Threads

- Thread = Execution of one sequence of instructions (including function/method calls, conditionals, loops).
- Normal Java program executes in a thread created for main (application) or borrowed from the browser (applets).
- Class Thread can be used to create additional threads that execute concurrently.
- Each new thread is associated with (controlled by) a Thread object.

Single Thread Example

```java
class Foo {
    void run() {
        for (int i=0; i<100; i++)
            System.out.println("foo ");
    }
}
class Bar {
    public static void main(char[]args) {
        Foo foo = new Foo();
        foo.run();
        for (int i=0; i<100; i++)
            System.out.println("bar ");
    }
}  
```

- Prints 100 "foo"s followed by 100 "bar"s

Concurrent Execution

- To begin concurrent execution, call method start of a Thread object. This sets up the new thread, then calls the object's run method.

```java
class Bar {
    public static void main(char[]args) {
        Foo foo = new Foo();
        foo.start();
        for (int i=0; i<100; i++)
            System.out.println("bar ");
    }
}  
```

- Prints 100 “foo”s and 100 “bar”s in some unpredictable order

Extending Class Thread

- Class Thread can be extended to create objects that run concurrently in their own thread.
- Execution begins in method run of the new class.

```java
class Foo extends Thread {
    void run() {
        for (int i=0; i<100; i++)
            System.out.println("foo ");
    }
}
```

- Foo.run overrides a (basically) empty method run in class Thread.

Uses for Threads

- Asynchronous or nonblocking I/O
  - Continue execution in one thread while waiting for I/O to complete or time out in another.
- Timers
  - Wait for an interval to expire, then cause something to happen (examples: animations; do something if the user doesn't respond after a reasonable interval, ...)
- Process multiple tasks simultaneously
  - Handle GUI in one thread while doing extended calculations in another.
- Parallel algorithms
  - If the JVM supports it, run parts of the computation concurrently on different processors.
Runnable Classes

- There are many situations where we want to execute a computation concurrently, but in a class that’s not a subclass of Thread.
- We still need a Thread object to create and control the thread.
- A thread can begin execution in any class that implements Runnable and contains a run method.

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

Using Runnable (cont.)

  ```java
  public void run() {
      foo();
  }
  public static void main(String[] args) {
      FooBar fb = new FooBar();
      Thread t = new Thread(fb);
      t.start();
  }
  ```

Synchronization

- Since threads may interleave execution in any order, we may need to control access to objects to ensure only one thread at a time can update related variables.

  ```java
class C {
    int x, y;
    public synchronized void setXY(int x, int y) {
      this.x = x; this.y = y;
    }
    public int sumXY() { return x + y; }
  }
  ```

- What happens if one thread executes `sumXY` while another thread is halfway through executing `setXY` on the same object?

  ```java
  synchronized methods
  ```

- Every object has an associated lock
- We can require threads to acquire the lock before executing one of the object’s methods by declaring the method to be synchronized.
- A synchronized method automatically acquires the object’s lock when it is called. Other threads are blocked until the lock is released automatically when the synchronized method terminates.

  ```java
  class C {
    int x, y;
    public synchronized void setXY(int x, int y) {
      this.x = x; this.y = y;
    }
    public synchronized int sumXY() { return x + y; }
  }
  ```