Java 1.5 Concurrency Utilities

- Comprehensive support for general-purpose concurrent programming; partitioned into three packages:
  - `java.util.concurrent` — support common concurrent programming paradigms, e.g., various queuing policies such as bounded buffers, sets and maps, thread pools
  - `java.util.concurrent.atomic` — lock-free thread-safe programming on simple variables such as atomic integers, atomic booleans
  - `java.util.concurrent.locks` — framework for various locking algorithms, e.g., read -write locks and condition variables.

Locks I

```
package java.util.concurrent.locks;
public interface Lock {
  public void lock(); // Wait for the lock to be acquired
  public Condition newCondition();
  // Create a new condition variable for use with the Lock
  public void unlock();
  …
}

public class ReentrantLock implements Lock {
  public ReentrantLock();
  public void lock();
  public Condition newCondition();
  public void unlock();
}
```

Locks II

```
package java.util.concurrent.locks;
public interface Condition {
  public void await() throws InterruptedException;
  //Atomically releases associated lock and cause thread to wait
  public void signal(); // Wake up one waiting thread
  public void signalAll();// Wake up all waiting threads
}
```


Generic Bounded Buffer I

```
class BoundedBuffer<Data> {

  private final Data buffer[];
  private int first, last, numberInBuffer;
  private final int size;
  private final Lock lock = new ReentrantLock();
  private final Condition notFull  = lock.newCondition();
  private final Condition notEmpty = lock.newCondition();

  public BoundedBuffer(int length) {
    buffer = (Data[]) new Object[size = length];
  }
```

Generic Bounded Buffer II

```
public void put(Data item) throws InterruptedException {
  lock.lock();
  try {
    while (numberInBuffer == size) notFull.await();
    last = (last + 1) % size;
    numberInBuffer++;
    buffer[last] = item;
    notEmpty.signal();
  } finally { lock.unlock(); }
}
public Data get() throws InterruptedException {
    lock.lock();
    try {
      while (numberInBuffer == 0) notEmpty.await();
      first = (first + 1) % size ;
      numberInBuffer--;
      notFull.signal();
      return buffer[first];
    } finally { lock.unlock();}
}
```


Asynchronous Thread Control

- Early versions of Java allowed one thread to asynchronously effect another thread through

```
public class Thread {
   ...
   public final void suspend();
   public final void resume();
   public final void stop();
   public final void stop(Throwable except)
             throws SecurityException;
```

All of the above methods are now obsolete and therefore should not be used

Thread Interruption

```
public class Thread …
  public void interrupt();
    // Send an interrupt to the associated thread
  public boolean isInterrupted();
  // Returns true if associated thread has been
  // interrupted, interrupt status is left unchanged

  public static boolean interrupted();
    // Returns true if the current thread has been
    // interrupted and clears the interrupt status
```


ECS 2009

Thread Interruption

When a thread interrupts another thread:

- If the interrupted thread is blocked in `wait`, `sleep` or `join`, it is made runnable and the `InterruptedException` is thrown
- If the interrupted thread is executing, a flag is set indicating that an interrupt is outstanding; there is no immediate effect on the interrupted thread
- Instead, the called thread must periodically test to see if it has been interrupted using the `isInterrupted` or `interrupted` methods
  - If the thread doesn't test but attempts to blocks, it is made runnable immediately and the `InterruptedException` is thrown

Summary

- True monitor condition variables are not directly supported by the language and have to be programmed explicitly
- Communication via unprotected data is inherently unsafe
- Asynchronous thread control allows thread to affect the progress of another without the threads agreeing in advance as to when that interaction will occur
- There are two aspects to this: suspend and resuming a thread (or stopping it all together), and interrupting a thread
- The former are now deemed to be unsafe due to their potential to cause deadlock and race conditions
- The latter is not responsive enough for real-time systems

Completing The Java Model

- Aims:
  - To introduce thread priorities and thread scheduling
  - To show how threads delay themselves
  - To summarises the strengths and weaknesses of Java model
  - To introduce Bloch's safety levels

Thread Priorities

- Although priorities can be given to Java threads, they are only used as a guide to the underlying scheduler when allocating resources
- An application, once running, can explicitly give up the processor resource by calling the `yield` method, placing the thread to the back of the run queue for its priority level

```
public class Thread ...
  public static final int MAX_PRIORITY  = 10;
  public static final int MIN_PRIORITY  = 1;
  public static final int NORM_PRIORITY = 5;

  public final int getPriority();
  public final void setPriority(int newPriority);
  public static void yield();
```


