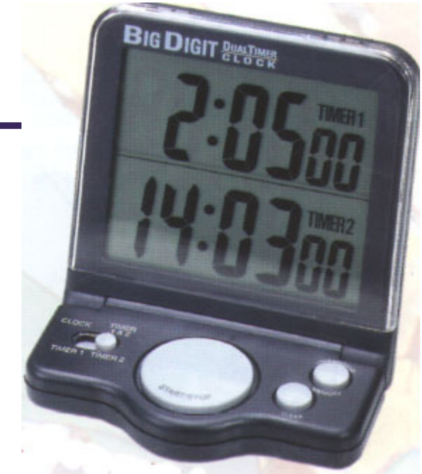
CSE 331

Timers, Threads, and Concurrency

slides created by Marty Stepp
based on materials by M. Ernst, S. Reges, D. Notkin, R. Mercer, Wikipedia

http://www.cs.washington.edu/331/

Timers



- **timer**: An object that executes an action repeatedly at given intervals.

- Used to:
  - create animations in GUI programs.
  - add delays and pauses when required / desired.
  - update the appearance of a UI periodically.

- A Timer is an example of a "callback." Your code starts the timer, then later at a specified time, the timer activates, causing an event in your system.

The Timer class

```
import javax.swing.Timer;
```

- `public Timer(int msDelay, ActionListener listener)`
  Constructs a timer to fire an action event every *msDelay* ms.

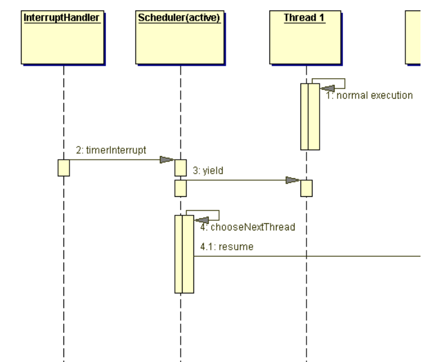
method	description
`start``()`	starts timer to generate events
`stop``()`	stops timer so that no more events will occur
`restart``()`	resumes a stopped timer, waiting its initial delay and then firing events at the usual rate
`addActionListener(`**AL**`)`	attaches an additional listener to the timer
`isRunning()`	returns `true` if the timer has been started
`setInitialDelay(`**ms**`)`	sets initial delay before first event, which can be different from delay between events
`setRepeats(`**boolean**`)`	set to `false` to cause a one-time event

Timer example

```
// This code might move a shape across the screen.
private int DELAY = 100;
...
public class Painter implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        x += 5;
        myPanel.repaint();
    }
}
...
Timer tim = new Timer(DELAY, new Painter());
tim.start();
```

Processes and threads

- **process**: A program running on the computer.
  - Processes have *memory isolation* (don't share data with each other).
- **thread**: A "lightweight process"; a single sequential flow of execution or isolated sub-task within one program.
  - A means to implement programs that seem to perform multiple tasks simultaneously (a.k.a. *concurrency*).
  - Threads within the same process *do* share data with each other.
    - i.e., Variables created in one thread can be seen by others.
    - "*shared-memory concurrency*"
  - sometimes called a *lightweight process*

