Threads

- **Threads** (an abbreviation of threads of control) are how we can get more than one thing to happen at once in a program. A thread has a minimum of internal state and a minimum of allocated resources.

- A thread (a.k.a. lightweight process) is associated with a particular process (a.k.a. heavyweight process, a retronym). A heavyweight process may have several threads and a thread scheduler. The thread scheduler may be in the user or in the system domain.

- Threads share the text, data and heap segments. Each thread has its own stack and status.

Examples where threads are useful: Windowing systems, Web browsers, Servers.

---

*How can you be in two place at once when you’re not anywhere at all?*  
–Firesign Theater.
Threads are part of the Java language. There are two ways to create a new thread of execution.

- One is to extend the class `java.lang.Thread`. This subclass should override the `run` method of class `Thread`. An instance of the subclass can then be allocated and started.
- The other way to create a thread is to declare a class that implements the `Runnable` interface. That class then implements the `run` method. An instance of the class can then be allocated, passed as an argument when creating `Thread`, and started.
In Java, each thread is an object!
Another Thread example in Java

/* threads/RunnableExample.java */
class Grape implements Runnable {
    private String name;
    Grape(String s) {name = s;}
    public String getName() {return name;}
    public void run() {
        for (int i=0; i<10000; i++) {
            System.out.println("This is the " + this.getName() + " thread.");
        }
    }
}

class RunnableExample {
    public static void main (String args[]) {
        Grape g1 = new Grape("merlot");
        Grape g2 = new Grape("pinot");
        Grape g3 = new Grape("cabernet");
        new Thread(g1).start();
        new Thread(g2).start();
        new Thread(g3).start();
    }
}
// See how many threads can be created,
// or, how big a quagmire we can create.
// threads/MaxThreads.java

public class MaxThreads {
    final static int MAX = 50;
    public static void main(String args[])
        throws InterruptedException
    {
        for(int i=0; i<MAX; i++) {
            Integer I = new Integer(i);
            new nuts(I.toString()).start();
        }
        Thread.sleep(20000);
    }
}

class nuts extends Thread {
    nuts(String s) {super(s);} //constructor

    public void run() {
        System.out.println("Thread number "+this.getName());
        System.out.flush();
        this.suspend();
    }
}
Relevant Java Classes/Interfaces

- See the `Object` class for synchronization methods.
- For automatic management of thread pools, see: `Executor` interface from `java.util.concurrent`.
Controlling Threads

- `start()` and `stop()`
- `suspend()` and `resume()`
- `sleep()`.
- `interrupt()`: wake up a thread that is sleeping or blocked on a long I/O operation
- `join()`: causes the caller to block until the thread dies or with an argument (in millisecs) causes a caller to wait to see if a thread has died
public class InterruptTest implements Runnable {

    public static void main(String[] args) throws Exception {
        Thread sleepyThread = new Thread(new InterruptTest());
        sleepyThread.setName("SleepyThread");
        sleepyThread.start();
        // now we two threads running, the main thread and the sleepy thread,
        // which goes to sleep after printing a message.

        Thread.sleep(500); // put main thread to sleep for a while
        sleepyThread.interrupt(); // interrupt sleepyThread’s beauty sleep
        Thread.sleep(500); // put main thread to sleep for a while
        sleepyThread.interrupt(); // interrupt sleepyThread’s beauty sleep
    }

    public void run() {
        Thread me = Thread.currentThread();
        while (true) {
            try {
                System.out.println(me.getName() + "": sleeping...");
                Thread.sleep(5*1000); // in millisecs
            } catch (InterruptedException e) {
                System.out.println(me.getName() +": argh! let me sleep #$@!");
            }
        }
    }
}
A thread continues to execute until one of the following things happens:

- it returns from its target `run()` method.
- it’s interrupted by an uncaught exception.
- it’s `stop()` method is called.

What happens if the `run()` method never terminates, and the application that started the thread never calls the `stop()` method? *The thread remains alive even after the application has finished!* (so the Java interpreter has to keep on running...)
Daemon Threads

- Useful for simple, periodic tasks in an application.
- The `setDaemon()` method marks a thread as a daemon thread that should be killed and discarded when no other application threads remain.

```java
class Devil extends Thread {
    Devil() {
        setDaemon(true);
        start();
    }
    public void run() {
        //perform evil tasks
        ...
    }
}
```
Java threads are preemptible. Java threads may or may not be time-sliced. The programmer should not make any timing assumptions.

Threads have priorities that can be changed (increased or decreased). An application cannot usurp resources from another application since all threads operate within one process.

This implies that multiple threads will have race conditions (read/write conflicts based on time of access) when they run. The programmer has to resolve these conflicts.

Example of a race condition: Account.java, TestAccount.java
Java has `synchronized` keyword for guaranteeing mutually exclusive access to a method or a block of code. Only one thread can be active among all synchronized methods and synchronized blocks of code in a class.

```java
// Only one thread can execute the update method at a time in the class.
synchronized void update() { //... }
```

Access to individual datum can also be synchronized.

```java
// The object buffer can be used in several classes, implying // synchronization among methods from different classes.

synchronized(buffer) {
    this.value = buffer.getValue();
    this.count = buffer.length();
}
```

Every Java object has an implicit monitor associated with it to implement the synchronized keyword. Inner class has a separate monitor than the containing outer class.

