

# DOCUMENTATION PROJECT 1

## Prerequisites

- Java Development Kit (JDK) installed (version 8 or higher recommended).
- A Java IDE (e.g., IntelliJ IDEa, eclipse) or a command-line environment.

## Pokemon Battle:

The Pokemon Battle is a console based program in Java that models battles between two Pokemon characters. Each Pokemon has attributes like HP, attack, defense, and speed) and can fight in a virtual stadium where turn order is determined by speed, and damage is calculated based on attack and defense stats. The project includes a base Pokemon class, specific Pokemon implementations (Pikachu and Charmander), a Stadium class to manage battles, and a testPokemon class to demonstrate functionality.

This documentation provides an overview of the project, and its components.

---

The project consists of the following Java classes:

### 1. **Pokemon**

- **Purpose:** Base class representing a generic Pokemon with battle statistics and methods.
- **Attributes:**
  - hp (health points): Tracks the Pokemon's health.
  - attack: Determines damage potential.
  - defense: Reduces damage taken.
  - speed: Determines turn order in battles.
- **Methods:** Getters (getHp, getAttack, getDefense, getSpeed), setters (setHp, setAttack, setDefense, setSpeed), and attacks (to deal damage to another Pokemon).

### 2. **Pikachu**

- **Purpose:** A specific Pokemon extending Pokemon with predefined stats.

### 3. **Charmander**

- **Purpose:** A specific Pokemon extending Pokemon with predefined stats.

### 4. **Stadium**

- **Purpose:** Manages a battle between two Pokemon, determining turn order and victory conditions.

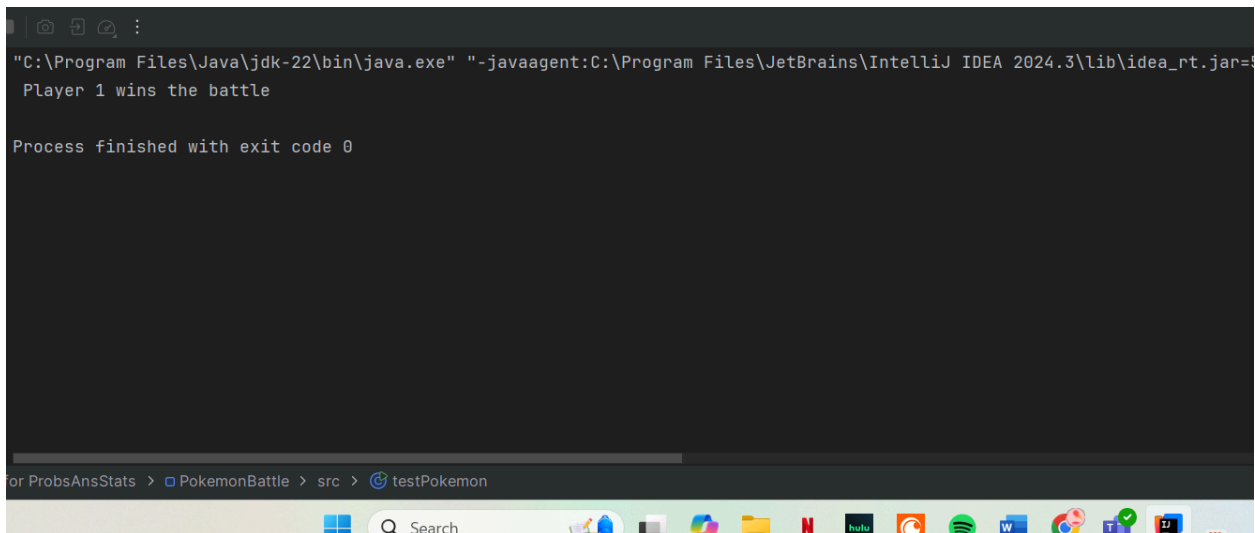
- **Key Method:** battle - Simulates the fight until one Pokemon's HP reaches 0 or below.

## 5. testPokemon

- **Purpose:** A test class with a main method to demonstrate a battle between Pikachu and Charmander.