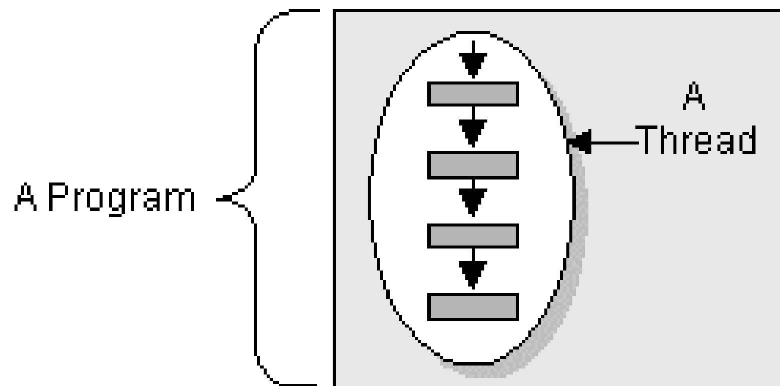


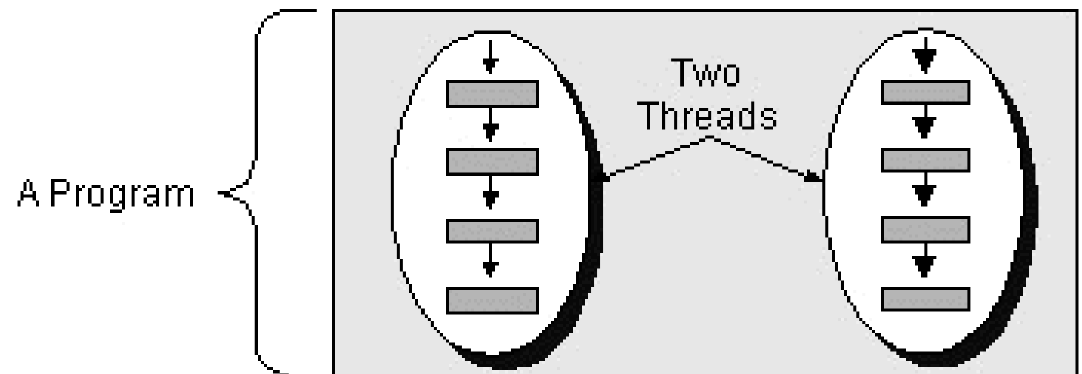
Places threads are used

- I/O
  - loading a file in the background
- Networking
  - example: thread that waits for another machine to connect
- Graphics and animation
  - `Timer` class is sometimes preferred for this (seen later)
- event-handling loops
  - largely handled for us by Java
- parallelized algorithms
  - example: multithreaded merge sort

A multithreaded program

- **1 thread:**
  - program executes sequentially
  - every program has a "main thread" for its `main` method
- **2 or more threads:**
  - runs each thread sequentially, but interleaves them
  - overall program is concurrent



