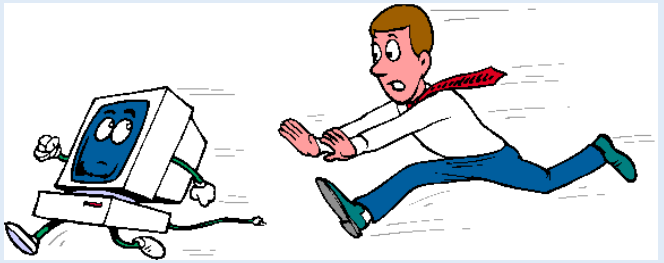
Colin Higgins – based on material from various sources including the Sun tutorial

Threads in Java



- An interface: Runnable
  - Run()
- A class: Thread
  - Start()
  - Stop()
  - Sleep()
  - … …

1. Creating a Thread



- Two ways to create a Thread:
  - 1. Implement `Runnable`, (interface)
    - Implement the `run()` method.
  - 2. Extend `Thread`, (sub-class)
    - Override the `run()` method.
- Due to single inheritance in Java, extending Thread means you cannot extend another class. The first alternative is more used.

Runnable

- Runnable is a very simple interface, it has just one method.
- The Thread's `start()` method calls the Runnable's `run()` method.

```
public interface Runnable {
     public abstract void run();
}
```

Interface - Simple example

```
public class HelloRunnable implements Runnable {

   public void run() {
       System.out.println("Hello from a thread!");
   }

   public static void main(String args[]) {
       HelloRunnable hr = new HelloRunnable();
       (new Thread(hr)).start();

       //or (new Thread(new HelloRunnable())).start();

       // Thread tt= new Thread (new HelloRunnable());
       // tt.start();
   }
}
```

Multiple Threading

- Step 1: create your class (the thread)
- Step 2: implement the run() method
- Step 3: instantiate your class (a Thread type variable)
- Step 4: start the thread by calling start().

Sub-classing (extending)- example

```
public class HelloThread extends Thread {

   public void run() {
       System.out.println("Hello from a thread!");
   }

   public static void main(String args[]) {
       (new HelloThread()).start();
   }

}
```

2. Implementing Run()

```
class MyThread implements Runnable {
  MyThread() {
  }
  public void run() {
    // Does nothing?
    // Loop forever...
  }
 public static void main(String[] args) {
  Thread mythread = new Thread(new MyThread());
     mythread.start();
     myThread.stop();
   }
}
```

3. Executing Threads

- **`run()`**
  - Run it! (Runnable)
- **start()**
  - Activates the thread and calls `run()`.
- **`stop()`**
  - Forces the thread to stop.
- **`suspend()`**
  - Temporarily halt the thread.
- **`resume()`**
  - Resume a halted thread.
- **`destroy()`**

  - equivalent to UNIX's "kill -9"

- **`isAlive()`**
  - Is it running?
- **`yield()`**
  - Let another thread run.
- **`join()`**
  - Wait for the death of a thread.
- **`sleep(long)`**
  - Sleep for a number of milliseconds.
- **`interrupt()`**
  - Interrupt sleep,wake up!

Sleep Example

```
public class SleepMessages {
   public static void main(String args[]) throws
   InterruptedException {
       String info[] = {
          ”Info one”,
          ”Info two”
       };
       for (int i = 0; i < info.length; i++)
          Thread.sleep(4000);
          System.out.println(info[i]);
       }
   }
}
```

Threads race

```
public class ThreadExample implements Runnable {
    public void run() {
        for (int i = 0; i < 3; i++)
         System.out.println(i);
    }
    public static void main(String[] args) {
      new Thread(new ThreadExample()).start();
      new Thread(new ThreadExample()).start();
      System.out.println("Done");
    }
}
```

Java Thread Example – Output

- Possible outputs
  - 0,1,2,0,1,2,Done // thread 1, thread 2, main()
  - 0,1,2,Done,0,1,2 // thread 1, main(), thread 2
  - Done,0,1,2,0,1,2 // main(), thread 1, thread 2
  - 0,0,1,1,2,Done,2 // main() & threads interleaved

main (): thread 1, thread 2, println Done

thread 1: println 0, println 1, println 2

thread 2: println 0, println 1, println 2

Daemon Thread

- Why doesn't the program quit as soon as Done is printed?
- Java threads types
  - User
  - Daemon
    - Provide general services
    - Typically never terminate
    - Call setDaemon() before start()
- Program termination
  - If all non-daemon threads terminate, JVM shuts down

