**Java MultiThreading:**

A *thread* is a single sequential flow of control within a program.

A thread is similar to a real process in that both have a single sequential flow of control. However, a thread is considered lightweight because it runs within the context of a full-blown program and takes advantage of the resources allocated for that program and the program's environment.

As a sequential flow of control, a thread must carve out some of its own resources within a running program. For example, a thread must have its own execution stack and program counter. The code running within the thread works only within that context.

The `run()` method gives a thread something to do. Its code implements the thread's running behavior.

Procedures for creating threads:

- Inheriting from Thread class
- Implementing Runnable Interface

# Subclassing Thread and Overriding run

The first way to customize a thread is to subclass `Thread` (itself a `Runnable` object) and override its empty `run` method so that it does something.

**Example: Simple Threads**

`SimpleThread.java`

```
Import java.io.*;
public class SimpleThread extends Thread {
    public SimpleThread(String str) {
        super(str);
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(i + " " + getName());
            try {
                sleep((long)(Math.random() * 1000));
            } catch (InterruptedException e) {}
        }
        System.out.println("DONE! " + getName());
    }
}
```

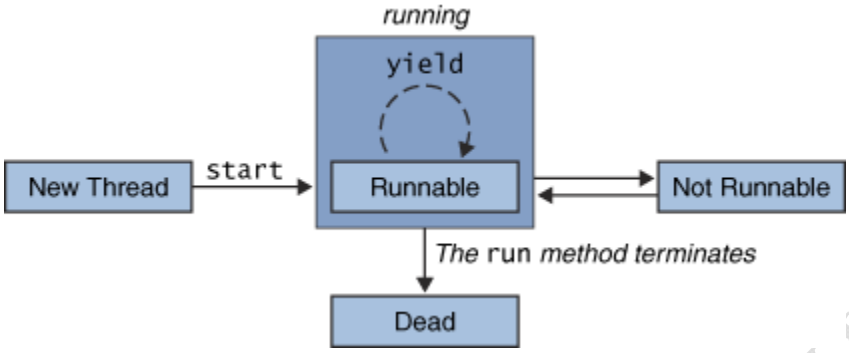
```
public class TwoThreadsTest {
    public static void main (String[] args) {
        new SimpleThread("Jamaica").start();
        new SimpleThread("Fiji").start();
    }
}
```

# Implementing Runnable

```
 import java.awt.*;
import java.util.*;
import java.applet.*;
import java.text.*;

public class Clock extends java.applet.Applet implements Runnable {
    private volatile Thread clockThread = null;
    DateFormat formatter;     // Formats the date displayed
    String lastdate;          // String to hold date displayed
    Date currentDate;         // Used to get date to display
    Color numberColor;        // Color of numbers
    Font clockFaceFont;
    Locale locale;

    public void init() {
        setBackground(Color.white);
        numberColor = Color.red;
        locale = Locale.getDefault();
        formatter =
            DateFormat.getDateTimeInstance(DateFormat.FULL,
            DateFormat.MEDIUM, locale);
        currentDate = new Date();
        lastdate = formatter.format(currentDate);
        clockFaceFont = new Font("Sans-Serif",
                                 Font.PLAIN, 14);
        resize(275,25);
   }

    public void start() {
      if (clockThread == null) {
         clockThread = new Thread(this, "Clock");
         clockThread.start();
      }
    }

    public void run() {
      Thread myThread = Thread.currentThread();
        while (clockThread == myThread) {
            repaint();
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e){ }
        }
    }

    public void paint(Graphics g) {
```

```
        String today;
        currentDate = new Date();
        formatter =
             DateFormat.getDateTimeInstance(DateFormat.FULL,
             DateFormat.MEDIUM, locale);
        today = formatter.format(currentDate);
        g.setFont(clockFaceFont);

        // Erase and redraw
        g.setColor(getBackground());
        g.drawString(lastdate, 0, 12);

        g.setColor(numberColor);
        g.drawString(today, 0, 12);
        lastdate = today;
        currentDate=null;

    }

    public void stop() {
        clockThread = null;
    }
}
```

# The Life Cycle of a Thread




## Creating a Thread

```
    if (clockThread == null) {
        clockThread = new Thread(this, "MyThreadName");
        clockThread.start();
    }
```

A thread in this state is merely an empty `Thread` object; no system resources have been allocated for it yet.

