Lab 3

- Add to the class comment, constructor comments, and public method comments to reflect that `RomanNumeral` is intended to support Roman numbers in standard form as described in the Wikipedia link above.

- the class comment should reference that this class represents a Roman numeral in standard form
- the constructor that takes a string as a parameter should specify that the string be in standard form
- `toString`'s return value is a string in standard form

- Add preconditions for the public constructors and methods as appropriate. Some preconditions may already have been identified in the previous step, but also be sure to consider limits on valid values for any parameters. Don't forget to consider values like `null` or an empty string. (Are those valid?) "Add" means to include a statement of the precondition in the relevant comment.

- for all types – are any values of that type legal?
  - standard form can only represent values 1-3999
  - `String` parameter to constructor must be in standard form
- for object types – what about `null`?
- for `String`s – what about the empty string?

Lab 3

- Check the preconditions identified — write code! For complex checks like "in standard form", first break the check down into simpler pieces — what properties does a Roman numeral in standard form have? (e.g. What symbols can appear? What order do they appear in? What rules are there for consecutive symbols?) Then write a descriptively-named private helper method to do each check. For now those helper methods can be placeholders and simply return true.

- preconditions for public constructors/methods are checked with an `if` – throw an `IllegalArgumentException` if violated
  - violated precondition is the fault of other code, not this method

- Add postconditions for the public methods as appropriate. "Add" means to include a statement of the postcondition in the relevant comment.

- consider return values – is any value of that type possible as a return?
  - `toString()` returns a standard form representation of the Roman numeral
  - `toInt()` returns an integer in the range 1-3999
- check with assertions – violated is the fault of this method's code

Lab 3

- State this class invariant: add a comment to the declaration of `numeral_` reflecting the intention (and assumption) that it is in standard form.

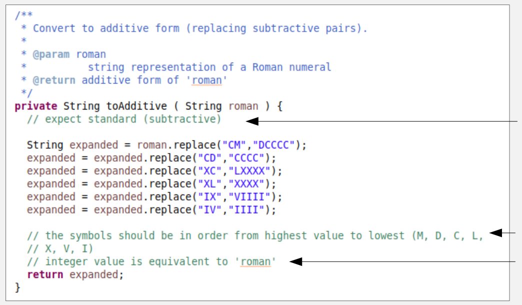
- do that!

- Check this invariant at the end of every constructor and public method — write code! As with the preconditions, break complex checks into simpler properties that can be checked and write descriptively-named private helper methods for each simpler properties. These helpers can be placeholders that simply return true.

- check class invariants with assertions – if they are violated, it is this method's fault
- needed for the `int`-parameter constructor and `addTo`
- can be omitted for the `String`-parameter constructor if the precondition is checked, though checking the class invariant does verify that the instance variable was initialized
- unnecessary for `toString()`, `toInt()`, `add` as they do not change `numeral_`

Lab 3

- Add and check pre- and postconditions for the provided private helper methods (`toAdditive`, `toSubtractive`, `merge`, `condense`, `getSymbolValue`, `toInt`, and `toRoman`). Since these methods support the particular implementation of `RomanNumeral`'s functionality (there's already thinking about algorithms that went into their creation), the pre- and postconditions have been identified for you — look for comments at the beginning and/or end of the bodies for each of those methods. "Add" here refers to adding information to the public comments for the method; "check" means writing code. As in earlier steps, break complex conditions down into simpler properties and write descriptively-named private helper methods to check (but you don't need to implement the bodies).



```java
/**
 * Convert to additive form (replacing subtractive pairs).
 *
 * @param roman
 *          string representation of a Roman numeral
 * @return additive form of 'roman'
 */
private String toAdditive ( String roman ) {
  // expect standard (subtractive)

  String expanded = roman.replace("CM","DCCCC");
  expanded = expanded.replace("CD","CCCC");
  expanded = expanded.replace("XC","LXXXX");
  expanded = expanded.replace("XL","XXXX");
  expanded = expanded.replace("IX","VIIII");
  expanded = expanded.replace("IV","IIII");

  // the symbols should be in order from highest value to lowest (M, D, C, L,
  // X, V, I)
  // integer value is equivalent to 'roman'
  return expanded;
}
```

use assertions to check private method preconditions and postconditions – violation is the fault of other code in this class

precondition – should be mentioned in the Javadoc comment and checked in the code

postconditions – should be mentioned in the Javadoc comment and checked in the code

Lab 3

- Don't rely on default values.

- this refers to variable initialization
  - local variables must be given a value before use but instance variables and array slots have default values
  - but always explicitly initialize instance variables and array slots

- Don't rely on pass through behavior in conditionals.

