Flip Individual Feedback Notes

- not every instance of something is marked in the individual feedback
  - don't stop with addressing only what is indicated – consider whether those things apply elsewhere in the program
- make sure comments are included
  - a class comment with a brief description of the program and your name
  - comments for each method describing its task, what its parameters are for, and what is returned
  - use Javadoc style
  - individual feedback did not address comments
- be sure to thoroughly test your program
  - bugs were noted as noticed, but individual feedback focused on the bigger picture of addressing specifications and program construction

Flip Individual Feedback Notes

- review your program for user friendliness
  - feedback addresses only the minimum requirement that there is enough output for players to be able to follow and play the game
  - how easy is it for users to get the information they need?
    - whose turn it is
    - what they need to decide on a legal and strategic move – their dice vs their opponent's dice, which of their dice are flippable, …
  - how easy it is for users to specify their moves and other input? how closely does it match with what the user is thinking, and how much do they have to type?

Flip – Rules and Game Play

- a player cannot flip the same die twice without playing
  - a player can choose to flip several turns in a row, but a different die must be chosen each time
- the player takes back dice from the middle after one of their dice has been played
  - dice are moved from the middle to that player's collection, not simply removed from the middle
- the player whose dice was played should pick the dice they want to remove from the middle
  - the program should not do this automatically
- a player is not required to take back dice from the middle after one of their dice has been played, even if there are dice they could take
  - (not taking dice might be a poor strategy, but poor strategy isn't against the rules)

Flip – Game Play

- it is not necessary to detect whether or not a move is possible as long as there's a way for the player to back out if they've chosen an impossible option
  - e.g. a player without any flippable dice at the start of a turn – they can still be presented with the option to flip or play, but when prompted to pick a die to flip, there should be a way to pick "no die" and instead choose play
  - but it is OK to prevent these situations if you want (it's a nice feature)

Flip – Robustness

For satisfactory mastery –

- ensure that only legal moves are made and the program doesn't crash if the user enters a bad value

Flip – Robustness

For proficient or outstanding mastery –

- handle the possibility of there not being a legal value to enter
  - players should not be able to get stuck in impossible situations
  - e.g. a player chooses to flip, but when prompted for the die to flip, discovers that they have no flippable dice...and the prompt requires a legal die to flip
    - either don't present the option to flip if there are no flippable dice, or have a way for the player to indicate "no die" and return to the choice to flip or play
- avoid harsh consequences for invalid input
  - e.g. forfeiting a turn
  - printing an error message is a good start, but the player should generally be given a chance to correct the problem – reprompt instead of moving on

Flip – Robustness

A strategy for handling robustness for input –

- start by assuming correct usage – get the program working without error-checking user input
- create a "get the value" function for each place where user input is read and checking is needed
  - move the code that prints the prompt and reads the value to the function
  - return the value read
- turn the functions into "get a legal value" functions
  - decide on how the function will indicate "not possible to get a legal value"
  - add error-checking (and reprompting) to the function
  - update the calling code to handle "not possible to get a legal value" getting returned

Flip – Code Organization

- use subroutines to increase readability and untangle distinct tasks
  - keep subroutine bodies short (including for `main`)
  - consider a subroutine for a task with complex logic – more than a few lines, tricky or non-obvious reasoning – especially if that is done more than once
  - avoid combining several distinct tasks that happen to involve the same loop into a single loop
    - e.g. adding up the players' dice

```
int sum1 = 0, sum2 = 0;
for ( int i = 0 ; i < 5 ; i++ ) {
  sum1 += dice1[i];
  sum2 += dice2[i];
}
```

- recognizing that this is really two tasks summing player 1's dice and summing player 2's dice – also makes it easier to see how a `sum` function can be used to improve readability and avoid repeated code

```
int sum1 = 0;
for ( int i = 0 ; i < 5 ; i++ ) {
  sum1 += dice1[i];
}
int sum2 = 0;
for ( int i = 0 ; i < 5 ; i++ ) {
  sum2 += dice2[i];
}
```

```
int sum1 = sum(dice1),
    sum2 = sum(dice2);
```

Flip – Code Organization

- use parameterized subroutines and a current player variable to avoid repeated code
  - avoid repeating taking-a-turn steps for each player
    - the only difference between player 1's turn and player 2's turn is who is considered the player and who is considered the opponent

      e.g. `private void takeTurn ( int[] player, int[] opponent, int[] middle ) { … }`

      then call `takeTurn(dice1,dice2,middle)` and `takeTurn(dice2,dice1,middle)`
  - avoid repeating the sequencing of turns for each possibility for who goes first
    - structure the main game play loop as repeating "do current player's turn, update current player" instead of repeating "do player 1's turn, do player 2's turn" (or "do player 2's turn, do player 1's turn")

Flip – Code Organization

- subroutines should have well-defined tasks
  - subroutines should generally do only one job: get input, compute or do something, or produce output
    - e.g. print dice, sum dice
  - avoid subroutines that simply group a series of lines of code
    - a red flag is a name like "start round" or "finish round" – if there's not a name which identifies the subroutine's task more specifically, it's not a well-defined task

Flip – Defensive Programming

- consider functions to compute values instead of storing redundant information
  - e.g. `while ( numdice(dice1) > 0 || numdice(dice2) ) { … }`

    instead of storing both the players' dice (`dice1`, `dice2`) and separate `numdice1`, `numdice2` variables
  - this avoids the bug of forgetting to update one value when the other changes
  - there is a tradeoff in terms of running time – it takes more time to go through an array to compute a value than to look up a stored value – but for this application that time is not significant

Flip – Programming Style

- prefer local variables in `main` to global `static` variables
  - prefer passing values to subroutines via parameters
  - prefer returning values instead of setting global variables

Flip – Representation

- the handout described storing counts instead of the dice values themselves

An important part of this program is keeping track of the dice and their status (flippable or not). A useful observation is that two dice showing the same value are interchangeable — all you need to keep track of is how many of each value a player has or are in the middle — e.g. player 1 might have one 1, two 3s, and one 6. Clever use of arrays can make this convenient. (Think about how this might work, then see section 3.8.3 in the textbook if you are stuck.) To keep track of flippable vs not flippable dice, consider having two arrays for each player — one keeping track of that player's flippable dice and one for the unflippable dice.

  - use slot *i* of the array to store the count for dice with value *i* – so slot 1 stores the 1s, slot 2 stores the 2s, etc
    - slot 0 is unused
    - this avoids having to always remember -1 or +1 when going between dice values and array indexes – a common source of bugs
  - since it was in “hints and suggestions” rather than a list of requirements, it is OK if you solved the problem another way (arrays, `ArrayList`)
  - make sure you read handouts carefully, and ask if there are instructions or information you don’t understand

Flip – Scanner

- only create one `Scanner` object for `System.in`, not one each time you need to read input
  - OK for this to be a global `static` variable (initialized in `main`), or pass it as a parameter to subroutines that need input
- Scanner can read input of various types
  - e.g. `scanner.nextInt()`, `scanner.nextBoolean()`
  - use `scanner.nextLine()` (only) when you specifically need a line of text
  - caveat: only `scanner.nextLine()` consumes the trailing newline that comes when the user presses Enter after their input
    - defensive programing – get in the habit of always including `scanner.nextLine()` after another use of scanner when reading user input

Flip – Other

- use the default package
  - Eclipse’s “create class” dialog tries to put classes in a package – delete the suggested entry there
  - if your classes aren’t under “(default package)” within `src`, delete the package … line at the beginning of the file and accept Eclipse’s error fix to move it to the default package