Assignment: Some Improvements

While the code from our initial exploration seems to work, and the classes we created seem agreeable, there are some areas in the code that we should improve on before moving forward.

1. We use `system "clear"` to clear the screen. Suppose we want to change this to some other command in the future - we'd have to change it in multiple places. Create a `clear` method and call this new method instead of `system "clear"`.

   ### Possible Solution

   ```ruby
   class TTTGame   # ... rest of class omitted for brevity
     def clear
       system "clear"
     end
   end
   ```

   This method name, however, collides with the local variable in `display_board`. Since the scope of the local variable is only within the method, let's rename that.

   ```ruby
   def display_board(clear_screen = true)
     clear if clear_screen    # ... rest of method omitted for brevity
   end
   ```

2. The first time we display a board, we want to suppress the clearing of the screen. This is so we can see the welcome message, or the play again message. However, the method invocation, `display_board(false)` is incredibly vague. Six months from now, no one will remember what that `false` stands for without looking at the method implementation. Let's change the method so that we can invoke it like this: `display_board(clear_screen: false)`.

   ### Possible Solution

   ```ruby
   def display_board(options = { clear_screen: true })
     clear if options[:clear_screen]    # ... rest of method omitted for
   brevity
   end
   ```

Note that when we invoke the method, we can do any of the following:

method invocation	effect
`display_board`	the `options` hash will be set to the default hash, `{clea`
`display_board({clear_screen: false})`	`options` will be replaced by the hash in the argument,
`display_board(clear_screen: false)`	same effect as above, despite the missing `{}`
`display_board clear_screen: false`	same effect as above, despite the missing `{}` and miss
`display_board(a: 1, b: 2)`	same effect as above, except the `options` hash will nov Surprisingly, this works for our method since we're ju calling `options[:clear_screen]`, which in this case wil evaluated as false)

Now that we can call `display_board(clear_screen: false)`, we stand a much better chance at remembering what this method does in the future. The code is almost self documenting.

3. **Though an improvement, the necessity to even pass in a qualifier to the `display_board` method points to a deeper problem. The method doesn't take a large number of options; it just takes 1 option. That option serves as a fork in the method: one fork clears the screen, then displays the board, while the other fork just displays the board. Let's create two methods: `display_board` and `clear_screen_and_display_board`. The former only displays the board, while the latter clears the screen first.**

Possible Solution

```
def clear_screen_and_display_board
  clear
  display_board
end
def display_board   # only code to display the board
end
```

We also have to replace all previous calls to `display_board` with `clear_screen_and_display_board`. And finally, we have to replace `display_board(clear_screen: false)` with our new `display_board`.

Now, all the methods are named appropriately, and we can invoke them without having to refer to their implementation. Even six months from now.

4. Speaking of better names, let's take a look at our `Board#detect_winner` method. The method name is ambiguous about what this method does. Just looking at the name, we wouldn't be surprised if it returned a `Player` object or a symbol. Let's rename it to reflect what it does: `winning_marker`. This new name reminds us that the method will return the winning marker, or `nil` in the case of no winning marker. After you change the method definition, don't forget to also update all method invocations.
5. Our `TTTGame#play` method reads very well. Most methods there are *declarative*. That is, we are just giving high level commands, like "display_board", "human_moves", and we're not focused on the *imperative* step-by-step instructions of how to do those things. Operating at this higher level of abstraction allows us to orchestrate the sequence of actions and organize the game flow much easier. However, we deviate a bit towards the end of the method after `play_again?`.

   For example, the below code after `play_again?` is very imperative in nature

```ruby
def play   # ... rest of method omitted for brevity
  break unless play_again?
  board.reset
  clear
  puts "Let's play again!"
  puts ""
end
```

   We should extract it to a well-named method to keep the `TTTGame#play` method at a declarative level. Let's move all of that into a method called `TTTGame#reset`.

Possible Solution

Implementing this method is as simple as copying pasting the imperative part over to the new method.

```ruby
def reset
  board.reset
  clear
  puts "Let's play again!"
  puts ""
end
```

This definitely works, but the `reset` method feels like it's doing a bit too much: it's affecting a change (resetting the board), as well as printing out some output. Let's move the two `puts` lines into another method.

```ruby
def reset
  board.reset
  clear
end
def display_play_again_message
  puts "Let's play again!"
  puts ""
end
```

Now, if we invoke the above two methods, our `TTTGame#play` method reads very fluidly. It's almost like reading natural English.

```ruby
def play
  clear
  display_welcome_message
  loop do
    display_board
      loop do
        human_moves
        break if board.someone_won? || board.full?
        computer_moves
        break if board.someone_won? || board.full?
        clear_screen_and_display_board
      end
      display_result
```

```
    break unless play_again?
    reset
    display_play_again_message
  end
  display_goodbye_message
end
```

6. As we glance down the list of methods, it's surprising that we display the board in the `TTTGame` class. That seems like a responsibility of the `Board` class. We should be able to *tell* the `board` to "display yourself". Let's move the logic from `display_board` to `Board#draw`. We'll still keep the `TTTGame#display_board` method, though, because the `TTTGame` needs to tweak the output a little (eg, the marker prompt at the top, and the padding.)

