Game of Solo Chess

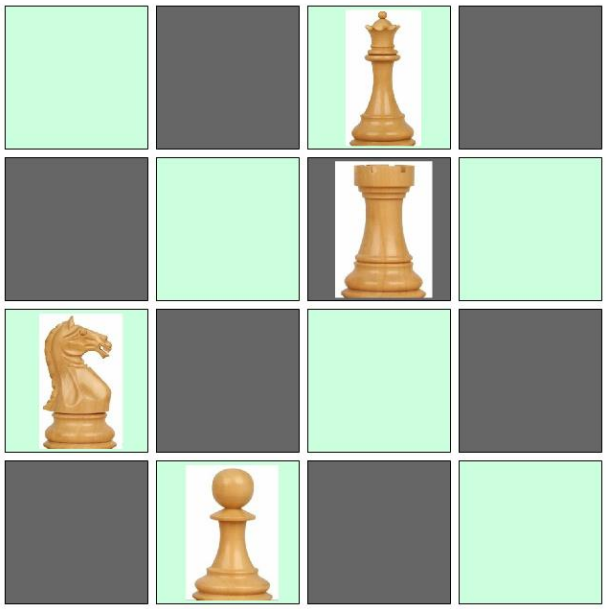
This program implements a popular board game that purports to "train" young minds for the game of chess. The following section describes how it is played.

How the game is played:

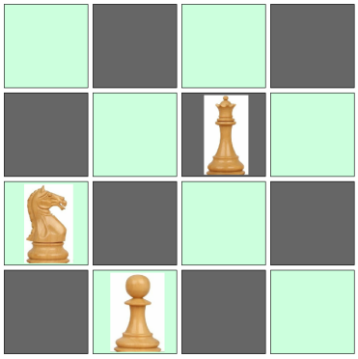
- Use a 4x4 chess board (instead of the standard 8x8)
- Use only 2 pieces each of bishop, knight, pawn, queen, and rook. Color doesn't matter.
- In the initial layout some of these pieces are laid out.
- Start playing with any piece:
    - Rule 1: Every move must kill another piece (using usual Chess rules)
    - Rule 2: Every move may use a different piece
    - Rule 3: When only 1 piece is left on the board, the game is over
- The challenge is thus to play (without violating the above rules) until only 1 piece is left.

Here is an example:

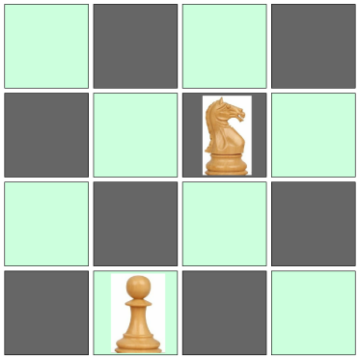
This image indicates one initial board position:



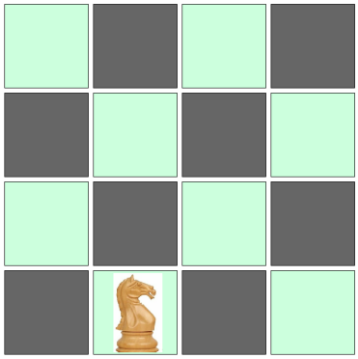
The following moves indicate how the game can be finished:



Queen takes rook



Knight takes queen



Knight takes pawn

Explore the game:

If you want to play with my final program to get a feel for this game, click the link given at the end of the article. Try not to peek at the scripts yet, since we want to design them ourselves below.

1. Click the "Green flag". Read help and then click to continue.
2. Pick one of the levels of expertise. The initial layout will be shown.
3. Play the game by following the instructions on the left side. Click "Undo" to retract a move. You can perform undo multiple times.

Click here to download all program files.

Scratch and CS Concepts Used

When we design this program, we will make use of the following Scratch and CS concepts. Learn these concepts if you don't know them before proceeding further.

