PRNG Lesson

This is a hands-on math exercise. Students will use a simple formula to simulate a PRNG.

### CSP Learning Objectives:

* *Abstraction: 2.3.1 - The student can use models and simulations to raise and answer questions. [P3]*

# Introduction

A *model* or *simulation* is an abstraction. It provides a simplification of some complicated real-world thing or event, with many of the real-world details left out or ‘abstracted away’. Think of a model airplane, which 'abstracts away' some of the details of a real airplane -- e.g., its size and weight, the materials its made of, and so on. What other models can we think of? The sculptor's model of Mt. Rushmore? A kitchen layout plan?

In this lesson we're going to look at a model of ***randomness***.

## Randomness

Randomness is an important element in many computer applications, especially games and simulations. Here are some examples of *random events:*

* Pick 1 of 25 numbered marbles out of a completely mixed up jar of marbles. (1 in 25 chance of picking a particular number).
* Flip a fair coin: 1 in 2 or 50/50 chance of it coming up heads.
* Pick a card out of a thoroughly shuffled deck (4 in 52 chance of drawing an Ace).

### What is a Random Event?

Think about the coin flip at the start of the Super Bowl (or some other sporting event).

1. What makes that a random event?
2. Why is it important that the coin flip have a 50/50 chance of coming up heads?
3. What sort of 'rigging' could occur to take away its fairness?
4. Could a deck of 52 cards be used (instead of a coin flip) to determine which team was on offense?
5. What other types of random events might suffice to select which team kicks off?

It's important that the coin flip be completely unpredictable. That's what we mean by a random event. Although we know that the chance of the coin coming up heads is 50/50, we don’t know that it will come up heads when we flip it. If we did know, that would be predictable. And that would not be a random event. For example, suppose the coin used at the Super Bowl were replaced by one that had ‘heads’ on both sides. Then flipping the coin would be completely predictable -- it will come up ‘heads’ the next time it is flipped.

It’s even more subtle than that. Suppose you knew that a particular (unfair) coin had a 75% chance of coming up heads. That is, on average it would come up heads 3 times out of every 4 flips. Although flipping the coin is still a random event, the coin is not a good model for a 50/50 coin flip. Someone who knew that it was unfair could exploit that information to their advantage.

### Computer Randomness

Computers can't do real randomness -- they can't really flip a coin or pick a card out of the deck. Computers use a form of randomness known as ***pseudo randomness*** -- that is, they ***simulate*** ***randomness***. A *pseudo random event* looks random but is completely predictable -- we say it is *deterministic* because its output can be known by someone who knows how the event was programmed. What looks random to the user is actually the result of a completely predictable mathematical algorithm.

When a computer application 'flips a coin', it's not really flipping a coin -- it is simulating a coin flip. Similarly, if you go to a casino and play computer poker on a machine, the computer is not really playing poker -- it is simulating a poker game. We want to understand how this works.

### A Pseudo Random Number Generator

As you might expect, computers simulate randomness using numbers. For pseudo random events we use a [pseudo random number generator (PRNG)](http://en.wikipedia.org/wiki/Random_number_generation) which is a mathematical formula that generates a random-looking sequence. If you know the first number in the sequence, also known as the *seed*, then you can predict every number in the sequence.

PRNGs are useful in:

* Cryptography -- used to generate secret keys.
* Computer games -- card games, shooter games, adventure games, etc.
* Simulations -- weather models, astronomical models, models of biological systems, etc.

## Randomness in App Inventor

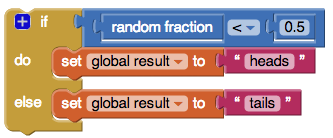
App Inventor has several blocks for randomness in the Toolbox’s Math drawer.

**1. Random Fraction:** Randomly selects a number between 0 and 1 (not including 1):



In other words, *random fraction* will generate numbers such 0, 0.1, 0.11, 0.2, 0.5, 0.999, and so on. It will never generate the number 1.