- 
- each Pokemon has four stats:
    - **HP (Health Points):** Represents remaining health; the Pokemon is knocked out when  $HP \leq 0$ .
    - **Attack:** The power used to deal damage.
    - **Defense:** Reduces incoming damage.
    - **Speed:** Determines which Pokemon attacks first in a turn.
  - The attack method calculates damage as (attacker's attack - defender's defense) and subtracts it from the defender's HP. Note: If attack < defense, the defender's HP increases (see Limitations).
  - Battles occur in the Stadium class:
    - The Pokemon with higher speed attacks first each turn.
    - If the defender survives ( $HP > 0$ ), it counterattacks.
    - The battle continues until one Pokemon's  $HP \leq 0$ .
    - The winner is announced based on remaining HP.
- 

## Output:



```
"C:\Program Files\Java\jdk-22\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2024.3\lib\idea_rt.jar=
Player 1 wins the battle

Process finished with exit code 0
```

for ProbsAnsStats > PokemonBattle > src > testPokemon

## Pokemon Monte Carlo:

The Monte Carlo Pokemon is a Java-based console application that uses Monte Carlo methods to estimate probabilities of Mulligan Probability and Rare Candy Brick in the Pokemon Trading Card Game. The simulation focuses on two key aspects:

1. **Mulligan Probability:** The likelihood of drawing a starting hand with no Pokemon cards, leading to a redraw or mulligan.
2. **Brick Probability:** The chance that all copies of a critical card (Rare Candy) end up in the prize pile, the player is unable to access them which is also known as "bricking".

The project models a simplified Pokemon deck with Pokemon, energy, and Trainer cards, simulating game setup and card draws to calculate these probabilities over multiple trials. This documentation provides an overview of the project, its components, how to use it.

---

### 1. Card

- **Purpose:** Abstract base class for all card types in the game
- **Attributes:**
  - **cardName:** Name of the card.

### 2. Pokemon

- **Purpose:** Represents a Pokemon card with battle-related attributes.
- **Attributes:**
  - **name** inherited from Card class
  - **hp:** Health points.
  - **attackName:** Name of the attack.
  - **weakness:** Type the Pokemon is weak to.

### 3. energy

- **Purpose:** Represents an energy card used to power attacks.
- **Attributes:**
  - **type:** energy type (e.g., "Fire").

### 4. Trainer

- **Purpose:** Represents a Trainer card with a specific effect.
- **Attributes:**
  - **name:** Trainer card name.
  - **attribute:** Description of the card's effect.

### 5. PokemonGame

- **Purpose:** Manages the simulation logic, deck creation, and probability calculations.

- **Attributes:**
  - deck, hand, discard, prize, bench (all List<Card>).
  - random: Random number generator for shuffling and drawing.
- **Methods:** Deck setup, card drawing, mulligan checking, checking Brick prize setting, and simulation methods.

## 6. MainPokemon

- **Purpose:** Test class with a main method to run the simulations and display results.
- **Key Feature:** Runs mulligan and brick simulations with 10,000 trials each.

## Card Classes

- **Pokemon:** Models a Pokemon card (e.g., Charmander) with stats for HP, attack, and weakness.
- **energy:** Models an energy card (e.g., Fire energy) needed for attacks.
- **Trainer:** Models a Trainer card (e.g., Rare Candy) with a name and attribute.

## PokemonGame Class

- **Simulation Logic:**
  - **Deck Setup (Initialize Deck):** Creates a 60-card deck with specified counts of Pokemon, energy, and Trainer (Rare Candy) cards, then shuffles it.
  - **Card Drawing (drawCard):** Draws a specified number of cards from the deck to the hand.
  - **Mulligan Check (hasPokemonInHand):** Returns true if the hand contains at least one Pokemon card, false otherwise.
  - **Prize Setting (setPrize):** Sets aside 6 prize cards from the top of the deck.
  - **Brick Check (checkBrick):** Returns true if all Rare Candy cards are in the prize pile.
- **Mulligan Simulation (calcMulliganPercent):**
  - Tests deck compositions with 1 to 60 Pokemon cards (rest as energy).
  - Draws 7-card hands over trial iterations.
  - Calculates the percentage of hands requiring a mulligan (no Pokemon).
- **Brick Simulation (simulateBrickOdds):**
  - Tests decks with 1 to 4 Rare Candy cards, 10 Pokemon, and the rest energy.
  - Simulates game setup (draw 7, set 6 prizes) with mulligan rule enforcement.
  - Calculates the percentage of games where all Rare Candy cards are prized.

