1 Flip

Flip is a 2-player ”strategic change-making game” published by Cheapass Games.

Each player starts with five dice. Roll the dice, and the player with the lowest total goes first. The players then alternate turns.

On each turn, the player either flips one of their own dice or plays one of their opponent's dice. However, players may *not* flip the same die twice without first playing one of the opponent's dice.

- *Flipping* means to turn the die to the opposite side of what is showing; since the top and bottom numbers of 6-sided dice always sum to 7, this means that the new value of a flipped die is 7 minus the old value. (6 flips to 1, 5 flips to 2, 4 flips to 3, 3 flips to 4, etc.)
- *Playing* means to put one of the opponent's dice into the middle of the table. They can then remove any number of dice whose total value does not exceed the value of the die played. For example, if a 6 is played, up to 5 points worth of dice can be removed from the middle.

The game continues until one player runs out of dice. The winner is the player with dice remaining, and the points scored is the sum of their dice.

Reset and continuing playing games until a player reaches 50 points.

Your program should display sufficient output so that the players can follow and play the game. In particular, the current game state (player 1's dice, player 2's dice, the dice in the middle, and which dice are flippable or not) should be displayed at the beginning of each turn, the winner and the current point totals for both players should be displayed at the end of each game, and the overall winner should be congratulated when 50 points is reached.

Your program should also perform appropriate error-checking and enforce the game rules. For example, players shouldn't be able to pick a non-existent die to flip or play, and players may not flip an unflippable die or remove too many points' worth of dice from the middle.

2 OOAD

For an object-oriented design for Flip, we begin with textual analysis of the game description above. In particular, we're looking for nouns that correspond to things that will need to be represented in the program. What comes up?

- The notion of a player: *Player*, *opponent*, and *winner* refer to different instances of the players participating in the game.

- The notion of a collection of dice: *Five dice*, *the dice*, *their own dice*, *opponent's dice*, *middle of the table* refer to different collections of dice.
- The notion of a single die: *Die* and *one of their own dice* both reference a single die, plus whenever there is a collection of something it is worth also considering the single thing.
- The notion of a score: *Score* is implied by "a player reaches 50 points".

There are other nouns, but not that refer to things in their own right: *turn* really refers to the process of taking a turn, *total value* refers to a property of something (and is just a number), *any number of dice* is specifying a quantity.

We begin our design:

class	**purpose**	**stored info**	**methods**
Player	a player in the game		
DiceCollection	a collection of dice		
Dice	a single 6-sided die		

"Score" was omitted because it is just a number. About `Dice` and `DiceCollection`: Normally singular names should be used for single things and plural names for collections of things, but "dice" is often treated specially because the singular "die" has another meaning in English and saying things like "Die class" is a bit awkward...

Now consider stored information and methods — repeat the textual analysis of the game description, looking for clues about what's important to keep track of for each class (stored information) and for verbs and operations that involve instances of the classes identified (methods). Be careful to think about the meaning of the text and how an operation impacts all of the objects and classes identified and not only focus on the exact words on the page. What comes up?

- *starts with five dice* → constructor for `DiceCollection` which initializes the collection with five dice
- *roll the dice* → roll a collection of dice
- *player with the lowest total goes first*, *points scored is the sum of their dice* → get total for a collection of dice
- *their own dice*, *their opponent's dice* → a player has a collection of dice
- *player either flips one of their own dice...may not flip the same die twice without first playing...* → flip a particular one of a collection of dice, check if a particular die is flippable, reset all dice to flippable

- *what is showing* → the current value on a die
- *top and bottom numbers of 6-sided dice always sum to 7* → the number of sides and the configuration of the faces of the die are necessary properties of individual dice, but those properties are fixed for this game so can be hardcoded. (Dice are common commodities, and one might imagine another game or an extension to this game where other-than-6-sided dice are used — that might justify the extra effort to make the `Dice` class more flexible now.)
- *plays one of their opponent's dice*, which is described as *put one of the opponent's dice in the middle of the table...remove any number of dice whose total value does not exceed the value of the die played* → add a die to the middle collection, remove dice from the middle collection, get value of a die, get total value of one or more dice as well as remove a die from the opponent's collection, add dice to the opponent's collection
- *until one player runs out of dice...player with dice remaining* → get the number of dice in a collection or determine whether or not a collection of dice is empty (either operation can be used to determine if a player is out of dice)
- *points scored is...* → add to player's score
- *until a player['s score] reaches 50 points* → player has a score
- *should display sufficient output...player 1's dice, player 2's dice...winner and current point totals for both players...overall winner congratulated* → needing to distinguish between the two players for informative prompts during gameplay means a player name or other identifier; since "player 1" and "player 2" are referenced, this can take the form of a player number rather than a name
- *should display...player 1's dice, player 2's dice, the dice in the middle, and which dice are flippable or not* → print dice collection
- *should display...winner and current point totals* → get player's identifier and score

This analysis leads to the table in figure 1. When assigning methods to a class, keep in mind that methods belong in the class being manipulated e.g. in *the player rolls the dice*, the "rolls" operation is manipulating the dice and so should belong to the dice collection, not the player.