Possible Solution

```
class Board   # ... rest of class omitted for brevity
  def draw
    puts "     |     |"
    puts "  #{get_square_at(1)}  |  #{get_square_at(2)}  |
#{get_square_at(3)}"
    puts "     |     |"
    puts "-----+-----+-----"
    puts "     |     |"
    puts "  #{get_square_at(4)}  |  #{get_square_at(5)}  |
#{get_square_at(6)}"
    puts "     |     |"
    puts "-----+-----+-----"
    puts "     |     |"
    puts "  #{get_square_at(7)}  |  #{get_square_at(8)}  |
#{get_square_at(9)}"
    puts "     |     |"
  end
end
```

Notice that the `Board#draw` method above won't contain any of the extra messages. Instead, we'll leave that in the original `TTTGame#display_board` method, which is below.

```
class TTTGame   # ... rest of class omitted for brevity
  def display_board
    puts "You're a #{human.marker}. Computer is a #{computer.marker}."
    puts ""
    board.draw
    puts ""
  end
end
```

Now, the `TTTGame#display_board` just calls `Board#draw`. Why did we only move the board output to the `Board#draw`method, and not the extra information about the player and computer marker, and the extra `puts ""` before and after the display of the board? The answer has to do with organizing the code.

`Board#draw` shouldn't know anything about player markers or extra padding. It should only be concerned with one thing: drawing the state of the board. You can almost think of this as the board's `to_s` method. It should be generic so that it can be used in a variety of yet unanticipated situations.

`TTTGame#display_board` is where we're organizing all concerns related to presentation of the board in the Tic Tac Toe game flow. It's here that we know exactly what extra information we want in the context of a game.

7. After the changes in the previous step, we are now calling `Board#get_square_at` from the `Board#draw` method. Since `Board#draw` is an instance method, it has access to the `@squares` hash directly. Should `Board#draw` use `get_square_at` or interrogate `@squares` directly?

Possible Solution

There's no definite rule for this, but if your class has getter and setter methods, you should probably use them. There are times to avoid the getter/setter methods, such as when those methods do some pre or post processing, and you wish to only work with the raw data in the instance variable.

If your objects do not need to expose their internal instance variables to the outside, then you don't need getter or setter methods at all. In those situations, you can also access the instance variables directly. Note: this is only talking about referencing instance variables in the same class; this is not talking about reaching into an object from the outside and accessing or modifying its instance variables.

In our situation, we no longer need the `Board#get_square_at` method at all, since the only place we used it was rendering of the board. Now that the board rendering code has been moved to `Board#draw`, we can remove the `Board#get_square_at`method altogether; there's no need for that method anymore.

This implies that we can reference the `@squares` instance variable directly from `Board#draw`.

```
class Board   # ... rest of class omitted for brevity
  def draw
    puts "     |     |"
    puts "  #{@squares[1]}  |  #{@squares[2]}  |  #{@squares[3]}"
    puts "     |     |"
    puts "-----+-----+-----"
    puts "     |     |"
    puts "  #{@squares[4]}  |  #{@squares[5]}  |  #{@squares[6]}"
    puts "     |     |"
    puts "-----+-----+-----"
    puts "     |     |"
    puts "  #{@squares[7]}  |  #{@squares[8]}  |  #{@squares[9]}"
    puts "     |     |"   end end
```

8. Though we no longer need the `Board#get_square_at` method, we still need the `Board#set_square_at` method. This is the method that gets invoked when either the human or the computer makes their move. The method, though, is a little clunky. Let's make it more idiomatic Ruby. Instead of calling

this: `board.set_square_at(square, human.marker)`, let's update the board like this: `board[square] = human.marker`. That reads a lot better.

Possible Solution

Recall from the "Fake Operators" assignment that we can facilitate the desired syntax by creating a `Board#[]=` method.

```
class Board   # ... rest of class omitted for brevity
  def []=(num, marker)
    @squares[num].marker = marker
  end
end
```

Ruby sees the `[]=` method and allows us to invoke it with a special syntax that resembles assignment. The result is code that reads more more fluidly (but more confusing for the beginner).