## MainPokemon Class

- **execution:** Runs both simulations with 10,000 trials each and prints results.

## Theoretical Context

- **Mulligan Probability:** Follows a hypergeometric distribution, decreasing as the number of Pokemon in the deck increases.
- **Brick Probability:** Also hypergeometric, representing the odds of drawing all Rare Candy cards into the 6 prize cards.

- Mulligan percentages for Pokemon counts 1 to 60.
- Brick percentages for Rare Candy counts 1 to 4.

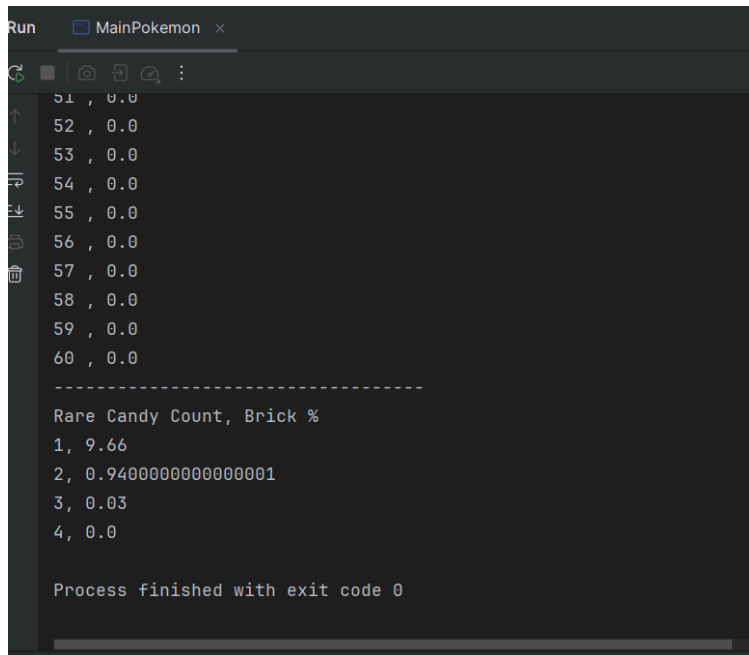
Once simulations are run through MainPokemon, output can be expected to show percentages for results

**Output:**

### The Mulligan Percentages:

```
"C:\Program Files\Java\jdk-22\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2023.3.2\lib\idea_rt.jar=50000:C:\Program Files\JetBrains\IntelliJ IDEA 2023.3.2\bin" -Dfile.encoding=UTF-8
Pokemon Count, Mulligan %
1 , 88.44999999999999
2 , 77.99000000000001
3 , 68.04
4 , 59.13
5 , 53.010000000000005
6 , 45.95
7 , 39.900000000000006
8 , 33.03
9 , 30.680000000000003
10 , 25.96
11 , 22.43
12 , 19.580000000000002
13 , 15.809999999999999
14 , 13.54
15 , 11.450000000000001
16 , 10.09
17 , 8.42
18 , 6.909999999999999
19 , 6.23
20 , 4.859999999999999
21 , 3.9
22 , 3.2099999999999995
23 , 2.35
24 , 2.11
25 , 1.81
26 , 1.58
27 , 1.05
28 , 0.88
29 , 0.54
30 , 0.51
31 , 0.38999999999999996
```

The rare candy Bricking Percentages:



```
Run MainPokemon x
51 , 0.0
52 , 0.0
53 , 0.0
54 , 0.0
55 , 0.0
56 , 0.0
57 , 0.0
58 , 0.0
59 , 0.0
60 , 0.0
-----
Rare Candy Count, Brick %
1, 9.66
2, 0.9400000000000001
3, 0.03
4, 0.0
Process finished with exit code 0
```

## Pokemon Card Game:

Project is inspired by the Pokemon Trading Card Game and uses the terminal for a 2 player simulation. Game is a simplified version and written in Java. Classes include Card class, types of Card like energy, pokemon and trainers. Specific pokemon, trainers and energy cards are also created, game logic is contained in PokemonGame and runs through Run.