- Arithmetic
  - Expressions
  - Basic operators (+, -, *, /)
  - Advanced operators: mod, floor, ceiling, %, //, etc.
- Concurrency
  - Synchronization using broadcasting
- Conditional statements:
  - Conditions: YES/NO questions
  - Relational operators (<, >, =)
  - Conditionals (IF)
  - Conditionals (If-Else)
  - Conditionals (Wait until)
  - Conditionals (nested IF)
  - Boolean operators (and, or, not)
- Data structures – list

    - List operations
    - Using list as 2-D array
- Data types – basic
    - Integers
- Data types – strings
    - String operations (join, letter, length of)
    - String traversal
- Divide and conquer (program design technique)
- Events
- Looping (iteration)
    - Looping - simple (repeat, forever)
    - Looping - nested
- Motion (Scratch and Snap)
    - Motion - absolute
- OOP
    - Clones
- Procedures
    - Built-in
    - User defined (custom)
    - Simple
    - With inputs
- Program output
    - Text
- Sequence
- STAMP - creating images
- User input
    - Click buttons
- Variables
    - Simple
    - Properties (built-in)
    - Local/global scope
- XY Geometry

High Level Design

Let's consider how the various features of this program can be separated out as distinct pieces. As usual, we have the front-end that interacts with the user, and the back-end that performs all game functions.

Front-end components:

- 4x4 chess-board:
    - Shows the current layout

    - When the user clicks on a chess piece, informs the backend which cell was clicked
- 5x2 chess pieces (rook, bishop, pawn, queen, knight)
    - If a cell contains a chess piece, its image is shown on top of the cell

For the frontend, we can simply borrow an older program called "matrix" (from the book "Practice CS Concepts with Scratch") which provides just this functionality. Since blank cells do not need to be click-sensitive, we can use stamping to draw the board and show chess pieces on top of them using cloning.

Back-end components:

- 4x4 layout of the board: list of 16 letters: 0 for empty, Q, B, K, R, P for pieces
    - The initial layout is copied from somewhere
    - For every valid (i.e. that satisfy game rules) pair of clicks, the layout is modified.

Objects:

We will distribute the code among the following objects (sprites):

- The "Square" sprite will contain logic to draw the visible 4x4 grid.
- The "Frontend" object will contain common code for all chess pieces.
- 5 chess objects: For each chess piece we will have a separate object containing its own logic – which would be fairly similar to each other.
- "Backend" will contain logic to manipulate the 4x4 board (i.e. the 16-item list) as per the game rules. This object (sprite) will drive the entire game.

We will add methods (procedures/scripts) to these objects as we process each feature idea below.

We may also add more objects as we learn more about the features of the program.

Global data:

List "L": 16 items, will hold the current status of the board, 0 for empty cell, r/b/q/k/p (for the respective chess piece)
"from" and "to": locations (1 to 16) for each move
"clicked": the most recent cell (1 to 16) that was clicked

Feature Idea # 1: The chess board

Draw a 4x4 grid of chess-like cells.

Design:

For this, we will simply borrow an older program called "chessboard" (from the book "Practice CS Concepts with Scratch") and modify it to draw only a 4x4 layout. Since the chessboard itself does not have to be sensitive to user clicks, (in this game, user can only click on chess pieces), we don't need to use clones to draw the chessboard.

Feature Idea # 2: The initial setup

When green flag is clicked, the game should display the initial board positions.

Design:

The game obviously depends on an initial layout. We will use a collection of possible layouts (copied from the actual board game). Each layout could be encoded into a 16-letter string: 0 for blank cell and r/k/q/b/p for a chess piece. For example, the encoding for the board shown at the top of this article would be "00q000r0k0000p00".

We will have 4 collections of such encoded layouts based on level of difficulty: expert, advanced, intermediate, and beginner.

We could save these 4 collections in 4 separate variables: one for each type. So, for example, "expert_string" would contain all expert-level layouts separated by commas. Since Scratch saves variable contents in its projects, we don't need to load these strings every time we run the program.

We will need 4 additional sprites (click buttons) to allow the user to pick a level by clicking. The program would then take the appropriate comma-separated string, split it into a list, and pick one of the layouts at random. This layout encoding (16-letter string) would then result into the actual chess layout as described in the next feature.

Feature Idea # 2: Layout of chess pieces

Display chess pieces according to the current layout.

Design:

The current layout, as described above is a 16-letter string. We will split it into a 16-item list L. This list will henceforth determine what each board position looks like (either blank or with a chess piece). The "frontend" object will scan this list and inform (by sending messages) the chess pieces to show up at their assigned places. This scanning script can simply broadcast the letter associated with the piece.

Since each piece can have 2 instances (i.e. 2 rooks, 2 pawns, etc.) we will use clones of each piece. The clone, when it receive the message (e.g. "r" for the rook), will create and place a clone at the same position as in L (e.g. if "r" appears at position 12 in L, the rook clone will appear at position 12 in the 4x4 grid).