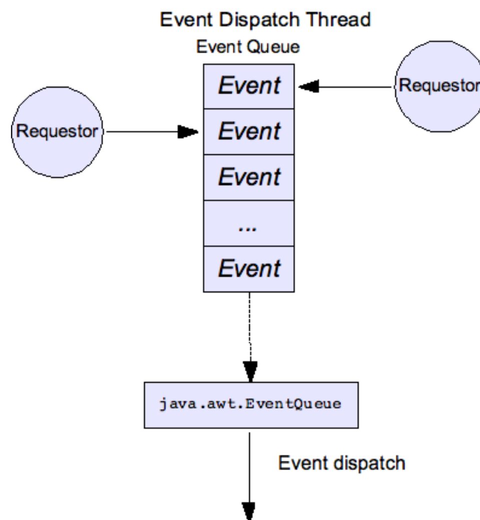


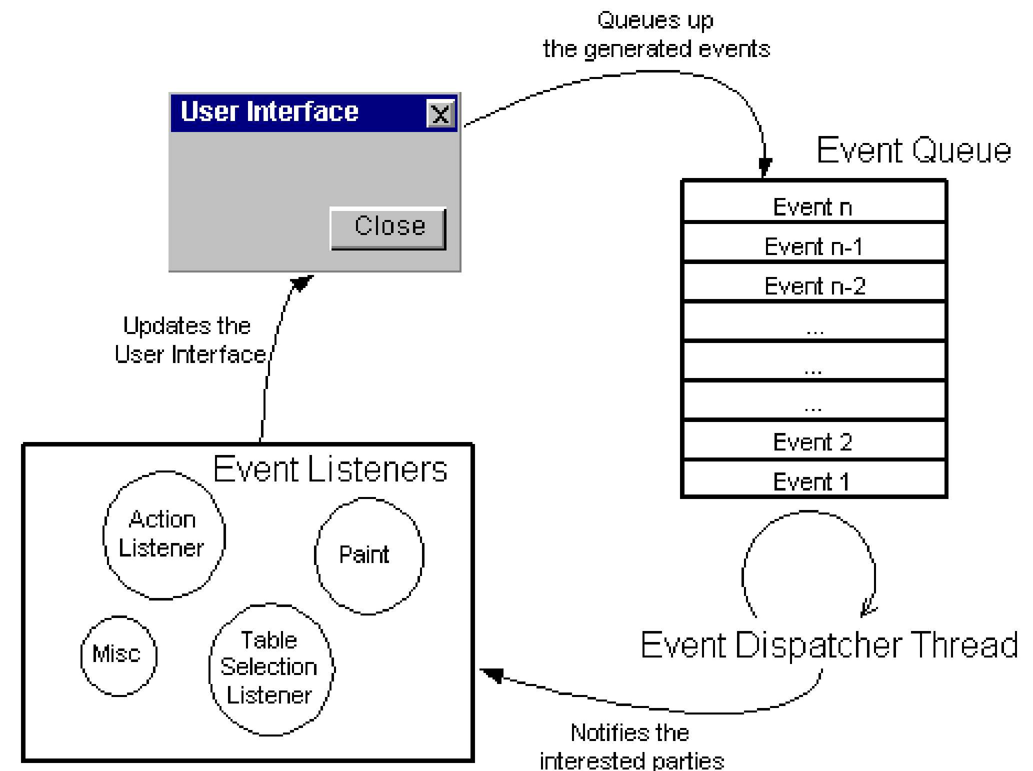
- **pre-emptive scheduling**: OS lets each thread/process run for a short *time slice* then switches to another.
  - High *priority* threads run first.

AWT/Swing event thread

- The first time your program creates an AWT or Swing component, it causes an *event dispatcher thread* to be started.
  - The event thread is the thread in which all event listeners (ActionListener, MouseListener, etc.) execute when events occur.
  - Listeners should be short so they don't block each other or the GUI.

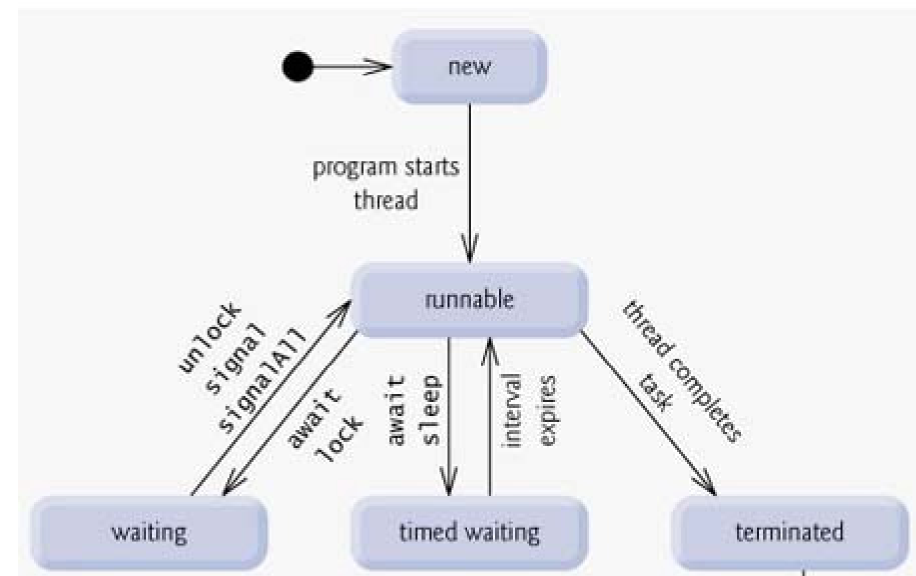
- **daemon thread**: One that runs solely to aid other threads.