- 
1. **Players:** Two players take turns.
  2. **Cards:** Three types of cards :
    - **Pokemon:** Combat units with HP, type, energy attached, weakness, and retreat cost.
    - **energy:** Used to power Pokemon for attacks or retreating, includes basic energy as well
    - **Trainer:** Special cards with one-time effects (e.g., drawing cards,HP increase).

### 3. The player decks:

- **Deck:** Main card pile for each player (shuffled at start).
- **Hand:** Cards players can play.
- **Prizes:** Six cards set aside; taking all prizes wins the game.
- **Discard Pile:** Used cards go here.
- **Active Pokemon:** One Pokemon in play, used for attacking.
- **Bench:** Up to five reserve Pokemon.

### Win Conditions

- **Prize Cards:** A player wins by taking all 6 of their opponent's prize cards (earned by knocking out Pokemon).
- **No Pokemon or cards:** A player wins if the opponent has no active or benched Pokemon left or cards left.

### How game works

1. **Draw Phase:** Draw 1 card from the deck.
2. **Action Phase:** Players can (in any order, with some restrictions):
  - Attach 1 energy card to their active Pokemon .
  - Play Pokemon to the bench (up to 5).
  - Play 1 Trainer card (once per turn).
  - Retreat the active Pokemon (if energy cost is met).
  - Attack if energy is attached(ends the turn).
  - Pass (ends the turn without attacking).
3. Check for win conditions; switch turns.
  - **Attack:** Costs 1 energy; deals 10 base damage.
  - **Weakness:** Damage doubles if the attacking Pokemon's type matches the defender's weakness.
  - **Knockout:** If a Pokemon's HP reaches 0, it's discarded, and the opponent takes a prize card.
  - **Special effects:** Trainer cards like ProfessorelmsAdvice can double damage for one attack.

---

### Abstract Base Class

#### Card

- **Purpose:** Abstract parent class for all card types.
- **Fields:**

- protected String type: The card type (e.g., "Pokemon", "energy", "Trainer").
- **Methods:**
  - public String getType(): Returns the card's type.

## Card Subclasses

### Pokemon

- **Purpose:** Represents Pokemon cards with combat stats.
- **Fields:**
  - name: Pokemon's name (e.g., "Pikachu").
  - pokeType: elemental type (e.g., "electric").
  - hp: Hit points.
  - energyAttached: Number of attached energies.
  - weakness: Type it's weak to.
  - retreatCost: energy needed to retreat.
- **Constructor:**
  - Pokemon(String name, int hp, String pokeType, int retreatCost, String weakness)
- **Methods:**
  - Getters and Setters

### Energy

- **Purpose:** Represents energy cards to power Pokemon.
- **Fields:**
  - energyType: Specific type (e.g., "electric", "Fire").
- **Constructor:**
  - energy(String energyType)
- **Methods:**
  - Getters and Setters

### Trainer

- **Purpose:** Represents trainer cards with special effects.
- **Fields:**
  - name: Trainer card name (e.g., "Bill").
  - ability: Description of the effect (e.g., "Draw 2 Cards").
- **Constructor:**
  - Trainer(String name, String ability)
- **Methods:**



- getName(): Returns the name.
- getAbility(): Returns the ability description.
- toString(): Returns "Bill (Draw 2 Cards)".

## Specific Card Classes

- **Pokemon:** Pikachu, Squirtle, Charmander, Torchic, Shinx
  - each extends Pokemon with predefined stats (e.g., Pikachu: 60 HP, electric, 1 retreat, ).
- **energy:** Basic energy, electric, Fire, Water
  - each extends energy with a specific type.
- **Trainer:** Bill, Hero, Ice, ProfessorelmsAdvice, ProfessorsResearch, Psychic
  - each extends Trainer with a name and ability (e.g., Bill: "Draw 2 Cards").