As you know from math class, there are an infinite number of values between 0 and 1. And they are evenly distributed between 0 and 1. A computer can’t really have an infinite amount of numbers, but it can have lots of values between 0 and 1 and they would be evenly distributed along that range. So in the real world and in App Inventor if you picked a random number between 0 and 1, you would have a 50/50 chance of getting a number that is less than 0.5. Knowing this, we could simulate a coin flip in App Inventor with the following code:



## 2. Random Integer: Randomly selects a number between two specified integers, inclusive:



For example, if you were modeling a die, which has the values 1 through 6 on its faces, you would use the following block:



And, if App Inventor’s PRNG is a good one, you would expect to that the chances of getting a 1 or a 6 or any other value would be 1 chance out of 6.

**3. Random Set Seed**: Sets the seed for the PRNG:



As we will see, if you set the seed to a certain value, the PRNG will always generate the exact same sequence of numbers. This predictability can be useful when you’re designing a game or a simulation.

### 

### 

### How does a PRNG Work?

The idea that you can simulate randomness with a mathematical formula might seem strange. So let’s develop a simple example of a PRNG. Don’t worry about the math used here. It will be pretty simple.

Consider a formula that determines how to calculate the next value in a sequence of numbers. Let’s label the numbers X1, X2, X3, and so forth. To calculate, X2 let’s multiply X1 by a certain number and add some other number to it. For example:

X2 = X1 \* 2 + 1

So, if X1 is 10, then X2 would be 21 (10 \* 2 + 1). Right? Finish completing the table below:

|  |  |  |  |
| --- | --- | --- | --- |
|  | *Xi* | *Calculation: Xi+1 = Xi \* 2 + 1* | *Xi+1* |
| *X1* | *10* | *10 \* 2 + 1 = 21* | *21* |
| *X2* | *21* |  |  |
| *X3* |  |  |  |
| *X4* |  |  |  |
| *X5* |  |  |  |

Now you have generated a sequence of numbers. But, they aren’t very random looking because they keep getting bigger and bigger. Thus, the formula

Xn+1 = Xn \* 2 + 1

is not a very good model of a PRNG.

NOTE that we have generalized the formula -- i.e., made it more abstract -- by substituting variables Xn and Xn+1 for the Xa and X2. But it still means the same thing: To generate the next value (Xn+1) from the current value (Xn), multiply the current value by 2 and add 1 (Xn \* 2 + 1).

NOTE that you could also describe this formula as an *algorithm* for generating the next value in the sequence given the current value in the sequence:

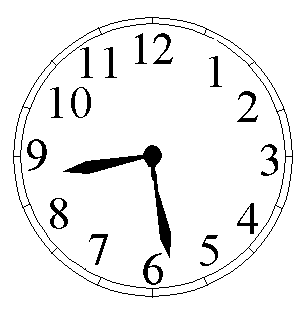
1. Let the current value, Xn be 10.

2. Multiply it by 2, giving 20.

3. Add 1, giving 21.

### 

### Clock Arithmetic



To prevent the sequence from always getting bigger, mathematicians use a trick that you might know as *clock arithmetic*. Suppose we have a 12 hour clock and you left home at 9:00 o’clock and returned 4 hours later. It would be 1:00 o’clock (not 13 o’clock). In other words, if you add (9 + 4) you get 13, but you take 12 away from 13 to get the actual time on the clock.

Let’s try another example. Suppose you left home at 11 and returned 5 hours later. You would be home at 4 o’clock. 11 + 5 = 16 - 12 = 4. Right? What if you left at 11:00 and return home 20 hours later. What time would that be? Well 11 + 20 = 31. If we subtract 12 from 31 we get 19. But there is no 19 o’clock, so we have to subtract 12 again. That will give us 7:00 o’clock. Right? Check the answer on a clock by starting from 11 and then counting 20. You should end up at 7:00 o’clock.

In other words, to calculate the time, if our sum is greater than 12 we keep subtracting 12 until we get a number between 1 and 12.

