Ethan Nelson

3/12/25

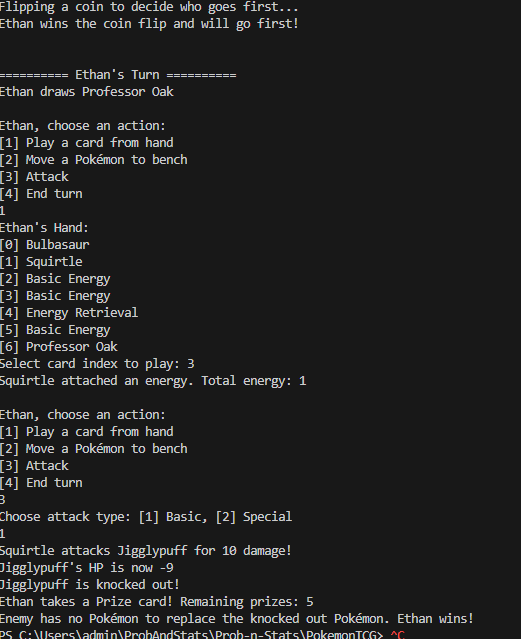
Project 1 Documentation

**PokemonTCG**

**Overview:**

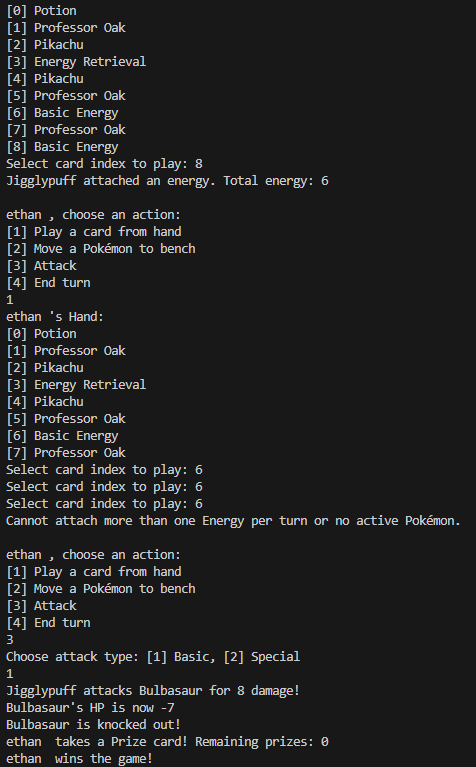
* **Card Base Class:** This is the parent class for all cards. Every card (Pokemon, Trainer, Energy) has a name. It serves as the starting point for more specific card types.
* **PokemonCard Class:** An abstract class that extends Card. It represents a Pokemon card and stores key attributes like HP, attack damage, attached energy, and the energy needed for a special attack. Each Pokemon type (Bulbasaur, Charmander, Meowth, Pikachu, Squirtle, Jigglypuff) extends this class and defines its own special attack.
* **EnergyCard Class:** A simple class that extends Card. It represents an Energy card that can be attached to Pokemon to use power attacks.
* **TrainerCard Class:** An abstract class that also extends Card. It represents Trainer cards which perform special actions when played. Specific Trainer cards (Switch, Potion, Energy Retrieval, Professor Oak) extend this class and use their unique effects.
* **Player Class:** This class holds all the information about a player. It contains a deck (a list of cards), a hand, a bench (up to 5 Pokemon), an active Pokemon, and a prize pile. It also includes methods for drawing cards, playing cards, and moving Pokemon to the bench.
* **GameEngine Class:** This class controls the game flow. It sets up the players, builds and shuffles decks, handles the initial hand and mulligan rule, manages turns (including drawing cards, playing cards, and attacking), and checks for win conditions (using prize cards). The main game loop is contained here.

This Screenshot shows a player winning when the other player has no more pokemon on the bench



I turned down the health for this example just to make things easier. Same for the next example.