## Main Game Logic

### PokemonGame

- **Purpose:** Manages the game state and flow.
- **Fields:**
  - Player-specific: deckPlayer1, deckPlayer2, handPlayer1, handPlayer2, prizesPlayer1, prizesPlayer2, discardPlayer1, discardPlayer2, benchPlayer1, benchPlayer2, activePlayer1, activePlayer2.
  - Game state: prizeCountPlayer1, prizeCountPlayer2, player1Turn, gameWon.
  - Utilities: random, scanner, trainerPlayedThisTurn, ProfessorelmsAdviceActivePlayer1, ProfessorelmsAdviceActivePlayer2.
- **Key Methods:**
  - initializeDecks(): Populates and shuffles decks with cards.
  - setupGame(): Sets up initial hands, prizes, and active/bench Pokemon.
  - playerTurn(): Handles a single player's turn logic.
  - applyTrainereffect(): executes trainer card effects.
  - handleKnockout(): Manages Pokemon knockouts and prize collection.
  - checkGameend(): Checks win conditions.
  - run(): Main game loop.
  - Helpers: drawCard(), printGameState(), getCurrentPlayer(), etc.

### Run

- **Purpose:** entry point to start the game.
- **Method:**
  - main(): Initializes and runs PokemonGame.

---

## Game Flow

### 1. Initialization:

- Decks are populated with Pokemon (Pikachu, Squirtle, Charmander, Torchic, Shinx), energy (electric, Water, Fire, Basic), and Trainers (Bill, Hero, Professor Elms Advice, ProfessorsResearch).
- each player draws 7 cards, reshuffling if no Pokemon are drawn.
- 6 prize cards are set aside per player.
- Players choose an active Pokemon and optionally bench others.

### 2. Gameplay Loop:

- Players alternate turns.
- each turn starts with drawing a card, followed by actions (energy, Pokemon, trainer, attack, retreat, or pass).
- Attacking ends the turn; knockouts trigger prize collection and bench replacement.

### 3. endgame:

- Game ends when all prizes are claimed or a player has no Pokemon left in play t or no more cards left to draw.

---

## Implemented Features

- Card types with inheritance (Pokemon, energy, Trainer).
- Turn-based gameplay with action limits.
- Combat with attacks, weakness and knockouts.
- Trainer effects (e.g., drawing cards, boosting HP, doubling damage).
- Retreat mechanic with energy cost.
- Bench management (up to 5 Pokemon).

---

○

```
Run Run x
Choose action (1: Play Energy, 2: Play Pokémon, 3: Play Trainer, 4: Attack, 5: Retreat, 6: Pass):
4
Pikachu attacked Shinx for 10 damage!
Shinx was knocked out!
Player 2 takes a prize card! 5 prizes remaining for Player 1

=== Game State Update ===
Current Player: Player 2

Player 1's State:
Hand: [Fire Energy, Water Energy, Water Energy, Basic Energy, Basic Energy, Charmander (Fire, HP: 50), Fire Energy]
Active Pokémon: None
Bench: []
Prizes left: 6
Deck size: 42
Discard pile: 5 cards

Player 2's State:
Hand: [Shinx (Electric, HP: 50), Water Energy, Basic Energy, Fire Energy, Shinx (Electric, HP: 50), Basic Energy, Water Energy, Bill (Draw 2 Cards), Torchic (Fire, HP: 50)]
Active Pokémon: Pikachu (HP: 10, Energy: 2)
Bench: []
Prizes left: 5
Deck size: 42
Discard pile: 0 cards

=====

Player 2 wins - Player 1 has no Pokémon left!
Game Over! Thanks for playing!
```

## Monte Carlo Door:

The Monte Carlo Monty Hall Door Simulation is a Java-based console application that uses a Monte Carlo method to estimate the probability of winning a prize in the Monty Hall problem under two strategies: switching doors or sticking with the initial choice. The Monty Hall problem is a classic probability puzzle where a contestant chooses one of three doors, one hiding a prize, and after a host reveals a non-prize door, the contestant can switch their choice.

### 1. Door

- **Purpose:** Represents a single door in the Monty Hall problem, which may or may not hide the prize.
- **Attributes:**
  - **hasPrize:** A boolean indicating whether the door conceals the prize.
- **Methods:** Constructor to set the prize status.

## 2. PlayGame

- **Purpose:** Manages the simulation logic for the Monty Hall problem, including single-game execution and multi-trial statistics.
- **Attributes:**
  - random: A static Random instance for generating random choices.
- **Methods:** playGame for one trial, and simulate for multiple trials.