- include a case in the `if` for each alternative
  - this doesn't say every `if` must have an `else` – "do nothing" is a valid alternative – just that every alternative other than do nothing should have a case in the `if`

🚫
```
int curplayer = 1;
if ( score2 < score1 ) {
  curplayer = 2;
}
```

the case of `curplayer` being 1 is handled by passing through the `if`

★
```
int curplayer;
if ( score2 < score1 ) {
  curplayer = 2;
} else {
  curplayer = 1;
}
```

an explicit case for each alternative – `curplayer` set to 2 and `curplayer` set to 1

Lab 3

- Don't trust outside data.

- "outside data" is from outside the program or module – user input, parameter values
  - → error-check user input, check preconditions

- Reduce vertical scope.

- "scope" refers to where a name (variable, parameter, method, class) is visible and usable within a program
- reduce vertical scope by reducing the distance between declaration and use
  - declare variables close to where they are first used rather than all at once at the beginning of a method or block
  - declare variables as locally as possible – inside a block if they are only used there
- the principle does not refer to keeping method bodies short, though that is also a good practice

Lab 3

- Names should reveal intent.
- Names should be informative so comments describing what the variable is for should generally not be necessary.

- be informative and reveal intent
  - e.g. `player1`, `player2` store what about a player? – most likely identity-related info (name, id) in the absence of other context, but the reader is forced to rely on assumptions

- but also be concise – longer is not automatically better
  - don't repeat what is clear from context
    - e.g. naming methods `flip` and `play` to carry out the (whole) flip and play actions is fine because "flip" and "play" are understood terms within the context of the game Flip – `flipDie`, `playDice` doesn't add anything
    - e.g. `sum(dice)` vs `sumArray(dice)`, `calculateDiceSum(dice)` – the array parameter means mentioning "array" in the method name is unnecessary, and the dice parameter (particularly with well-named variables) provides the context to distinguish summing dice from some other sum operation
  - every part of the name should add something
    - e.g. `flipIt`, `playIt` – "it" doesn't add any meaning or clarity

Lab 3

- Avoid short variable names and long function names in large scopes.

- length is a proxy for specificity – short = more generic / less detail, long = more specific / more detail
  - more generic (shorter) variable names are OK in small scopes such as a loop body because it is more apparent from context what that name refers to
    - such names are more unclear in larger scopes, such as method bodies or as instance variables
    - e.g. a loop variable `i` for an array index is clear within a small for loop because it is clear from the loop pattern that it is just a current finger moving through the array – but an instance variable `i_` is less clear because the declaration is well separated from the loop or other usage that indicates what that index is for
  - functions with larger scopes (e.g. public methods rather than private ones) are more likely to be called often and to have a higher level of abstraction, so names should be shorter and less detailed
- always be as concise as possible
  - this principle isn't saying to make names longer just to make them longer but rather to consider the level of detail included in the name and include more detail (only) when needed

Lab 3

- Avoid disinformation.

- “disinformation” refers to names that are misleading because they include words with meanings at odds with the purpose of the variable/method
  - e.g. `int[] numlist` – in Java, a list is something different from an array so numlist misleads the reader about the type of variable
  - e.g. using `x`, `y` to refer to positions within a 2D array or `row`, `col` to refer to pixel locations within a drawing window – goes against conventions and is especially confusing because x ↔ col and y ↔ row
  - e.g. using `dice1`, `dice2` to refer to player 1’s flipped and unflipped dice is misleading because 1, 2 are expected to refer to player 1 and player 2
- use established terms
  - e.g. the flip rules refer repeatedly to “the middle” – so `middle`, referring to the dice collection in the middle, is understood while `sharedDicePool` is both unfamiliar (not referenced in the rules so what is it?) and potentially misleading (it sounds like a collection of dice both players can draw from...which is the case, but only in very specific circumstances)

Lab 3

- Make meaningful distinctions and pick one word per concept.

- “make meaningful distinctions” means that the difference between similar names/variables should be clear
  - e.g. `score1`, `score2` to distinguish between player 1’s score and player 2’s score is clear because there is a notion of player 1 and player 2 but `scoreA`, `scoreB` is less clear
  - e.g. `flipped1`, `unflipped1` to distinguish between player 1’s flipped and unflipped dice is clear but `dice1A`, `dice1B` is not
- “one word per concept” means to use the same word or naming scheme for the same/similar concepts
  - e.g. `player_one` and `player2` or `dice_one` and `score1` – either use words or numbers to refer to which player, then use that for all things belonging to a particular player