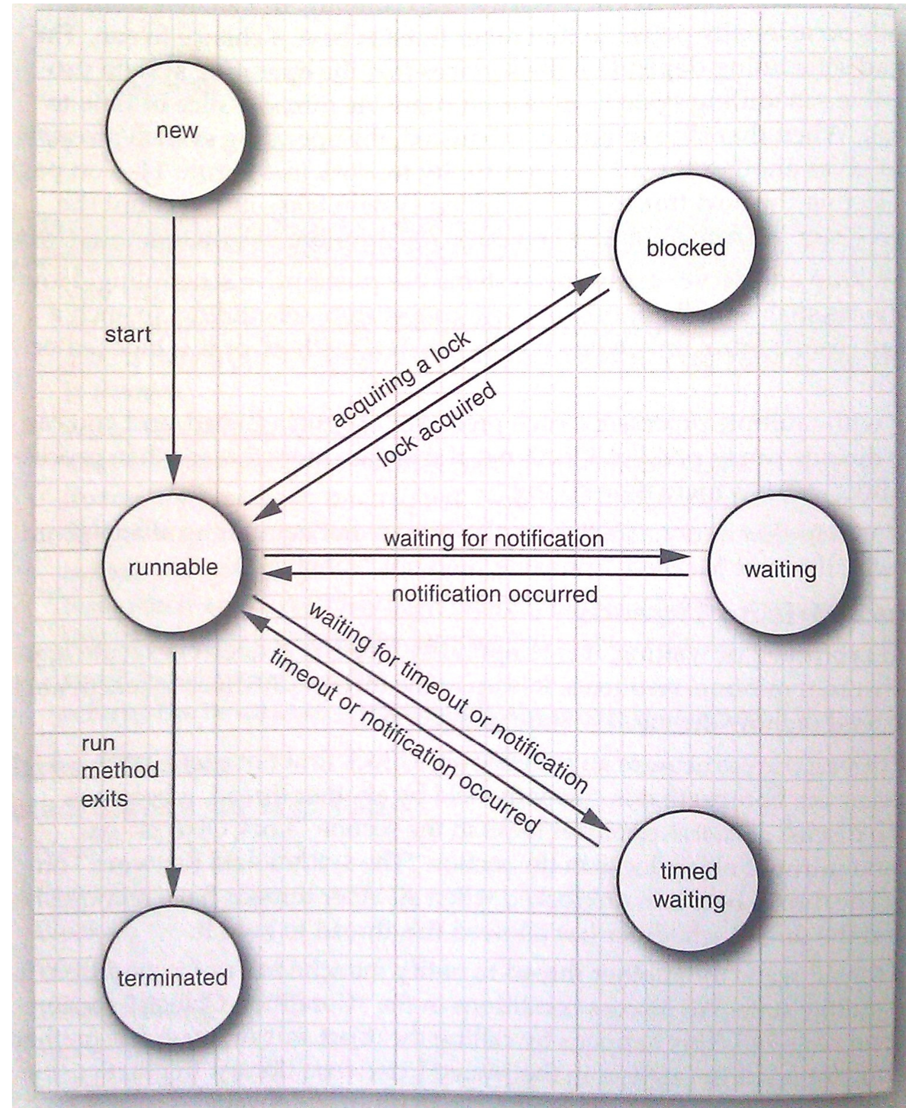


Thread states

- **new**: Just constructed; hasn't started to run yet.
- **runnable**: Running, or enqueued to be run.
- **blocked** or **waiting**: Asleep, not running; waiting for some signal from elsewhere in the program to resume.
- **timed waiting**: Waiting a given exact amount of time to resume.
- **terminated**: Done running.
  - see `Thread`'s `getState()`



Thread state diagram



(*Core Java* Figure 14-3)

Thread object methods

Method name	Description
`getPriority()` `setPriority(`**int**`)`	gets/sets this thread's running priority. Possible values: Thread.MIN_PRIORITY, NORM_PRIORITY, MAX_PRIORITY
`getName()` `setName(`**name**`)`	gets/sets the name of this thread as a string
`getState()`	thread's state. One of Thread.State.NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, or TERMINATED
`interrupt()`	stops the thread's current time slice
`isAlive()`	returns `true` if the thread is in runnable state
`join()` **`join(ms)`**	waits indefinitely, or for a given number of milliseconds, for the thread to finish running
`start()`	puts a thread into runnable state
`stop()`	instructs a thread to stop immediately (*deprecated*)