## 3. DoorSimulation

- **Purpose:** Test class with a main method to run the simulation and display results.
  - **Key Feature:** Runs simulations for both switching and not switching strategies with 1,000,000 trials each.
- 

### Door Class

- **Behavior:** A simple data structure representing a door with a boolean (hasPrize) to indicate if it has prize or not

### PlayGame Class

- **Multiple Trials (simulate):**
  - Runs playGame for the specified number of trials.
  - Calculates and returns the win percentage as a double (0-100).
- **Algorithm:** Uses random number generation to mimic player and host decisions, adhering to Monty Hall rules.

### DoorSimulation Class

- **execution:** Instantiates a PlayGame object and runs two simulations:
    - One with switchDoor = false (no switching).
    - One with switchDoor = true (switching).
  - **Default Parameters:** 1,000,000 trials for each strategy.
-

**Output:**

```
"C:\Program Files\Java\jdk-22\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2023.3\lib\idea_rt.jar" 1600000000000
Win rate without switching: 33.2795%
Win rate with switching: 66.6049%

Process finished with exit code 0
```

### Monte Carlo Birthday:

The Monte Carlo Birthday simulation estimates the probability that at least two people in the same class share the same birthday. The simulation assigns random birthdays to a specified number of people across multiple trials and calculates the percentage of trials where a birthday collision occurs.

This documentation provides an overview of the project, its components, how to use it.

## Project Structure

The project consists of the following Java classes:

## 1. Person

- **Purpose:** Represents an individual with a randomly assigned birthday.
- **Attributes:**
  - birthday: An integer from 1 to 365 representing the day of the year.
- **Methods:** Constructor to assign a random birthday, and getBirthday to retrieve it.

## 2. Birthday

- **Purpose:** Manages the Monte Carlo simulation to calculate the probability of shared birthdays.
- **Attributes:**
  - **people:** Number of people in the group.

- trials: Number of simulation trials.
  - **Methods:** runTrials to compute the probability, and sameBirthday to simulate one trial.
3. **RunBirthday**
- **Purpose:** Test class with a main method to run the simulation and display results.
  - **Key Feature:** Runs a simulation for 33 people over 10,000 trials by default.
- 

## Person Class

- **Behavior:** each Person object is assigned a random birthday between 1 and 365 (inclusive) upon creation, simulating a day of the year in a non-leap year.
- **Purpose:** Provides a simple abstraction for individuals in the simulation.

## Birthday Class

- **Simulation Logic:**
  - **Single Trial (sameBirthday):** Creates a group of people, assigns each a random birthday, and checks for duplicates. Returns true if at least two people share a birthday, false otherwise.
  - **Multiple Trials (runTrials):** Runs the sameBirthday method trial times and calculates the percentage of trials with at least one shared birthday.
- **Algorithm:** Uses an array to track birthday occurrences, incrementing a counter for each birthday assigned. If a birthday's counter exceeds 1, a collision is detected.

## RunBirthday Class

- **execution:** Instantiates a Birthday object with 33 people and 10,000 trials, runs the simulation, and prints the resulting probability as a percentage.

## Output

```
"C:\Program Files\Java\jdk-22\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2
Probability two people in class share a birthday: 77.14%

Process finished with exit code 0
```

## Stats Library

### Overview

A Java utility library for statistical analysis, probability calculations, and set operations.

### Key Components

- **StatsLibrary**: Statistical and probability methods
  - Descriptive statistics (mean, median, mode)
  - Probability distributions (binomial, geometric, negative binomial, hypergeometric.)
  - Probability laws
- **SetOperations**: Union and intersection of integer lists
- **ProbPC**: Factorial, permutation, and combination calculations

---

The project consists of the following Java classes:

1. **StatsLibrary**

- **Purpose:** Core class containing statistical, probability, and distribution-related methods.
- **Key Features:** Mean, median, mode, standard deviation (sample and population), probability laws, discrete probability distributions, expected values, and variances.

2. **SetOperations**

- **Purpose:** Handles basic set operations (union and intersection) on `ArrayList<Integer>` objects.
- **Key Features:** Computes the union and intersection of two integer lists.

3. **ProbPC**

- **Purpose:** Calculates permutations and combinations using `BigInteger` for large numbers.
- **Key Features:** Factorial, permutation, and combination computations with support for large values.