We can now delete `Board#set_squares_at`, and change all its invocations with `Board#[]=`.

Note that if we ever need a getter method for the marker of a square, we can create a `Board#[]` method, which reads better than our old `Board#get_square_at`.

9. The next improvement we'll make is related to our most complicated method: `Board#winning_marker`. The problem with this method is that it relies on knowledge of both the human and computer markers. This doesn't feel quite right. Why does the `Board` class have to know about specific markers in the `TTTGame` class? A board object contains the state of the board. It's responsible for knowing things related to a board: whether all squares are marked, how to draw itself, how many empty squares are left, and whether a marker has won. The goal of `Board#winning_marker` is to return some winning marker or `nil`, but in our implementation, we hardcoded the human and computer markers. This board's implementation is tied to the implementation of `TTTGame` class. But in this case, that's not necessary. We should change the implementation of `Board#winning_marker` to see if any marker, not just the human or computer's, has won. If so, return that marker, and if not, return nil.

Possible Solution

First, we'll need to create a `Square#marked` method (it's not mandatory, but will help us write more concise code).

```
class Square  # ... rest of class omitted for brevity
  def marked?
    marker != INITIAL_MARKER
  end
end
```

In our `Board#winning_marker` method, we can call the method we wished existed.

```
def winning_marker
  WINNING_LINES.each do |line|
    squares = @squares.values_at(*line)
    if three_identical_markers?(squares)      # => we wish this method
existed
      return squares.first.marker             # => return the marker,
whatever it is
     end
   end
   nil
end
```

Finally, we can implement the desired method.

```
def three_identical_markers?(squares)
  markers = squares.select(&:marked?).collect(&:marker)
  return false if markers.size != 3
  markers.min == markers.max
end
```

Let's walk through the above three lines.

The first line is dense. First, we select only the marked squares, using the newly created `Square#marked?` method. Next, we transform that array of marked squares into an array of markers, or strings. The array of strings is assigned to the `markers` variable.

The second line is a guard. We can return false if there aren't three marked squares, because winning means 3 marked squares in a row.

By the time we get to the third line, we know that we have a 3-item array of strings. Now we just need to tell if these 3 strings are the same. We're relying on `Array#max` and `Array#min` here. Given an array of strings, `Array#max` will return the string that starts with the letter closest to `'Z'`. `Array#min`, however, will return the string that starts with the letter closest to `'A'`. Therefore, if both of those methods return the same value, then all elements in the `markers` array are identical.

There are lots of ways to determine if all elements in an array contains the same value, so we just picked one that looked cleanest. You could loop, of course. You could also do `markers.uniq.size == 1`. This relies on the fact that `Array#uniq` removes duplicate entries, so if there's only 1 element left, then all elements were the same.

Finally, let's move the `three_identical_markers?` method to a `private` method, since it's not being invoked from outside the class.

After implementing these changes, we can now also delete `Board#count_human_marker` and `Board#count_computer_marker` since they are no longer being used. Note that despite the dramatic update, we did not change the input or return values of the method at all, thereby saving us from having to make any changes to methods that rely on `winning_marker`, such as `someone_won?`. Our `Board` class now feels much cleaner and more general purpose. It's aware of generic board-related behaviors, and can return the winning marker, without mind to which exact marker it is.

10. Our code is looking good, but there's a little bit of redundant code in the main game loop. The code below has a pattern that seems ripe for extraction, can you see it?

```
loop do
  human_moves
  break if board.someone_won? || board.full?
  computer_moves
```

```
  break if board.someone_won? || board.full?
  clear_screen_and_display_board
end
```

It'd be nice to be able to introduce some notion of a "current player", and we could then remove the redundancy, like this:

```
loop do
  current_player_moves
  break if board.someone_won? || board.full?
  clear_screen_and_display_board if human_turn?
end
```

The trick is to alternate the "current player" after each turn. How can we do this?

Possible Solution

The first change we'll make is to introduce a new "state" in the game to keep track of who the current player is.

```
class TTTGame   # ... rest of class omitted for brevity
  def initialize
    @board = Board.new
    @human = Player.new(HUMAN_MARKER)
    @computer = Player.new(COMPUTER_MARKER)
    @current_marker = HUMAN_MARKER
   end
end
```

Notice that we're calling the new state `@current_marker`. Since we already have two constants `HUMAN_MARKER` and `COMPUTER_MARKER` that differentiates between the two players, we can piggyback on that to determine who the current player is. If we keep track of the current marker, we should be able to decide who should take the next move.