Feature Idea # 3: The move

When user clicks on two valid pieces, make that move.

Design:

According to the game's rules, every move involves one chess piece taking another. Thus, we have a 2-step transaction here: in step 1, user clicks on a piece, and in step 2, he/she clicks either on the same piece (to cancel the move) or on another piece to take it. How can we program this? Here is one possible approach.

When a piece is clicked it will simply save its grid location (1 to 16) in a global variable ("clicked") and inform the backend.

The backend will use two global variables: "from" and "to". Initially "from" would have some invalid value such as -1 to indicate that the user is yet to begin the move. As soon as user makes the first click, we save the location in "from". The second time user clicks, "from" would not be -1, so we will know this is the second click. If the second click is the same as the first, we simply cancel the move by setting "from" back to -1. Otherwise, we do the following:

- Validate the move: Send a message to the piece at the first click to check if the move is valid (this feature is implemented later). Proceed only if true.
- Save the move in an "undo" list (for the "undo" feature implemented later).
- Update list L to show the move.
- Send a message to the piece at the second click to inform it to pack its bags.
- Send a message to the piece at the first click to inform it to move to the second click.

Each chess piece object will implement its own logic for steps 1, 4, and 5 as follows:

- For Step 1, each piece will implement a "ValidateMove" method (script), which will verify that the intended move is legal (e.g. rook can only move straight). If it is not, this method will return error to the backend which should then cancel the move by setting "from" back to -1. We will implement this method later and for now we will assume the user knows what he/she is doing.
- For step 4, each piece will implement a "Die" method which will cause the clone to delete itself.
- For step 5, each piece will implement a "Move" method which will cause the clone to move to the specified "to" position.

Save as Program Version 1

Congratulations! You have completed all the basic features of the game. Compare your program with my program at the link below.

File: solochess-1.sb2

How to play the game:

1. Click the “Green flag”.
2. Pick one of the levels of expertise. The initial layout will be shown.
3. Play the game by following the instructions on the left side.

Feature Idea # 5: Undo

User should be able to undo his/her moves.

Design:

For this, we will need to save every move – the best place would be a list. Let us call it "undoL" – the undo list. We will save in a single string the "from" and "to" locations as well as which piece was taken (killed) by which piece, for example, "2,15,k,p" would mean knight at 2 was moved to replace a pawn at 15.

Next, we will provide an "Undo" click button. When user clicks this button, the latest element in "undoL" would be popped and processed. For example, if it is "2,15,k,p", we need to perform the following actions to undo:

- The knight at 15 came from 2, so we will ask it to move back to 2 (using the "Move" method).
- A new pawn needs to appear at 15. A pawn clone can be created and placed the same way as during the initial placement (backend will send a "p" message).

Of course, don't forget to update list L with the new layout since pieces have moved around.

Feature Idea # 6: Help

Provide a help screen.

Design:

This is a straightforward task. Create a "Help" sprite and display it first when Green flag is clicked. You will need to ensure all other characters/sprites hide at this time. When the user clicks to continue, hide the "help" sprite and continue the program.

Save as Program Version 2

Congratulations! You have completed the undo feature of the game. Compare your program with my program at the link below.

File: solochess-2.sb2

Feature Idea # 7: Allow valid moves

In feature idea #3 above, we move pieces without checking if the moves follow chess rules. Implement these rules. For example, bishops only travel diagonally.

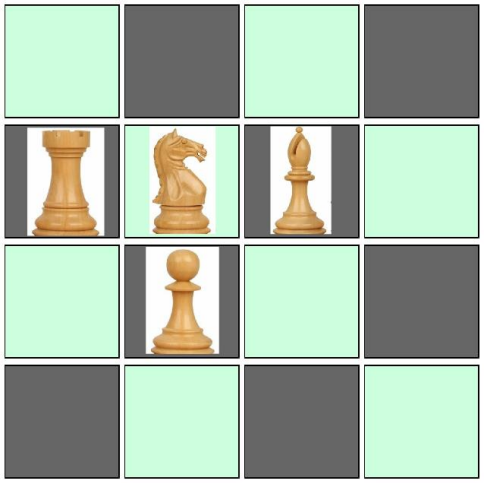
Design:

This feature would be implemented by each piece separately (as the "ValidateMove" method) since the rules of movement are unique to each piece.