4. **PCtester**

- **Purpose:** Test class for `ProbPC`, showcasing permutation and combination calculations.

5. **testSetOperations**

- **Purpose:** Test class specifically for `SetOperations`.

6. **testStatsLibrary**

- **Purpose:** Test class specifically for `StatsLibrary` with a custom dataset.
- 

## Functionality

### 1. StatsLibrary

#### Descriptive Statistics

- **Mean:** Average of an integer array (`getMean`).
- **Median:** Middle value of a sorted integer array (`getMedian`).
- **Mode:** Most frequent value in an integer array (`getMode`).
- **Standard Deviation:** Sample (`getSampleStandardDeviation`) and population (`getPopulationStandardDeviation`) measures of dispersion.

#### Probability Laws



- **Multiplicative Law (Independent events):**  $P(A \cap B) = P(A) * P(B)$  (multiplicativeLawIndependent).
- **Additive Law (Mutually exclusive events):**  $P(A \cup B) = P(A) + P(B)$  (additiveLawMutuallyexclusive).
- **Additive Law (General):**  $P(A \cup B) = P(A) + P(B) - P(A \cap B)$  (additive Law).
- **Conditional Probability:**  $P(A|B) = P(A \cap B) / P(B)$  (conditionalProbability).
- **Bayes' Theorem:**  $P(A|B) = P(B|A) * P(A) / P(B)$  (bayesTheorem).

## Combinatorial Functions

- **Factorial:** Computes  $n!$  (factorial).
- **Combination:**  $C(n, r)$  (combination).
- **Permutation:**  $P(n, r)$  (permutation).

## Probability Distributions

- **Binomial:** Probability mass function (binomialPMF), expected value (binomialexpectedValue), variance (binomialVariance).
- **Geometric:** Probability of first success on the  $k$ -th trial (geometricProbability), expected value (geometricexpectedValue), variance (geometricVariance).
- **Negative Binomial:** Probability of  $r$ -th success on the  $x$ -th trial (negativeBinomial), expected value (negativeBinomialexpectedValue), variance (negativeBinomialVariance).
- **Hypergeometric:** Probability of  $k$  successes in  $n$  draws without replacement (hypergeometric), expected value (hypergeometricexpectedValue), variance (hypergeometricVariance).

## 2. SetOperations

- **Union:** Combines two integer lists, removing duplicates (SetUnion).
- **Intersection:** Finds common elements between two integer lists (SetIntersect).

## 3. ProbPC

- **Factorial:** Computes  $n!$  using BigInteger (factorial).
- **Permutation:**  $P(n, r)$  for large values (permutation).
- **Combination:**  $C(n, r)$  for large values (combination).
- **PCtester:** Tests ProbPC with large  $n$  and  $r$  values (e.g.,  $P(100, 30)$ ).
- **testSetOperations:** Demonstrates SetOperations with custom lists.
- **testStatsLibrary:** Tests StatsLibrary with a custom dataset.

