Random Stuff
Many programs make use of random numbers. We've used Java's Random class several times this semester. Of course, computers don't really make random numbers -- computers follow algorithms, and those aren't random. What computers generate are "pseudo-random" sequences of numbers -- sequences that look random, even though they are deterministically generated.
Lots of people have had the idea of using the system clock or some other connection to the physical world for random number generation. No one has been able to make very effective use of that. If a program accesses the system clock once, the time at which that happens is fairly random. If you do multiple accesses within a loop, the time difference is fairly predictable and the numbers don't look random.
Here are some properties we would like our random number generator to have:

A. If you generate a large collection of random numbers, the count of how many fall in a specific region should be proportional to the size of that region.

B. If you look at the sequence of random values mod N the resulting sequence should also look random, no matter what value N has.

C. If you generate a large number of pairs of random numbers there should not be a correlation between the first and second elements of the pairs, so if you plot them as points you shouldn't see predictable patterns.

D. Some simulations require large sets of random numbers to be generated quickly, so random number generation must be fast.
There is money to be made from random numbers, so a lot of work has gone into this issue. If you are interested, see Chapter 3 of Knuth's *The Art Of Computer Programming*. There is plenty of more recent work that is highly mathematical. Unfortunately, most of the advanced algorithms for random number generation are slow.

The Java Utilities class Random, along with the mathematical libraries of most programming languages, actually does something quite simple.
One simple technique is a *linear congruential* random number generator. This works by choosing some positive integer value for $X_0$, then defining subsequent values as

$$X_{i+1} = (A \times X_i) \mod M$$

where $A$ and $M$ are fixed constants.

For example, if $X_0 = 5$, $M = 13$ and $A = 5$ we get

<table>
<thead>
<tr>
<th>$i$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_i$</td>
<td>5</td>
<td>12</td>
<td>8</td>
<td>1</td>
<td>5</td>
<td>12</td>
<td>8</td>
<td>1</td>
<td>5</td>
<td>12</td>
<td>8</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

If we change $M$ to 17 and $A$ to 7 we get a much longer cycle:

<table>
<thead>
<tr>
<th>$i$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_i$</td>
<td>5</td>
<td>1</td>
<td>7</td>
<td>15</td>
<td>3</td>
<td>4</td>
<td>11</td>
<td>9</td>
<td>12</td>
<td>16</td>
<td>10</td>
<td>2</td>
<td>14</td>
<td>13</td>
<td>6</td>
<td>8</td>
<td>5</td>
</tr>
</tbody>
</table>
Some authors, after a lot of testing, say that good choices for the constants are

\[ M = 2^{31}-1 = 2,147,483,647 \] (which is prime)
\[ A = 48,271 \]
The starting number for this sequence is called the *seed*. The Java documentation doesn't say where it gets the seed it uses, only that the seed is "very likely to be distinct from any other invocation" of the random number generator. Since java does the computation using 48 bits, there are about $10^{14}$ possible choices. If you used one per second it would take about 3 million years to use them all. We probably won't be coding in Java 3 million years from now....

One reasonable guess is that Java uses the system clock to choose the initial seed.
Note that you can set the value of the seed when you construct a random number generator:

```java
Random rand = new Random(34);
```
creates as random number generator that uses 34 as the value of its seed. Using a seed that you know can be helpful when you are trying to debug a program that makes use of random numbers and you want the program to behave the same way each time you run it. After you set the seed the random number sequence will be the same every time you run the program.
Note that Java actually uses

\[ X_{i+1} = (A \times X_i + C) \mod 2^B \]

for its random number generator, where

- \( A = 25,214,903,917 \)
- \( C = 11 \)
- \( B = 48 \)

because division by powers of 2 can be implemented with a bit-shift, making this quick to calculate. Java does the full calculation in terms of 48-bit integers, and does some arithmetic games to make sure that there is no integer overflow.

This isn't as good (meaning it doesn't have as long a period) as the simple \( X_{i+1} = (A \times X_i) \mod M \) generator with \( A=48,271 \) and \( M=2^{31}-1 \), but it is faster to calculate because the \( \mod 2^B \) operation can be performed very quickly.
One trick every programmer should know is a simple array shuffle algorithm. The Fisher-Yates algorithm, also known as the Knuth Shuffle walks from one of the array to the other. At each step we choose a random index between the current location and the end, and interchange the data items at those two locations:

```java
for( int i = A.length-1; i >= 1; i-- ) {
    int j = rand.nextInt(i+1);
    swap(A, i, j);
}
```

This shuffle is *unbiased* in that every permutation of the array is equally likely, assuming that the random number generator is *perfectly random*. Of course, random number generators aren't perfectly random.
In fact, if the random number generator is based on 48-bit integers, like Java's is, there are no more than $2^{48}$ different states for the generator. This means it can't possibly generate all $52!$ different shuffles of a deck of cards.