Some notes and observations:

- The methods identified in figure 1 focus on what operations the classes need to have in order to support the game logic. That is by design — carrying out game logic, such as the series of steps involved in a "play" turn, and enforcing game rules, such as making sure the opponent doesn't remove too many dice from the middle, should be the job of the main program rather than the dice collection. (`DiceCollection`'s purpose is "a collection of dice" — knowing how to play Flip

class	**purpose**	**stored info**	**methods**
Player	a player in the game	player's dice (collection) score id (number)	get score update score (add points) get id
DiceCollection	a collection of dice	the dice in the collection	create with 5 dice roll dice get total sum of dice values flip a particular die is a particular die flippable? reset all dice to flippable add a particular die, add multiple dice remove a particular die, remove multiple dice get dice count (number of dice in the collection) print
Dice	a single 6-sided die	current value been flipped?	get value is flippable? flip die reset (make flippable again)

Figure 1: Flip class design #1.

is not part of that purpose.) This allows `DiceCollection` to be simpler, more modular, and more reusable — it only has to manage the collection, it's not affected if game play or rules change (and game logic and rules can be limited to the main program instead of spread over many classes), and it can potentially be reused in new variants of Flip.

- Being able to add or remove one die at a time is sufficient here — it's always possible to add/remove multiple dice by writing a loop to repeat the add/remove single die operation. On the other hand, not having to write that loop repeatedly can be convenient — though there's also the complicating factor of how to provide a varying number of things as parameters so actually using an add/remove multiple dice method might not be all that convenient after all. In the end, implement the add/remove multiple dice versions if they help make your code simpler and omit them if not.

At this point we have a usable class design. `Player`, `DiceCollection`, and `Dice` should be the only new classes we need (other than a class for the main program), and the textual analysis of the game description plus careful thought about how the various game actions impact objects of those types should mean that the major methods needed have been identified. There are likely to be some supporting methods missing, such as constructors and getters. That's OK — adding those as you start writing code isn't likely to cause major upheaval for the overall design.

3 Alternative Class Designs

Observe that all of the interaction with dice goes through `DiceCollection` and that since the dice themselves are actually interchangeable (if there are two flippable 4s, it doesn't matter which is the one flipped) and flippable status matters only within a player's dice collection, it's not actually necessary to have `Dice` objects — "a particular die" can be specified by position in the collection or by its value. This leads to an alternative design, shown in figure 2. (Note that since there is no longer a need for a class representing a single die, the plural name `Dice` has been taken for the collection.)

class	**purpose**	**stored info**	**methods**
Player	a player in the game	player's dice (collection) score id (number)	get score update score (add points) get id
Dice	a collection of dice	the dice in the collection and their flippable status	create with 5 dice roll dice get total sum of dice values flip a particular die is a particular die flippable? reset all dice to flippable add a particular die, add multiple dice remove a particular die, remove multiple dice get dice count (number of dice in the collection) print

Figure 2: Flip class design #2.

4 Implementation

Both designs need to store a collection of dice. In design #1, `DiceCollection` stores a collection of `Dice` objects. Since the number of dice in a collection can vary over the course of the game, use an `ArrayList` rather than an array for the instance variable in `DiceCollection`. For design #2, the observation that what matters is the number of dice of each value and flippable status rather than the individual dice themselves can be used: keep track of the dice using an array of counts where slot i stores the number of dice with value i. Use separate arrays for the counts of flippable and not flippable dice.

Both designs also need a way to refer to "a particular die". This can be by position in the collection, by value showing on the die, or (in design #1) via the `Dice` object itself. Consider what is convenient. For example, for "flip a particular die", the die to flip is chosen by the user. How are you having them specify the die? It can make sense to carry that same thing over to the method's parameters. For adding and removing dice, you'll need the `Dice` objects themselves in design #1 (so you might need to add getters to retrieve a `Dice` object given a position or value) but can work with the dice values in design #2.