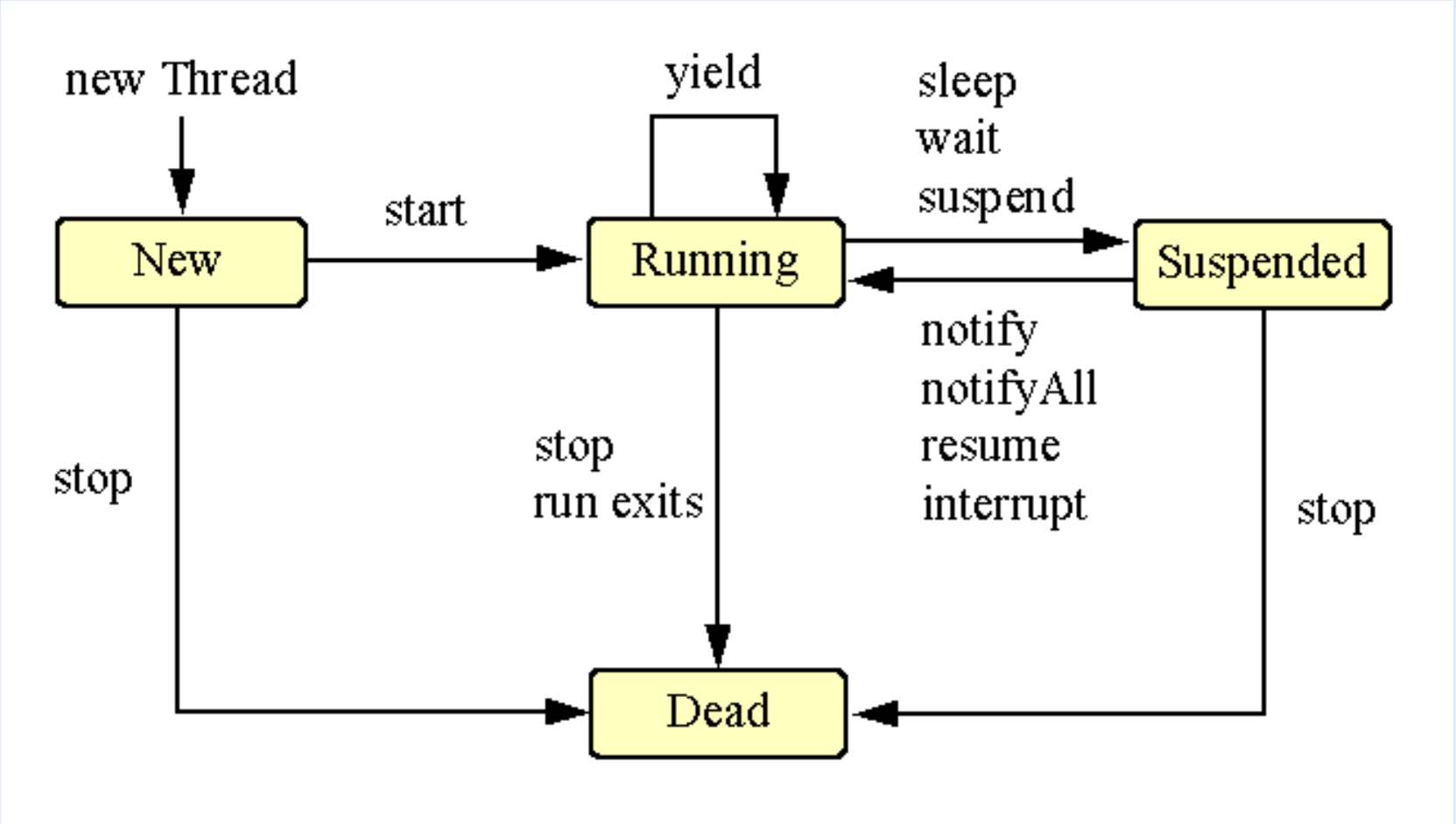
JVM

- When running a JAVA program, how many processes are activated in memory?
  - Only one, JVM!!!
- When running a JAVA program, how many threads are active in memory?
  - It depends on JVM implementation and the functions of JVM used by the customer‘s program.
  - The customer‘s program can create threads

InterThread Communication

- Inter-thread communication methods are declared in `java.lang.Object`.
- Each object could be associated with a monitor (a sort of thread lock) – details in concurrency module.
- `wait()`
  - Suspend the thread.
  - Wait can also be time limited.
- `notify()`
  - Unlock the first monitored thread.
  - (The first that called wait() within the monitor.)
- `notifyAll()`
  - Unlocks all monitored threads.
  - Highest prioritised first!

