# Cosc 241
# Programming and Problem Solving
# Lecture 11 (1/4/19)
# Random

Michael Albert
michael.albert@cs.otago.ac.nz

```
int getRandomNumber()
{
    return 4;  // chosen by fair dice roll.
               // guaranteed to be random.
}
```

http://xkcd.com/221/

UNIVERSITY of OTAGO
Te Whare Wānanga o Otāgo
NEW ZEALAND

# Why randomness?

- Game playing: dice rolls, shuffling cards, choosing lottery winners.
- Game playing: unpredictable actions by AI agents.
- Simulation and testing.
- Security: disc wiping and digital document shredding.
- Cryptography and communication protocols including https.
- Some quotes and some history.

# Random vs. pseudo-random

- “True” random numbers (bits, integers, . . . ) are generated by observation of some unpredictable physical process.
- This is generally a slow and relatively speaking computationally expensive process which until recently required special purpose hardware.
- “Pseudo-random” numbers are generated as a sequence by a specific deterministic mathematical algorithm called a pseudo-random number generator – they appear unpredictable when observed, but if the initial *seed* is known as well as the algorithm, then they can be predicted with 100% accuracy.
- So “pseudo-random” means “very fast, but not actually random at all”.

# Randomness in Java

Java provides three ways to access (pseudo-)randomness:

- Math.random() which "Returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0."
- The java.util.Random class: "...used to generate a stream of pseudorandom numbers. The class uses a 48-bit seed, which is modified using a linear congruential formula."
- The java.security.SecureRandom class: "...provides a cryptographically strong random number generator (RNG)."

# Linear congruential formulas

- A classical type of pseudo-random number generator.
- The seed $s$ is updated by a rule of the form

$$s \leftarrow ( s \times a + b ) \,\%\, c$$

  where $a$, $b$ and $c$ are fixed in the algorithm.
- Either $s$ or some part of it is returned as the next value.

# In Java

```
seed = (seed * 0x5DEECE66DL + 0xBL) & ((1L « 48) - 1)
```

- Note the use of hexadecimal representation of long values (0x means “read as hexidecimal” and the trailing L means “long”).
- The $c$ value is $2^{48}$ and the modulus is implemented via a bitwise manipulation (taking “and” with a string of 47 ones).

# NextInt

If r is a Random object, then r.nextInt(n) is supposed to return a pseudo-random integer between 0 and n-1. How?

```
public int nextInt(int n) {
   if (n <= 0)
     throw new IllegalArgumentException("n must be positive");

   if ((n & -n) == n)  // i.e., n is a power of 2
     return (int)((n * (long)next(31)) >> 31);

   int bits, val;
   do {
       bits = next(31);
       val = bits % n;
   } while (bits - val + (n-1) < 0);
   return val;
 }
```

The documentation states “The algorithm is slightly tricky”. It repays consideration.

# Picking a winner

Is easy:

```
private static final Random R = new Random();

public  static <T> T winner(T[] entries) {
        return entries[R.nextInt(entries.length)];
}
```

What if we want multiple winners? How to avoid duplicating picks and/or testing for equality?
One method:

- pick a *subset* of the indices of the array of entries (this represents the set of winners),
- shuffle that subset (now we have them in order),
- now create the array of winners using those indices to pick from the entries.

# Shuffling

- How can we shuffle an array of items?
- We want to make sure that every ordering is equally likely.
- Think of a simple physical model.

# Shuffling code

```
public class Shuffler{

    private static final Random R = new Random();

    public void shuffle(int[] a) {
        for(int i = a.length-1; i > 0; i--) {
            swap(a, i, R.nextInt(i+1));
        }
    };

    private void swap(int[] a, int i, int j) {
        int t = a[i];
        a[i] = a[j];
        a[j] = t;
    }

}
```

# We still have a problem

We would like a method that takes in an array `entries` of type `T` and an `int n` and returns an ordered list of `n` different winners from `entries`.

- It should not rearrange `entries`.
- It should not use more storage than necessary (i.e. about an additional `n` items for the list of winners).
- It should not require tests for equality or duplicate rejection.