Next, let's implement the `current_player_moves` method. This method will just inspect the `@current_marker` instance variable and call either `human_moves` or `computer_moves`.

```
def current_player_moves
  if @current_marker == HUMAN_MARKER
    human_moves
  else
    computer_moves
  end
end
```

That looks reasonable enough. If it's currently the human's turn, call `human_moves`, otherwise call `computer_moves`. Next is the tricky part: don't forget to alternate the player!

We can actually do this right in the same method, like this:

```
def current_player_moves
  if @current_marker == HUMAN_MARKER
    human_moves
    @current_marker = COMPUTER_MARKER
  else
    computer_moves
    @current_marker = HUMAN_MARKER
  end
end
```

This will ensure that after the move has been executed, the `@current_marker` state is set to the other player.

Next, let's implement the `human_turn?` method. We only want to print the board and clear the screen when it's the player's turn. Otherwise we'll get extra unneeded output.

```
def human_turn?
  @current_marker == HUMAN_MARKER
end
```

Now that we have this method, we can also utilize it in our `current_player_moves` method as well.

```
def current_player_moves
  if human_turn?
    human_moves
    @current_marker = COMPUTER_MARKER
  else
    computer_moves
    @current_marker = HUMAN_MARKER
  end
end
```

The last thing we need to do is make sure to reset the `@current_marker` to whoever the first player is after the game is over. Otherwise, the current player may not be consistent if we play again.

```
def reset
  board.reset
  @current_marker = HUMAN_MARKER
  clear
end
```

Now this introduces a minor potential problem. Suppose we wanted to allow the computer to move first. If you didn't know the code well (or let's say you come back to it six months later), you might think that changing the `@current_marker` in the `TTTGame#initialize` method was enough. But it's very likely that you'd forget about the need to also make the same update in the `TTTGame#reset` method.

Let's fix this by creating a new constant called `FIRST_TO_MOVE` and set that to `HUMAN_MARKER`. Then, in the `initialize` and `reset` methods, we'll set `@current_marker` to `FIRST_TO_MOVE`.

```
class TTTGame   # ... rest of class omitted for brevity
  FIRST_TO_MOVE = HUMAN_MARKER
  def initialize     # ...
```

```ruby
    @current_marker = FIRST_TO_MOVE
  end
  def reset     # ...
    @current_marker = FIRST_TO_MOVE
  end
end
```

Now, if you want the computer to move first, just change `FIRST_TO_MOVE`!

11. Other than `initialize`, the `TTTGame` class only has one public method: `play`. The rest of the methods are called by `play`, which is internal to the class. Therefore, let's make all other methods in `TTTGame` private.

Complete Code

Here's the code that includes all the improvements and refactorings from above.

Show

Lecture: Discussion on OO TTT Code

Below are some ideas for you to ponder. No need to implement a solution.

1. Did you notice how tiresome it was to test for regression after every small change or refactoring? Besides being really careful, what else can we do to alleviate this burden? If you said "tests", you are right. One of the most important jobs for tests is preventing regression. We'll talk about writing tests in detail in a later lesson. Testing to drive out design is another important reason to use tests, but that's an entirely different topic we'll cover much later in the program.
2. Did you feel modifying this object oriented program to be easier, or felt safer, than the procedural Tic Tac Toe program? You should have! Object oriented programming forces you to set up more indirection, but that indirection gives us an opportunity to

isolate concerns so they do not ripple across an entire codebase. Changes are encapsulated to a class or object. The interface methods to collaborate with a class or object can remain the same while the implementation changes. This is one of the biggest benefits of object oriented programming.

3. Some of the classes have a generic name, like `Player` or `Board`. Suppose the end goal is to wrap our game up into a library (perhaps a gem?) and allow other developers to use it. Our generic class names are now in the global namespace. How do we fix this? (Answer: we should use a namespace for our application's classes, so it doesn't bleed into the global namespace.)
4. The `Player` class is quite bare. Do we even need a class in this case? Could we just use a `Struct` for `Player`, since it's currently nothing more than a data structure? That is, it contains some data, but no behaviors.
5. As we have more classes, we start to build a "dependency graph" of our classes. In OOP, we don't want the dependency graph to look like a spider web. Put another way, classes should collaborate with a few other classes. If all classes are collaborating with each other, the OO design should be reconsidered. For example, our dependency graph looks like this:
    - TTTGame collaborates with Player
    - TTTGame collaborates with Board
    - Board collaborates with Square

    Notice that Player knows nothing about the Square, and Board knows nothing about the Player. This is how we encapsulate and mitigate the ripple effects of change.