Thread states



The suspended state

- A running Thread becomes suspended when:
  - –it calls `sleep()` to tell the scheduler that it no longer wants to run;
  - –it blocks for I/O; or
  - –it blocks on entry to a `synchronized` object/ method or in `wait()` for condition synchronisation.
  - –It blocks waiting to `join()` another thread (when that thread finishes)

Volatile

- Consider the following code in thread's run()

```
Run(){
   // currentValue is declared in main()
   currentValue=5;
   for (;;) {
       System.out.print(currentValue+",");
       Thread.sleep(1000);
   }
}
```

- Another thread concurrently *increases* the value of currentValue
- Surprisingly, the thread will print 5,5,5,5,5,...

Volatile

- Since the value of currentValue *is not modified in the loop*, the compiler simply continues printing the same value
- Define the variable as ***volatile*** by:

```
volatile int currentValue;
```

- This will cause to perform *actual memory access and value update* for every access to the variable

Thread priorities

- Threads have *priorities* which heuristically influence schedulers:
  - each thread has a priority in the range **Thread.MIN_PRIORITY** to **Thread.MAX_PRIORITY**
  - by default, each new thread has the same priority as the thread that created it---the initial thread associated with a main method by default has priority **Thread.NORM_PRIORITY**
  - the current priority of a thread can be accessed by the method **getPriority** and set via the method **setPriority**.
- When there are more runnable threads than CPUs, a scheduler is generally biased in favour of threads with **higher priorities**.

Thread example

```
public class HelloRunnable implements Runnable {

   public void run() {
       System.out.println("Hello from a thread!");
   }

   public static void main(String args[]) {
       Thread tt1 = new Thread(new HelloRunnable());
       tt1.start();

       Thread tt2 = new Thread(new HelloRunnable());
       tt2.start();
   }

}
```

Thread Interference

```
public class counter {
   private int count = 0;
   public void increment() {
     count++;
   }
   public void decrement() {
     count--;
   }
   public int value() {
     return count;
   }
}
```

// Suppose Thread A invokes increment() at about the same time Thread B invokes decrement().

Thread Interference

- If the initial value of count is 0, their interleaved actions might follow this sequence:

- Thread A: Retrieve count.
- Thread B: Retrieve count.
- Thread A: Increment retrieved value; result is 1.
- Thread B: Decrement retrieved value; result is -1.
- Thread A: Store result in count; count is now 1.
- Thread B: Store result in count; count is now -1.

Thread Interference

- Interference happens when two operations, running in different threads, interleave.
- Scheduling of multiple threads is not guaranteed to be fair.
- If a producer thread and a consumer thread are sharing the same kind of data in a program then either producer may produce the data faster or consumer may retrieve an order of data and process it without its existing.

Synchronize threads

- When two or more threads share the same resource (variable or method), only one of them can access the resource at one time.
- Two approaches to synchronize threads

```
synchronized void myObjectMethod() {
   … …
}

synchronized(Object) {
   … …
}
```

Synchronize threads

- *Synchronized methods*: Any method is specified with the keyword **synchronized** is only executed by one thread at a time.

- *Synchronized statements*: Any object is stated as **synchronized** is accessed by only one thread at any given time.

Synchronized methods

// A synchronized method -> mutual exclusion (lock) per object

// A synchronized static method -> mutual exclusion (lock) per class

```
synchronized void myObjectMethod() {
}

synchronized static void myClassMethod() {
}
```

Synchronized methods

- When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object.
- When a synchronized method exits, it automatically establishes a happens-before relationship with *any subsequent invocation* of a synchronized method for the same object. This guarantees that changes to the state of the object are visible to all threads.