When a thread is in this state, you can only start the thread.

Calling any method besides `start` when a thread is in this state makes no sense and causes an `IllegalThreadStateException`.

## Starting a Thread

```
  if (clockThread == null) {
        clockThread = new Thread(this, "Clock");
        clockThread.start();
  }
```

The `start` method creates the system resources necessary to run the thread, schedules the thread to run, and calls the thread's `run` method.

```
public void run() {
   Thread myThread = Thread.currentThread();
   while (clockThread == myThread) {
        repaint();
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            // The VM doesn't want us to sleep anymore,
            // so get back to work
        }
    }
}
```

`Clock`'s `run` method loops while the condition `clockThread == myThread` is `true`.

## Making a Thread Not Runnable

A thread becomes Not Runnable when one of these events occurs:

- Its `sleep` method is invoked.
- The thread calls the `wait` method to wait for a specific condition to be satisifed.
- The thread is blocking on I/O.

- If a thread has been put to sleep, the specified number of milliseconds must elapse.
- If a thread is waiting for a condition, then another object must notify the waiting thread of a change in condition by calling `notify` or `notifyAll`.
- If a thread is blocked on I/O, the I/O must complete.

## Stopping a Thread

Although the `Thread` class does contain a `stop` method, this method is deprecated and should not be used to stop a thread because it is unsafe.
Rather, a thread should arrange for its own death by having a `run` method that terminates naturally.

# Thread Scheduling

Execution of multiple threads on a single CPU in some order is called *scheduling*.

The Java runtime environment supports a very simple, deterministic scheduling algorithm called *fixed-priority scheduling*. This algorithm schedules threads on the basis of their priority relative to other Runnable threads.

When a thread is created, it inherits its priority from the thread that created it.

A thread's priority can be modified at any time after its creation by using the `setPriority()` method. Thread priorities are integers ranging between `MIN_PRIORITY` and `MAX_PRIORITY` (constants defined in the `Thread` class). The higher the integer, the higher the priority.

At any given time, when multiple threads are ready to be executed, the runtime system chooses for execution the Runnable thread that has the highest priority.

Only when that thread stops, yields, or becomes Not Runnable will a lower-priority thread start executing.

A thread can voluntarily yield the CPU by calling the `yield()` method. The `yield()` method gives other threads of the same priority a chance to run. If no equal-priority threads are Runnable, the `yield()` is ignored.

If two threads of the same priority are waiting for the CPU, the scheduler arbitrarily chooses one of them to run. The chosen thread runs until one of the following conditions is true:

- A higher priority thread becomes runnable.
- It yields, or its `run` method exits.
- On systems that support time-slicing, its time allotment has expired.

Then the second thread is given a chance to run, and so on, until the interpreter exits.

The Java runtime system's thread scheduling algorithm is also preemptive. At any given time, the highest priority thread is running. However, this is not guaranteed. The thread scheduler may choose to run a lower priority thread to avoid starvation. For this reason, use thread priority only to affect scheduling policy for efficiency purposes. Do not rely on it for algorithm correctness.

# Synchronizing Threads

Race Condition: A *race condition* is a situation in which two or more threads or processes are reading or writing some shared data, and the final result depends on the timing of how the threads are scheduled.

Solution:

- Critical Sections & Locks (Monitors)
- Notify() & wait()

Within a program, the code segments that access the same object from separate, concurrent threads are called *critical sections*. A critical section can be a block or a method and is identified with the synchronized keyword.

The Java platform associates a lock with every object and the lock is acquired upon entering a critical section.

The same thread can call a synchronized method on an object for which it already holds the lock, thereby reacquiring the lock. The Java runtime environment allows a thread to reacquire a lock because the locks are *reentrant*. Reentrant locks are important because they eliminate the possibility of a single thread's waiting for a lock that it already holds.

To avoid race condition and provide some concurrency control or thread synchronization, the Object class provides a collection of methods — wait, notify, and notifyAll — to help threads wait for a condition and notify other threads when that condition changes.