Thread static methods

Static method name	Description
`activeCount()`	number of currently runnable/active threads
`dumpStack()`	causes current thread to print a stack trace
`getAllStackTraces()`	returns stack trace data for all currently running threads
`getCurrentThread()`	returns the current code's active thread
`holdsLock(`**obj**`)`	returns `true` if current thread has locked the given object
`interrupted()`	returns `true` if current thread is in interrupted state
`setDefaultUncaughtExceptionHandler(`**handler**`)`	redefines what to do if an exception is not caught and reaches all the way up the stack to the top level
`sleep(ms)`	causes the current thread to wait for at least the given number of ms before continuing
`yield()`	temporarily pauses the current thread to let others run

The Runnable class

- To cause your own code to run in its own thread, write a class that implements the `Runnable` interface.
  - Then construct a new `Thread`, passing your runnable object, and start the thread.
  - ```
    public Thread(Runnable runnable)
    ```
  - ```
    public interface Runnable {
        public void run();
    }
    ```

Runnable example

```
public class MyRunnable implements Runnable {
    public void run() {
        // perform a task...
    }
}

...
Thread t = new Thread(new MyRunnable());
t.start();
```

- Sometimes done with an *anonymous inner class*:

```
new Thread(new Runnable() {
    public void run() {
        // perform a task...
    }
}).start();
```

Problem: Infinite loops

- Suppose we want to write a testing system that runs methods written by students and verifies their behavior.
  - But some students write poor code that goes into an infinite loop:

```
// run code that might have an infinite loop
// (student might have a bug in his code)
StudentCode student = new StudentCode();
student.doStuff();  // might hang

// if doStuff hangs, this will never be reached
System.out.println("Completed.");
```

  - The student's bug will hang our entire testing system! What to do?

Waiting for a thread

```
// run code that might have an infinite loop
StudentCode student = new StudentCode();
...
Thread t = new Thread(new Runnable() {
    public void run() {
        student.doStuff();  // might hang
    }
});

// run code in thread and give up after 10 sec
t.start();
try {
    t.join(10000);
} catch (InterruptedException ie) {}
if (t.isAlive()) {
    System.out.println("Timed out!");
} else {
    System.out.println("Completed.");
}
```

Sleeping a thread

```
try {
    Thread.sleep(ms);
} catch (InterruptedException ie) {}
```

- Causes current thread to wait for the given number of milliseconds.
- If the program has other threads, they will be given a chance to run.
- Useful for writing code that checks for an update periodically.

```
// check for new network messages every 2 sec
while (!done) {
    try {
        Thread.sleep(2000);
    } catch (InterruptedException ie) {}
    myMessageQueue.read();
    ...
}
```

Thread safety

- **thread safe**: Can be used in a multithreaded environment.
  - Many of the Java library classes are *not* thread safe!
  - In other words, if two threads access the same object, things break.

- Examples:
  - AWT and Swing are not thread safe; if two threads are modifying a GUI simultaneously, they may put the GUI into an invalid state.
  - `ArrayList` and other collections from `java.util` are not thread safe; two threads changing the same list at once may break it.
  - `StringBuilder` is not thread safe.

- Counterexamples:
  - The `Random` class chooses numbers in a thread-safe way.

Unsafe code

- How can the following class be broken by multiple threads?

```
 1 public class Counter {
 2   private int c = 0;

 3   public void increment() {
 4     int old = c;
 5     c = old + 1;   // c++;
 6   }

 7   public void decrement() {
 8     int old = c;
 9     c = old - 1;   // c--;
10   }

11   public int value() {
12     return c;
13   }
14 }
```

Scenario that breaks it:

- Threads A and B start.
- A calls `increment` and runs to the end of line 4. It retrieves the `old` value of 0.
- B calls `decrement` and runs to the end of line 8. It retrieves the `old` value of 0.
- A sets c to its `old` (0) + 1.
- B sets c to its `old` (0) - 1.
- The final `value()` is -1, though after one increment and one decrement, it should be 0!