Account Example


```
// This is an example of synchronized methods.
public class Account {
    private double balance;
    public Acount(double initialDeposit) {
      balance = initialDeposit;
    }
    public synchronized double getBalance() {
      return balance;
    }
    public synchronized void deposit(double a){
      balance += a;
    }
  }
```

Synchronized statements

- The synchronized statement:
  - Intrinsic lock: an internal entity associated with an object
  - mutual exclusion (lock) of data access instead of a synchronized method
  - very lightweight

- Syntax:

```
synchronized(Object) {
}
```

Intrinsic Lock and Synchronization

- Every object has an intrinsic lock associated with it.
- As long as a thread owns an intrinsic lock, no other thread can acquire the same lock. The other thread will block when it attempts to acquire the lock.
- When a thread releases an intrinsic lock, a happens-before relationship is established between that action and any subsequent operation of the same lock.
- When a thread invokes a synchronized method, it automatically acquires the intrinsic lock for that method's object.

Synchronized statements

/* Unlike synchronized methods, synchronized statements must specify the object that provides the intrinsic lock: */

```
public void addName(String name) {
   synchronized(this) {
      lastName = name;
      nameCount++;
   }
   nameList.add(name);
}
```

Synchronized statements

// Example (mutual exclusive lock on the variable values):

```
public static void abs(int[] values)
{
   synchronized(values)
   {
      for (int i=0; i < values.length; i++)
      {
        if (values[i] < 0)
          values[i] = -values[i];
      }
   }
}
```

Synchronized Statements

// Here we create two objects solely to provide locks:

```
public class MoreComplex {
   private long c1 = 0;
   private long c2 = 0;
   private Object lock1 = new Object();
   private Object lock2 = new Object();
   public void inc1() {
        synchronized(lock1) {
            c1++;}
   }
   public void inc2() {
        synchronized(lock2) {
            c2++;
} } }
```

Deadlock

```
public class Deadlock {
static class Person {
   private final String name;
   public Person(String name) {
        this.name = name;
   }
   public String getName() {
        return this.name;
   }
   public synchronized void wave(Person p) {
        System.out.format("%s: %s has waved to me!%n", this.name,
            p.getName());
        p.waveBack(this);
   }
   public synchronized void waveBack(Person p) {
        System.out.format("%s: %s has waved back to me!%n",
                    this.name, p.getName());
   }
}
```

```
public static void main(String[] args) {
        final Person alex = new Person(”Alex");
        final Person beth = new Person(”Beth");
        new Thread(new Runnable() {
            public void run()  { alex.wave(beth); }
        }).start();
        new Thread(new Runnable() {
            public void run() { beth.wave(alex); }
        }).start();
    }
}
```

- When Deadlock runs, it's extremely likely that both threads will block when they attempt to invoke waveBack.

Starvation & Livelock

- Starvation:
  - Thread unable to gain access to shared resource.
  - Hence unable to make progress.
  - becaue shared resources are made unavailable for long periods by "greedy" threads
- Livelock:
  - Thread 1 responds to action of thread 2
  - Thread 2 responds to thread 1
  - Repeat forever!

Thread pools

- `java.util.concurrent` package
- A thread pool has *worker threads*. Worker thread exist separately from the tasks it executes and is often used to execute multiple tasks.
- One common type of thread pool is the *fixed thread pool*. This type of pool always has a specified number of threads running; if a thread is somehow terminated while it is still in use, it is automatically replaced with a new thread.

Multiple thread server

```
int portNumber = 8250;
ServerSocket s = new ServerSocket(portNumber);

while(true)    // accept multiple clients
{
   Socket clientSoc = s.accept();
   Thread t = new ThreadedEchoHandler(clientSoc);
   t.start();  // Handle the client communication
}

…

Class ThreadedEchoHandler implements runnable {
// override run()
}
```

Executors

- Objects that create and manage threads.
- Executor interfaces:
  - The Executor interface
  - The ExecutorService interface
  - The ScheduledExecutorService interface

- If r is a Runnable object, and e is an Executor object you can replace

```
(new Thread(r)).start();
```

with

```
e.execute(r);
```

To create a thread pool

public static ExecutorService newFixedThreadPool(int nThreads)

Creates a thread pool that reuses a fixed number of threads operating off a shared unbounded queue. At any point, at most nThreads threads will be active processing tasks. If additional tasks are submitted when all threads are active, they will wait in the queue until a thread is available. If any thread terminates due to a failure during execution prior to shutdown, a new one will take its place if needed to execute subsequent tasks.

Fork/Join

- The fork/join framework is an implementation of the ExecutorService interface that helps you take advantage of multiple processors.
- Class ForkJoinPool
- Usage of ForkJoinPool

```
static final ForkJoinPool mainPool = new ForkJoinPool();
...
public void sort(long[] array) {
     mainPool.invoke(new SortTask(array, 0, array.length));
}
```