- The `notifyAll` method wakes up all threads waiting on the object in question (in this case, the `CubbyHole`). The awakened threads compete for the lock. One thread gets it, and the others go back to waiting. The `Object` class also defines the `notify` method, which arbitrarily wakes up one of the threads waiting on this object.

There are the three versions of the `wait` method contained in the `Object` class:

- **wait()**
  - Waits indefinitely for notification. (This method was used in the producer-consumer example.)
- **wait(long *timeout*)**
  - Waits for notification or until the *`timeout`* period has elapsed. `timeout` is measured in milliseconds.
- **wait(long *timeout*, int *nanos*)**

- Waits for notification or until *timeout* milliseconds plus nanos nanoseconds have elapsed.

Besides using these timed wait() methods to synchronize threads, you also can use them in place of sleep(). Both wait() and sleep() delay for the requested amount of time. You can easily wake up wait() with a notify() but a sleeping thread cannot be awakened prematurely.

Side effects of concurrency control:

- *Starvation* occurs when one or more threads in your program are blocked from gaining access to a resource and, as a result, cannot make progress.
  - *If all threads are same priority than starvation wont occur*
- *Deadlock*, the ultimate form of starvation, occurs when two or more threads are waiting on a condition that cannot be satisfied. Deadlock most often occurs when two (or more) threads are each waiting for the other(s) to do something.

**Example: The Producer Consumer Thread Synchronization**

```
public class Producer extends Thread {
    private CubbyHole cubbyhole;
    private int number;

    public Producer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }

    public void run() {
        for (int i = 0; i < 10; i++) {
            cubbyhole.put(number, i);
            try {
                sleep((int)(Math.random() * 100));
            } catch (InterruptedException e) { }
        }
    }
}


public class Consumer extends Thread {
    private CubbyHole cubbyhole;
    private int number;

    public Consumer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }

    public void run() {
```

```
        int value = 0;
        for (int i = 0; i < 10; i++) {
            value = cubbyhole.get(number);
        }
    }
}


public class CubbyHole {
    private int contents;
    private boolean available = false;

    public synchronized int get(int who) {
        while (available == false) {
            try {
                wait();
            } catch (InterruptedException e) { }
        }
        available = false;
        System.out.println("Consumer " + who + " got: " + contents);
        notifyAll();
        return contents;
    }

    public synchronized void put(int who, int value) {
        while (available == true) {
            try {
                wait();
            } catch (InterruptedException e) { }
        }
        contents = value;
        available = true;
        System.out.println("Producer " + who + " put: " + contents);
        notifyAll();
    }
}

public class ProducerConsumerTest {
    public static void main(String[] args) {
        CubbyHole c = new CubbyHole();
        Producer p1 = new Producer(c, 1);
        Consumer c1 = new Consumer(c, 1);

        p1.start();
        c1.start();
    }
}
```

Here's the output of `ProducerConsumerTest`:

```
Producer #1 put: 0
Consumer #1 got: 0
Producer #1 put: 1
Consumer #1 got: 1
Producer #1 put: 2
Consumer #1 got: 2
Producer #1 put: 3
Consumer #1 got: 3
```

```
Producer #1 put: 4
Consumer #1 got: 4
Producer #1 put: 5
Consumer #1 got: 5
Producer #1 put: 6
Consumer #1 got: 6
Producer #1 put: 7
Consumer #1 got: 7
Producer #1 put: 8
Consumer #1 got: 8
Producer #1 put: 9
Consumer #1 got: 9
```

# Explicit Locks and Condition Variables

Another way to ensure exclusive access to a section of code is to use an explicit lock. An explicit lock is more flexible than using the `synchronized` keyword because the lock can span a few statements in a method, or multiple methods in addition to the scopes (block and method) supported by `synchronized`.

To create an explicit lock you instantiate an implementation of the `Lock` interface, usually `ReentrantLock`. To grab the lock, you invoke the `lock` method; to release the lock you invoke the `unlock` method. Since the lock is not automatically released when the method exits, you should wrap the `lock` and `unlock` methods in a `try`/`finally` clause.