***Step 1*: Ensure pawn moves are valid.**

Design:

Although pawns move straight up only, while killing they move diagonally. Consider the board as shown below:



In this layout, the pawn can kill the rook or the bishop, but not the knight or any other piece on the board (if there were any). How do we check that the move picked by the user ("from" to "to") is valid?

After careful analysis, we come up with the following observations:

1. "from" is always greater than "to"
2. "from" cannot be less than 5
3. "to" cannot be greater than 12
4. The difference between them can either be 3 or 5.
5. When the difference is 3, "from" cannot be in the last column
6. When the difference is 5, "to" cannot be in the last column

The following algorithm takes care of all these observations. Since we only allow a gap of 3 or 5, items 1 thru 4 are automatically taken care of. (For example, if from = 4 (violating #2) and to = 1, the gap would be 3 and the first "if" below would be violated.)

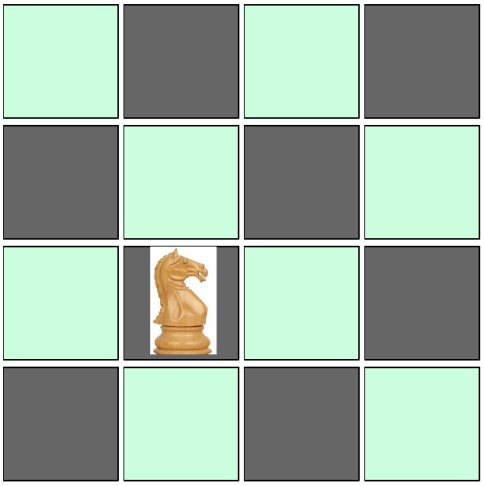
```
Algorithm IsMoveValidPawn for the pawn:
Input: from and to (numbers 1 to 16)
Gap = from – to
If Gap is 3 AND "from" is not in the last column
          Return True
If Gap is 5 AND "to" is not in the last column
          Return True
Return false
```

***Step 2*: Ensure knight moves are valid.**

Design:

Knight can move as follows:

- Pick any of the 4 directions (north/south/east/west)
- Move 2 steps straight
- Move 1 step at the right angle



In the layout above the knight can move to 1, 3, 8, or 16. How do we check that the move picked by the user ("from" to "to") is valid?

After careful analysis, we come up with the following observations:

- It is better to use row (R) and column (C) to do this check.
- There are 8 possible movements as below. (Not all would be valid)
- New positions if it moved 2 steps east, 1 step north or south: (R-1,C+2), (R+1,C+2)
- New positions if it moved 2 steps west, 1 step north or south: (R-1,C-2), (R+1,C-2)
- New positions if it moved 2 steps north, 1 step east or west: (R+2,C+1), (R+2,C-1)
- New positions if it moved 2 steps south, 1 step east or west: (R-2,C+1), (R-2,C-1)
- If we removed the invalid positions (by looking at the new row and column values) we would have a list of valid moves
- We could then check if "to" is one of the valid moves.

The following algorithm takes care of these observations.

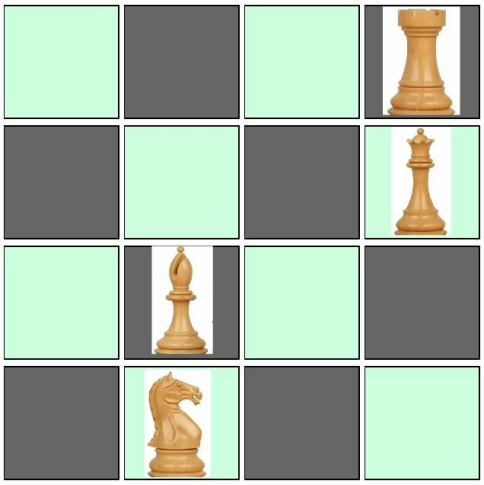
```
Algorithm IsMoveValidKnight for the knight:
Input: from and to (numbers 1 to 16)
R = calculate row # of "to"
C = calculate col # of "to"
Create a list L of all valid moves as described above
      Example: (R-1, C+2) is a possible move
      If R-1 is a valid row and C+2 is a valid column
            newposition = ((R-1)-1) * 4 + (C+2)
            Add newposition to L
If "to" is a member of L return True, else return False
```

***Step 3**: Ensure bishop moves are valid.*

Design:

Bishop can move as follows:

- Pick any of the 4 diagonal directions (nw/sw/se/sw)
- Move straight until it hits another piece



In the above layout, the bishop can legally move to position 4 and any other move would be invalid (for this game). How do we check that the move picked by the user ("from" to "to") is valid?