Object locks

- *Every* Java object has a built-in internal "lock".
  - A thread can "wait" on an object's lock, causing it to pause.
  - Another thread can "notify" on an object's lock, unpausing any other thread(s) that are currently waiting on that lock.
  - An implementation of *monitors*, a classic concurrency construct.

method	description
`notify()`	unblocks one random thread waiting on this object's lock
`notifyAll()`	unblocks all threads waiting on this object's lock
`wait()` `wait(ms)`	causes the current thread to wait (block) on this object's lock, indefinitely or for a given # of ms

  - These methods are not often used directly; but they are used internally by other concurrency constructs (see next slide).

The synchronized keyword

```
// synchronized method: uses "this" object's lock
public synchronized type name(parameters) {
    statement(s);
}

// synchronized static method: uses the given class's lock
public static synchronized type name(parameters) {
    statement(s);
}

// synchronized block: uses the given object's lock
synchronized (object) {
    statement(s);
}
```

- The keyword `synchronized` causes a given method or block of code to acquire an object's lock before running.
  - Ensures that only one thread can be in the given block at a time.

The volatile keyword

```
private volatile type name;
```

- **volatile field**: An indication to the VM that multiple threads may try to access/update the field's value at the same time.
  - Causes Java to immediately flush any internal caches any time the field's value changes, so that later threads that try to read the value will always see the new value (never the stale old value).
  - Allows limited safe concurrent access to a field inside an object even if another thread may modify the field's value.
  - Does not solve all concurrency issues; should be replaced by `synchronized` blocks if more complex access is needed.

Synchronized counter

```
public class Counter {
    private volatile int c = 0;

    public synchronized void increment() {
        int old = c;
        c = old + 1;  // c++;
    }

    public synchronized void decrement() {
        int old = c;
        c = old - 1;  // c--;
    }

    public int value() {
        return c;
    }
}
```

- Should the `value` method be synchronized? Why/why not?

Deadlock

- **liveness**: Ability for a multithreaded program to run promptly.
- **deadlock**: Situation where two or more threads are blocked forever, waiting for each other.
  - Example: Each is waiting for the other's locked resource.
  - Example: Each has too large of a `synchronized` block.
- **livelock**: Situation where two or more threads are caught in an infinite cycle of responding to each other.
- **starvation**: Situation where one or more threads are unable to make progress because of another "greedy" thread.
  - Example: thread with a long-running synchronized method

Concurrency and collections

- Collections from `java.util` are *not* thread safe!
  - When two or more threads try to modify the same collection through an iterator, you get a ConcurrentModificationException.
  - When ≥ 2 threads modify a collection otherwise, bad state can result.

- But Java provides thread-safe collection *wrapper objects* via static methods in the `Collections` class:

Method
`synchronizedCollection(`**coll**`)`
`synchronizedList(`**list**`)`
`synchronizedMap(`**map**`)`
`synchronizedSet(`**set**`)`

When to make it safe?

- When creating classes for a library, you may not know exactly how the classes will be used.
  - Synchronizing everything makes your class slower.
  - Give clients a choice by supplying a *thread-safe wrapper* object.
- Or use an *immutable object*, especially if the object is small or represents a fundamental data type.
  - Immutable objects are inherently thread safe.
- When making an object thread-safe, synchronize only the *critical sections* of the class.

New classes for locking

```
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
```

Class/interface	description
`Lock`	an interface for controlling access to a shared resource
`ReentrantLock`	a class that implements `Lock`
`ReadWriteLock`	like `Lock` but separates read operations from writes
`Condition`	a particular shared resource that can be waited upon; conditions are acquired by asking for one from a `Lock`

- These classes offer higher granularity and control than the `synchronized` keyword can provide.
  - Not needed by most programs.
  - `java.util.concurrent` also contains blocking data structures.