To wait on an explicit lock, you create a condition variable (an object that supports the `Condition` interface) using the `Lock.newCondition` method. Condition variables provide the methods `await` to wait for the condition to be true, and `signal` and `signalAll` to notify all waiting threads that the condition has occurred. Like the `Object.wait` method, `Condition.await` has several variants, which are listed in the next table.

| **`Condition.await` Methods** | |
|---|---|
| **Method** | **Description** |
| `await` | Waits for a condition to occur. |
| `awaitInterruptibly` | Waits for a condition to occur. Cannot be interrupted. |
| `awaitNanos(long timeout)` | Waits for a condition to occur. If the notification does not occur before a timeout specified in nanoseconds, it returns. |
| `await(long timeout, TimeUnit unit)` | Waits for a condition to occur. If the notification does not occur before a timeout specified in the provided time unit, it returns. |
| `await(Date timeout)` | Waits for a condition to occur. If the notification does not occur before the specified time, it returns. |
| | |

In the following example, `CubbyHole` has been rewritten to use an explicit lock and condition variable. To run this version of the Producer-Consumer example, execute

`ProducerConsumerTest2`.

```
import java.util.concurrent.locks.*;
public class CubbyHole2 {
    private int contents;
    private boolean available = false;
    private Lock aLock = new ReentrantLock();
    private Condition condVar = aLock.newCondition();
```

```
    public int get(int who) {
        aLock.lock();
        try {
            while (available == false) {
                try {
                    condVar.await();
                } catch (InterruptedException e) { }
            }
            available = false;
            System.out.println("Consumer " + who + " got: " +
                               contents);
            condVar.signalAll();
        } finally {
            aLock.unlock();
            return contents;
        }
    }

    public void put(int who, int value) {
                    aLock.lock();
    try {
        while (available == true) {
            try {
                condVar.await();
            } catch (InterruptedException e) { }
        }
        contents = value;
        available = true;
        System.out.println("Producer " + who + " put: " +
                           contents);
        condVar.signalAll();
        } finally {
            aLock.unlock();
        }
    }
}
```

# Using the Timer and TimerTask Classes

The `Timer` class in the `java.util` package schedules instances of a class called `TimerTask`.

Example: Reminder

Reminder.java

```
import java.util.Timer;
import java.util.TimerTask;

/**
 * Simple demo that uses java.util.Timer to schedule a task
 * to execute once 5 seconds have passed.
 */

public class Reminder {
    Timer timer;

    public Reminder(int seconds) {
        timer = new Timer();
        timer.schedule(new RemindTask(), seconds*1000);
        }

    class RemindTask extends TimerTask {
        public void run() {
            System.out.println("Time's up!");
            timer.cancel(); //Terminate the timer thread
        }
    }

    public static void main(String args[]) {
        new Reminder(5);
        System.out.println("Task scheduled.");
    }
}
```

When you run the example, you first see this:

```
Task scheduled.
```

Five seconds later, you see this:

```
Time's up!
```



## Performing a Task Repeatedly

Here's an example of using a timer to perform a task once per second.

```
import java.util.Timer;
import java.util.TimerTask;
import java.awt.Toolkit;
```

```
/**
 * Schedule a task that executes once every second.
 */

public class AnnoyingBeep {
    Toolkit toolkit;
    Timer timer;

    public AnnoyingBeep() {
      toolkit = Toolkit.getDefaultToolkit();
        timer = new Timer();
        timer.schedule(new RemindTask(),
                    0,        //initial delay
                    1*1000);  //subsequent rate
    }

    class RemindTask extends TimerTask {
      int numWarningBeeps = 3;

        public void run() {
          if (numWarningBeeps > 0) {
              toolkit.beep();
            System.out.println("Beep!");
            numWarningBeeps--;
          } else {
              toolkit.beep();
                System.out.println("Time's up!");
              //timer.cancel(); //Not necessary because we call
System.exit
              System.exit(0);   //Stops the AWT thread (and everything
else)
          }
        }
    }

    public static void main(String args[]) {
      System.out.println("About to schedule task.");
        new AnnoyingBeep();
      System.out.println("Task scheduled.");
    }
}
```



doing, even while entering a new event