## Output

```
"C:\Program Files\Java\jdk-22\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ
Mean: 2.4
Median: 2.0
Mode: 2.0
Sample Std Dev: 1.140175425099138
Population Std Dev: 1.019803902718557
C(5,2) = 10
P(5,2) = 20
P(A and B) = 0.12
P(A or B) = 0.7
P(A or B)(P intersect B) = 0.6
Conditional P(A|B) = 0.3
Binomial P(Y=2) = 0.3086999999999999
Geometric P(Y=2) = 0.21
Negative Binomial: 0.1875
Hypergeometric: 0.47619047619047616
Binomial E[X] = 1.5
Binomial Var[X] = 1.0499999999999998
Geometric E[X] = 3.3333333333333335
Geometric Var[X] = 7.777777777777778
Negative Binomial E[X] = 6.0
Negative Binomial Var[X] = 6.0
Hypergeometric E[X] = 2.0
Hypergeometric Var[X] = 0.6666666666666666

Process finished with exit code 0
|
```

```
"C:\Program Files\Java\jdk-22\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\
Our union result: [1, 2, 3, 4, 5, 6]
Our intersection result: [3, 4]

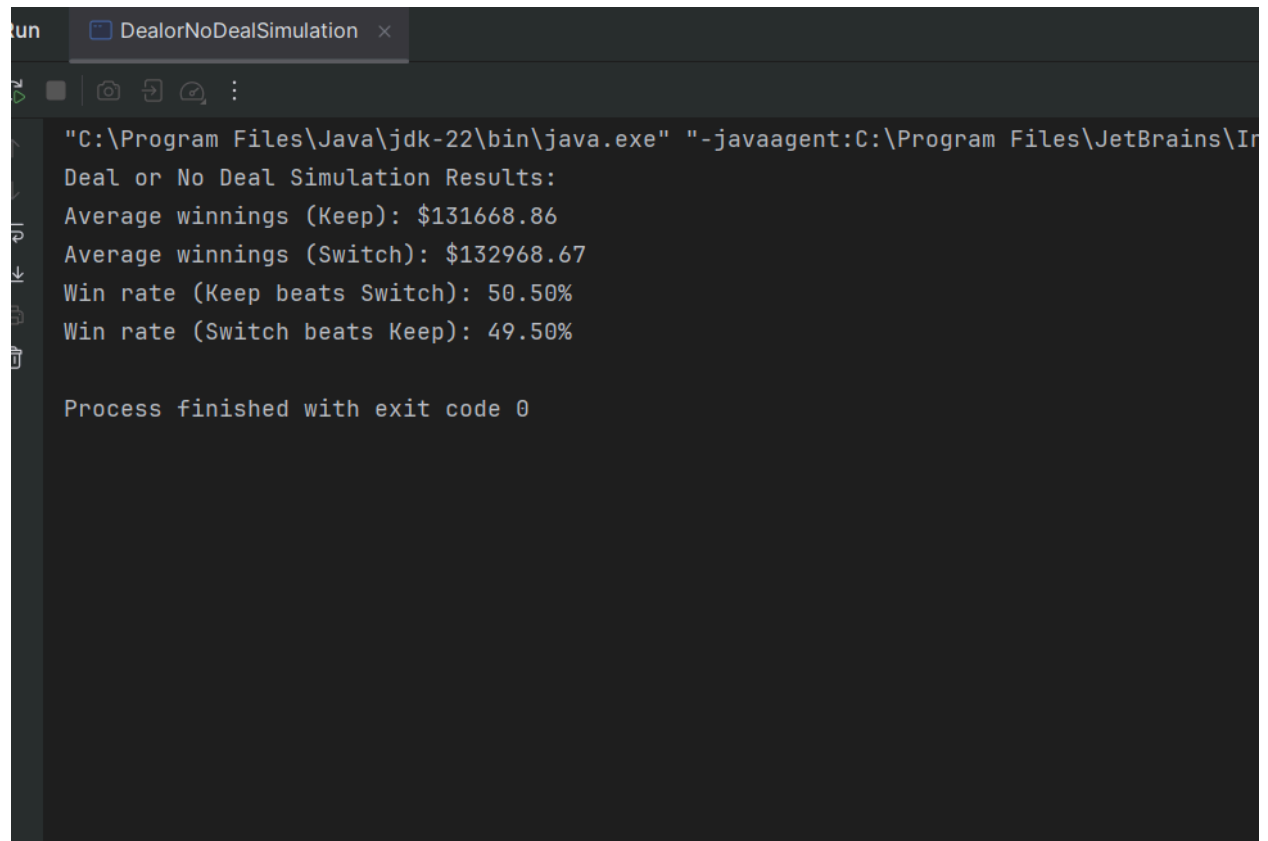
Process finished with exit code 0
```

```
"C:\Program Files\Java\jdk-22\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2024
Permutation: 7791097137057804874587232499277321440358327700684800000000
Combination: 29372339821610944823963760
```

```
Process finished with exit code 0
```

\*Extra Credit Deal or No Deal Game added using Monte Carlo

Output:



```
run DealorNoDealSimulation x
"C:\Program Files\Java\jdk-22\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\I
Deal or No Deal Simulation Results:
Average winnings (Keep): $131668.86
Average winnings (Switch): $132968.67
Win rate (Keep beats Switch): 50.50%
Win rate (Switch beats Keep): 49.50%

Process finished with exit code 0
```