Java allows **Rentrant Synchronization**, that is, a thread can reacquire a lock it already owns. For example, a synchronized method can call another synchronized method.
Synchronization Example 1

- Race conditions: `Account.java`, `TestAccount.java`
- Thread safe version using `synchronized` keyword: `RentrantAccount.java`
The `wait()` and `notify()` methods (of the `Object` class) allow a thread to give up its hold on a lock at an arbitrary point, and then wait for another thread to give it back before continuing.

Another thread must call `notify()` for the waiting thread to wakeup. If there are other threads around, then there is no guarantee that the waiting thread gets the lock next. *Starvation* is a possibility. We can use an overloaded version of `wait()` that has a timeout.

The method `notifyAll()` wakes up all waiting threads instead of just one waiting thread.
class MyThing {
    synchronized void waiterMethod() {
        // do something
        // now we need to wait for the notifier to do something
        // this gives up the lock and puts the calling thread to sleep
        wait();
        // continue where we left off
    }

    synchronized void notifierMethod() {
        // do something
        // notifier the waiter that we’ve done it
        notify();
        // do more things
    }

    synchronized void relatedMethod() {
        // do some related stuff
    }
}
Synchronization Example 2: Producer/Consumer Problem

- A **producer** thread creates messages and places them into a queue, while a **consumer** thread reads them out and displays them.
- The queue has a maximum depth.
- The producer and consumer don’t operate at the same speed. In the example, the producer creates messages every second but the consumer reads and displays only every two seconds. **How long will it take for the queue to fill up? What will happen when it does?**
Producer and Consumer sharing a synchronized queue.

See examples: threads/SharedQueue.java, threads/Producer.java, threads/Consumer.java, threads/PC.java
Thread Groups

- Hierarchical collection of threads.
  ```java
  ThreadGroup myTaskGroup = new ThreadGroup("My Task Group");
  Thread myTask = new Thread(myTaskGroup, taskPerformer);
  ```

- The ThreadGroups `stop()`, `suspend()`, `resume()` operate on all the threads in the group. A thread group can also be marked as a daemon. Otherwise we have to use `destroy()` to remove the thread group.

- The methods `activeCount()` tells us how many threads are in the thread group; the method `enumerate()` gives us a list of the threads. The argument to enumerate is an array of threads.

- Thread groups can contain other thread groups recursively. In that case the methods apply recursively.
ThreadGroup example

// threads/ThreadGroupExample.java
public class ThreadGroupExample {
    public static void main( String args[] ) {
        ThreadGroup myTaskGroup = new ThreadGroup("My Task Group");

        Thread thread1 = new Thread(myTaskGroup, new MyThread(), "one");
        Thread thread2 = new Thread(myTaskGroup, new MyThread(), "two");
        Thread thread3 = new Thread(myTaskGroup, new MyThread(), "three");

        thread1.start();
        thread2.start();
        thread3.start();

        System.out.println("group has "+ myTaskGroup.activeCount()+ " threads");

        Thread[] tasks = new Thread[myTaskGroup.activeCount()];
        myTaskGroup.enumerate(tasks);
        for (int i=0; i<tasks.length; i++)
            System.out.println(tasks[i].toString());

        System.out.println(myTaskGroup.toString());

        myTaskGroup.stop(); //stop all threads in the group
    }
}

class MyThread implements Runnable {
    public void run() {
        while (true) {
            try {Thread.sleep(10000);} catch (InterruptedException e) { System.err.println(e);} 

        }
    }
}
Many of the data structures in `java.util` package aren't synchronized. We can convert them into synchronized versions using wrappers.

The synchronization wrappers add automatic synchronization (thread-safety) to an arbitrary collection. Each of the six core collection interfaces – Collection, Set, List, Map, SortedSet, and SortedMap – has one static factory method.

```java
public static <T> Collection<T> synchronizedCollection(Collection<T> c);
public static <T> Set<T> synchronizedSet(Set<T> s);
public static <T> List<T> synchronizedList(List<T> list);
public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m);
public static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s);
public static <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> m);
```

Create the synchronized collection with the following trick.

```java
List<Type> list = Collections.synchronizedList(new ArrayList<Type>());
```

A collection created in this fashion is every bit as thread-safe as a normally synchronized collection, such as a Vector.
In the face of concurrent access, it is imperative that the user manually synchronize on the returned collection when iterating over it. The reason is that iteration is accomplished via multiple calls into the collection, which must be composed into a single atomic operation.

```java
Collection<Type> c = Collections.synchronizedCollection(myCollection);
synchronized(c) {
    for (Type e : c)
        process(e);
}
```

If an explicit iterator is used, the iterator method must be called from within the synchronized block. Failure to follow this advice may result in nondeterministic behavior.

One minor downside of using wrapper implementations is that you do not have the ability to execute any noninterface operations of a wrapped implementation.

For more details, see:
http://docs.oracle.com/javase/tutorial/collections/implementations/wrapper.html
References

- Brian Goetz, Tim Peierls, Joshua Bloch and Joseph Bowbeer: *Java Concurrency in Practice*
- Lewis and Berg: *Multithreaded Programming with Java Technology*