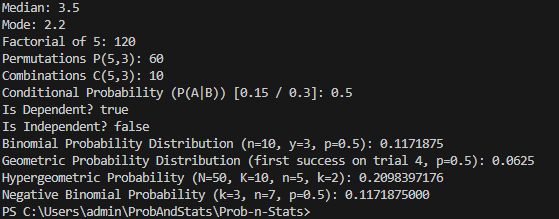
This Screenshot shows winning the normal way of obtaining all 6 of the prize pile cards.



**Next we have the StatsLibrary class which holds almost every formula we went over so far**

* **standeredDev(ArrayList<Double> data)** Computes the standard deviation for a list of doubles (using a simple integer-based approach).
* **getMean(int[] userInputNumbers)** Calculates the arithmetic mean of an array of integers.
* **getMedian(int[] a)** Determines the median value from an array of integers.
* **getMode(double a[])** Finds the mode in an array of doubles.
* **factorial(int num)** Computes the factorial of a number using BigInteger (handles large numbers).
* **permutations(int n, int r)**  Calculates the number of permutations using the factorial function.
* **combinations(int n, int r)** Calculates the number of combinations using the factorial function.
* **conditionalProbability(double pAAndB, double pB)** Computes the conditional probability
* **isDependent(double pA, double pB, double pAGivenB, double pBGivenA, double pAAndB)** Checks if two events are dependent based on provided probabilities.
* **isIndependent(double pA, double pB, double pAGivenB, double pBGivenA, double pAAndB)** Checks if two events are independent based on provided probabilities.
* **binomialProbabilityDistribution(double p, double q, int n, int y)** Computes the binomial probability using the formula:  
   C(n,y)×py×q(n−y)\displaystyle C(n, y) \times p^y \times q^{(n-y)}C(n,y)×py×q(n−y).
* **geometricProbabilityDistribution(double q, double p, int y)** Computes the geometric probability that the first success occurs on the y’th trial.
* **hypergeometricProbabilityBI(int N, int K, int n, int k)**  Calculates the hypergeometric probability using BigInteger and BigDecimal for precision.
* **negativeBinomialProbability(int k, int n, BigDecimal p)** Computes the negative binomial probability using BigInteger and BigDecimal.

Here's a Screenshot of the output from the main method



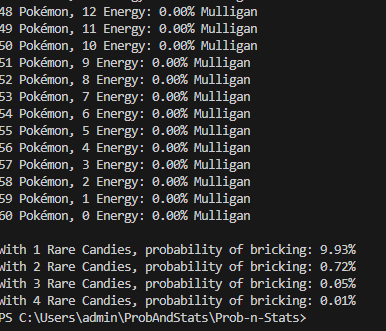
**Then we have the SetOperations, consisting of different methods that all perform some type of operation on a set.**

* **union(ArrayList<Integer> a1, ArrayList<Integer> a2)** Returns the union of two ArrayLists.
* **intersect(List<Integer> list1, List<Integer> list2)** Returns the intersection of two lists.
* **complement(List<Integer> sample, List<Integer> subset)** Returns the complement of the subset in the sample.
* **independentIntersection(double probA, double probB)** Calculates the intersection probability for independent events by multiplying their probabilities.
* **dependentIntersection(double probA, double conditionalProbBGivenA)** Calculates the intersection probability for dependent events.
* **exclusiveUnion(double probA, double probB)** Calculates the union probability for mutually exclusive events by adding their probabilities.
* **nonExclusiveUnion(double probA, double probB, double intersectionProb)**Calculates the union probability for events that are not mutually exclusive by adding the probabilities and subtracting the overlapping probability.

**Next we have our MonteCarlo simulations, including the odds two people share the same birthday in a given room of people, the classic three door game, along with the odds of a mulligan in pokemonTCG given a certain amount of pokemon, and the probability of bricking given a certain amount of rare candies in a deck.**

Heres a Screenshot of the output.

****

****