And notice that the only times we can get on our 12 hour clock are 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, and 12. And actually 12 is the same as 0. Mathematicians have a name for this operation. It’s called the *modulo* operation or *mod* for short. Let’s rewrite our clock calculations to include this operation:

9:00 o’clock plus 4 hours: (9 + 4) mod 12 = 1

11:00 o’clock plus 5 hours: (11 + 5) mod 12 = 4

11:00 o’clock plus 20 hours: (11 + 20) mod 12 = 7

Now, try these [clock arithmetic exercises](http://www-math.ucdenver.edu/~wcherowi/clockar.html).

To make our PRNG behave more randomly, let’s apply some clock arithmetic to the formula, but let’s use a 13 hour clock. So the numbers allowed will be 1 through 13, (or 0 through 12, which amounts to the same). So let’s rewrite our formula as follows:

X2 = X1 \* 2 + 1 *mod* 13

Now, if X1 is 10, then X2 will be (10 \* 2 +3 = 21 - 13) = 8. What would X3 be? It would be 8 \* 2 + 1 = 17 - 13 = 4. Let’s make a little table of these numbers. The way to read this table is that a number, Xi, is plugged into the formula to get the next number, Xi+1. Then that number is carried down to the next row to become the new value of Xi. So X1 is 10 and X2 is 8 and so on.

Finish completing the table below:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Xi | Calculation: Xi+1 = Xi \* 2 + 1 mod 13 | Xi+1 |
| X1 | 10 | 10 \* 2 + 1 mod 13 = 21 - 13 = 8 | 8 |
| X2 | 8 |  |  |
| X3 |  |  |  |
| X4 |  |  |  |
| X5 |  |  |  |
| X6 |  |  |  |
| X7 |  |  |  |
| X8 |  |  |  |
| X9 |  |  |  |
| X10 |  |  |  |
| X11 |  |  |  |
| X12 |  |  |  |

We’re now starting to see a more random looking sequence. Notice that once the 10 repeats itself the entire sequence will repeat itself over-and-over. So that’s what we mean when we say that a *pseudo random number generator* (PRNG) is completely predictable. Given any number in the sequence, we can calculate the next number. But, at the same time, the PRNG will generate a sequence of numbers that looks completely random.

Notice how this particular example has generated 12 distinct numbers before repeating itself. This is not a very good PRNG because the number 12 never comes up. Mathematicians work very hard to choose the constants in these kinds of formulas (2, 1, and 13), so that the sequence behaves more like a truly random sequence. And the designers of App Inventor have used a well-accepted standard PRNG for their implementation of the *random-fraction* and *random-integer* blocks. So you can be fairly confident that App Inventor’s PRNG provides a pretty good simulation of randomness.

## Conclusion

To sum up, a PRNG is a model (or simulation) of randomness. Like our simple example, if you start from a given value (the seed) the PRNG will generate a sequence of numbers and will eventually return to the seed value. So it is completely predictable. As such it is an abstraction. It uses a completely deterministic formula or algorithm to generate a sequence of numbers that behave like real random numbers.

What does it mean ‘to behave like real random numbers’? Well, among other things, the PRNG is just as likely to generate a random fraction that is less than 0.5 as one that is greater than 0.5. If you used it to simulate a coin flip, it should generate a ‘head’ 50% of the time, on average. If you used it to simulate a die, it should generate a ‘1’ every 1 in 6 rolls, on average.

***What determines whether a PRNG is a good model or not is how well it simulates a truly random sequence.***

## 

## 

# Reflection for the Student

In your portfolio, create a new page named ***PRNG****.* On this page, explain in your own words what a PRNG is and how a PRNG works. Also, provide an explanation of the difference between a *truly random event* and a *pseudo random number*.

# Exercises (Formative Assessment)

Here are some [interactive exercises](https://sites.google.com/site/mobilecsp/lessons/prng) based on this lesson.

# Food for Thought or Discussion

Much of the commerce and other transactions that take place on the Internet or World Wide Wide depend on strong ***encryption*** protocols which, in turn, depend upon good PRNGs. Because mathematicians and computer scientists know (or can find out) the formulas that are used in these PRNGs, what prevents them from stealing our secrets and data? More generally, how can completely predictable algorithms be used to secure the Internet? This point will be discussed when we take up cryptography.

# Follow up Activity

Design an app that tests whether App Inventor’s PRNG is a good one? What should such an app do?