6. Analyze the `Board` and `Square` classes. Look at their main methods (or behaviors) in those two classes.

Board	Sq
`[]=`	`to`
`unmarked_keys`	`un`
`full?`	`ma`
`someone_won?`	`ma`

Board	Sq
`winning_marker`	`ma`
`reset`	
`draw`	
`three_identical_markers?`	

7.
8. Notice how the methods only deal with concerns related to the class. The only suspicious method may be `Board#three_identical_markers?` in that the game logic of "3 winning squares" has leaked into the board. However, that's a private method, so if the logic of what "winning" means changes (for example, if we had a 6x6 grid in the future), we can update the private method while still preserving our public interface, which is `Board#someone_won?` and `Board#winning_marker`.
9. When working with classes, it's important to focus on the behaviors and data in that class. It can be tempting to inject additional collaborators into the class, but keep in mind doing so will also introduce additional dependency. The `Board` knows about `Square`, but it doesn't know anything about `Player` or even the `TTTGame`. In that way, it tries to be a generic class, like `Array` or `Hash`.
10. What we just talked about in the previous point is hard to understand without more experience. Let's try to apply that knowledge to an example. Let's suppose that we're looking at the methods `TTTGame#human_moves` and `TTTGame#computer_moves` and we feel that this is behavior that should be moved to the `Player` class. We'll create a new method called `Player#move` that will handle making the move.

    The first thing we have to change is for the `Player` class to be aware of the difference between a "human" and a "computer". Therefore, we have to be able to differentiate between the two at object instantiation time.

```
class Player   # ... rest of class omitted for brevity
  def initialize(marker, player_type = :human)
    @marker = marker
    @player_type = player_type
  end
```

```ruby
  private

  def human?
    @player_type == :human
  end
end
```

This will allow us to instantiate `Player` objects like below.

```ruby
@human = Player.new(HUMAN_MARKER)
@computer = Player.new(COMPUTER_MARKER, :computer)
```

Now, we can create a `Player#move` method that will be self-aware of how to handle making a move depending on whether it's a human or computer. We can consolidate `TTTGame#human_moves` and `TTTGame#computer_moves` into one method `Player#move`.

```ruby
class Player   # ... rest of class omitted for brevity
  def move
    if human?
      puts "Choose a square (#{board.unmarked_keys.join(', ')}): "
      square = nil
      loop do
        square = gets.chomp.to_i
        break if board.unmarked_keys.include?(square)
        puts "Sorry, that's not a valid choice."
      end
      board[square] = marker
    else
      board[board.unmarked_keys.sample] = marker
    end
  end
end
```

The above code is copied and pasted directly, except a small change where we're setting the marker, we can just call the getter method `marker` without specifying the object, since we are within the `Player` object already.

This code won't work. The reason is because the `Player#move` method needs to be aware of a `board` to mark, and `Player`objects do not have any notion of a board. Seems like an easy fix: update the `Player#move` method to take a `board` object, and we're done.

That may be true, technically, but from a design perpsective, we've just introduced a new dependency between the `Player` and `Board` classes. Is this wrong? Is it so bad? The answer is uncertain, as it depends on the tradeoffs you're willing to make. For example, what if we renamed the `Player` class to `TTTPlayer`. Then in that case, it may be ok to allow a `TTTPlayer` object to be aware of the `Board`.

The key concept to understand here, though, is that there is a collaboration between `Player` and `Board`. Should that collaboration be organized in the `Player` class or the `Board` class? We could have `@human.marks(board)` or `board.marked_by(@human)` as equally valid choices. Or, we can decide that `Player` and `Board` objects should not directly collaborate with each other, and instead rely on an orchestrator class to use either object. That's what `TTTGame#human_moves` and `TTTGame#computer_moves` are -- the place where we decided to capture the collaboration between `Player` and `Board` objects.

In OOP, there are certainly incorrect ways to program, but there is rarely the "right" way. It all comes down to tradeoffs between tightly coupled dependencies, or loosely coupled dependencies. Tightly coupled dependencies are easier to understand, but offer less flexibility. Loosely coupled dependencies are more difficult to understand, but offer more long term flexibility. Which path is "right" depends on your application. Most of the time, beginners tend to over-apply design patterns. Don't pre-maturely optimize or build big architecture when you don't need to. On the other hand, recognize when you're introducing coupling and dependency, and eliminate unnecessary coupling if you can.

This is the "art" part of programming, and this is just a small taste of software design, patterns and architecture. Mastering this will be a life long journey, and your intuition will slowly improve as you gain experience.

11. **Consider if `Board` had used an array of `Square` objects, rather than a hash. How would that have changed things? What if `Square` objects contained the position `number` that they were in? We want a collection of squares that have different numbers. Should we use a `Set` collection instead?**