After careful analysis, we come up with the following observations:

- If "from" and "to" are diagonally positioned (newrow – oldrow must be the same in value as newcolumn – oldcolumn) it is a valid move.
- Next we need to find in which direction "to" is situated. This can be found by comparing row/column of "to" with those of "from". For example, if newrow < oldrow and newcolumn < oldcolumn, "to" must be northwest of "from".
- We find the first valid cell (i.e. occupied) in that direction. If it is the same as "to" it is a valid move, else it's an invalid move.

The following algorithm takes care of these observations.

```
Algorithm IsMoveValidBishop for the bishop:
Input: from and to (numbers 1 to 16)
R1 and C1: calculate row/column of "from" (1 to 4)
R2 and C2: calculate row/column of "to" (1 to 4)
g1 = R2 - R1
g2 = C2 - C1
If value of g1 and g2 are not equal return False
Determine "direction" of the move by comparing g1 and g2.
     For example: If g1<0 and g2<0 the direction is Northwest.
For each direction:
```

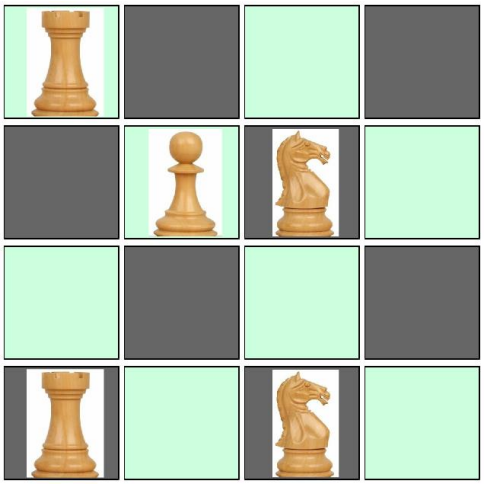
```
Imagine bishop moving from "from" one cell at a time.
Move until an occupied cell is found.
If occupied cell same as "to" return True
Else return False
```

<u>Step 4</u>: Ensure rook moves are valid.

Design:

Rook can move as follows:

- Pick any of the 4 diagonal directions (N/E/W/S)
- Move straight until it hits another piece



In the above layout, the rook at 13 can legally move to positions 1 and 15 and any other move would be invalid (for this game). How do we check that the move picked by the user ("from" to "to") is valid?

After careful analysis, we come up with the following observations:

- If "from" and "to" are positioned straight up/down or sideways (both newrow – oldrow and newcolumn – oldcolumn cannot be non-zero) it is a valid move.
- Next we need to find in which direction "to" is situated. This can be found by comparing row/column of "to" with those of "from". For example, if newrow < oldrow and newcolumn = oldcolumn, "to" must be North of "from".
- We find the first valid cell (i.e. occupied) in that direction. If it is the same as "to" it is a valid move, else it's an invalid move.

The following algorithm takes care of these observations.

```
Algorithm IsMoveValidRook for the rook:
Input: from and to (numbers 1 to 16)
R1 and C1: calculate row/column of "from" (1 to 4)
R2 and C2: calculate row/column of "to" (1 to 4)
g1 = R2 - R1
```

```
g2 = C2 - C1
Only one of g1 and g2 must be non-zero. If not, return False
Determine "direction" of the move by comparing g1 and g2.
      For example: If g1=0 and g2<0 the direction is West.
For each direction:
      Imagine rook moving from "from" one cell at a time.
      Move until an occupied cell is found.
      If occupied cell same as "to" return True
      Else return False
```

<u>Step 5</u>: Ensure queen moves are valid.

Design:

Queen can move either as a rook or as a bishop. So we can simply use the bishop and rook algorithms designed earlier.

```
We first run the bishop algorithm
      If it returns True, we return True
Next, we run the rook algorithm and return whatever it returns.
```

Save as Program Version Final

Congratulations! You have completed all features of the game. Compare your program with my program at the link below.

File: solochess-3.sb2
Link: https://scratch.mit.edu/projects/371410747/

How to play the game:

1. Click the "Green flag". Read help and then click to continue.
2. Pick one of the levels of expertise. The initial layout will be shown.
3. Play the game by following the instructions on the left side. Click "Undo" to retract a move. You can perform undo multiple times.

Author: Abhay B. Joshi (abjoshi@yahoo.com)
Last updated: 6 March 2020