Warning

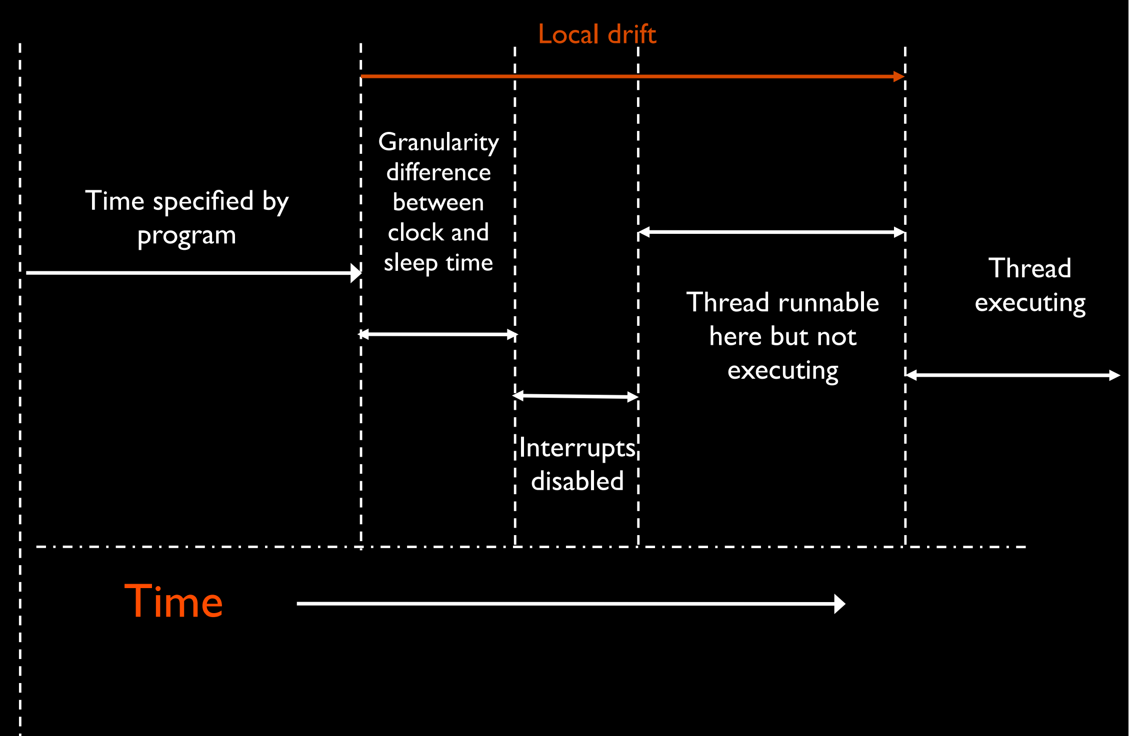
- From a real-time perspective, Java's scheduling and priority models are weak; in particular:
  - no guarantee is given that the highest priority runnable thread is always executing
  - equal priority threads may or may not be time sliced
  - where native threads are used, different Java priorities may be mapped to the same operating system priority

Delaying Threads: Clocks

- Java supports the notion of a wall clock
- `System.currentTimeMillis` returns the number of milliseconds since 1/1/1970 GMT and is used by used by `java.util.Date`
- However, a thread can only be delayed from executing by calling the `sleep` methods in the `Thread` class
- `sleep` provides a relative delay (sleep from now for some time), rather than sleep until 15th December 2003

```
class Thread...  ...
  static void sleep(long ms) throws InterruptedException;
  static void sleep(long ms,int nanoseconds)throws Interrup
```


Sleep Granularity



Absolute Delays I

- Consider an embedded system where the software controller needs to invoke two actions
- The second action must occur a specified period (say 10 seconds) after the first action has been initiated
- Simply sleeping for 10 seconds after a call to the first action will not achieve the desired effect for two reasons
  - The first action may take some time to execute. If it took 1 second then a sleep of 10 would be a total delay of 11 seconds
  - The thread could be pre-empted after the first action and not execute again for several seconds
- This makes it extremely difficult to determine how long the relative delay should be

Absolute Delays II

```
try{
  long start = System.currentTimeMillis();
  action_1();
  long end = System.currentTimeMillis();
  Thread.sleep(10000-(end-start));
} catch (InterruptedException ie) {...};
action_2();
```

What is wrong with this approach?

Timeout on Waiting I

- In many situations, a thread can wait for an arbitrary long period time within synchronized code for an associated **`notify`**
- The absence of the call, within a specified period of time, sometimes requires that the thread take some alternative action
- Java provides two methods for this situation both of which allows the **`wait`** method call to timeout
- There are two important points to note
  - As with sleep, the timeout is a relative time and not an absolute time
  - It isn't possible to know if the thread is woken by timeout or notify

Timeouts on Waiting

```
public class TimeoutException extends Exception {}

public class TimedWait {
  public static void wait(Object lock, long millis)
      throws InterruptedException, TimeoutException{
          // assumes the lock is held by the caller
    long start = System.currentTimeMillis();
    lock.wait(millis);
    if(System.currentTimeMillis() >= start + millis)
        throw new TimeoutException();
  }
}
```

What is wrong with this approach?

Strengths of the Java Concurrency Model

- Main strength is simplicity and direct support by the language
- Many of errors that potentially occur with uses of an operating system interface for concurrency do not exists in Java
- Language syntax + strong type checking gives some protection e.g., it is not possible to forget to end a synchronized block
- Portability is enhanced as the concurrency model is the same irrespective of the operating system on which the program runs

Weaknesses I

- Lack of support for condition variable
- Poor support for absolute time and time-outs on waiting
- No preference given to threads continuing after a notify over threads waiting to gain access to the monitor for the first time
- Poor support for priorities

Weaknesses II

- Synchronized code should be kept as short as possible
- Nested monitor calls should be avoided because the outer lock is not released when the inner monitor waits; this can lead to deadlocks
- It is not always obvious when a nested monitor call is made:
  - non-synchronized methods can still contain a synchronized block
  - non-synchronized methods can be overridden with a synchronized method; method calls which start off unsynchronized may be used with a synchronized subclass
  - interface methods cannot be labelled as synchronized

Bloch's Thread Safety Levels

- Immutable — Objects are constant and cannot be changed
- Thread-safe — Objects are mutable but they can be used safely in a concurrent environment as the methods are synchronized
- Conditionally thread-safe — Objects either have methods which are thread-safe, or have methods which are called in sequence with the lock held by the caller
- Thread compatible — Instances of the class provide no synchronization. However, instances of the class can be safely used in a concurrent environment, if the caller provides the synchronization by surrounding each method with the appropriate lock
- Thread-hostile — Instances should not be used in a concurrent environment even if the caller provides external synchronization. Typically because accessing static data or the external environment