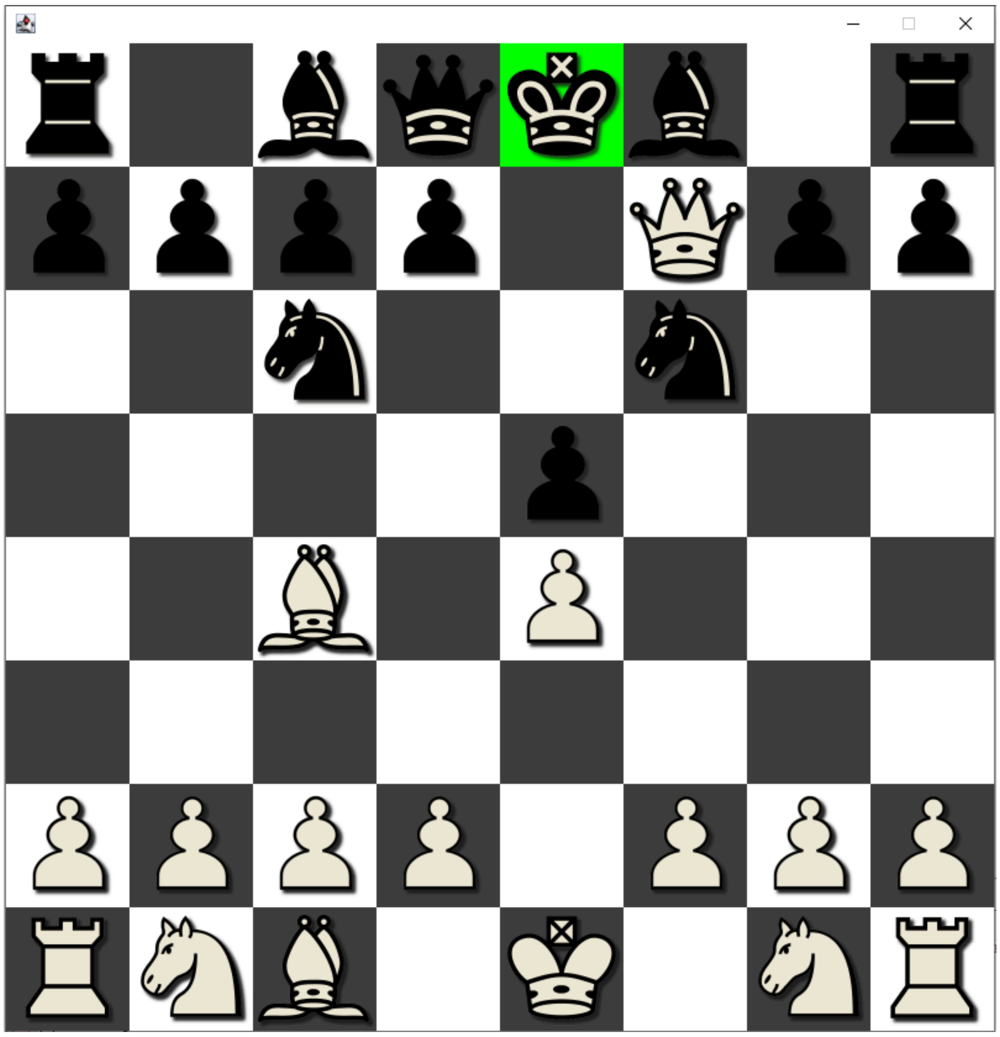
Chess - Development Guide

Prof. Tornese Gianluca
Version: 1.0.0
16/11/2023



Introduction

This document describes how to implement the game *Chess* with the Java programming language, relying on the swing graphical library.

It starts by presenting an overall view of the system, to show the project as a whole.

It then proceeds in describing each class individually, by detailing the content (methods and attributes).

In terms of coding, knowledge about the following OOP concepts is required:

- Classes and instances
- Access modifiers (public, protected, private)
- Keyword new and constructor method
- Getter and setter methods
- Keyword static
- Exception handling
- Interfaces
- Basic understanding of event-programming

Often this document will try and guide the development through a set of hints and suggestions, rather than providing the direct solution (code snippet). When pseudocode is provided, it is on purpose incomplete, to stimulate active reasoning and your overall engagement.

Few suggestions: save and compile often (better be safe than sorry), be brave and don't get discouraged, it will all come together in the end.

Let the coding begin!

Project and packages

Let's call our *NetBeans* project *chess*, with great imagination.

In this project the classes will be organized in two packages (sub-folders): *core* and *gui*.

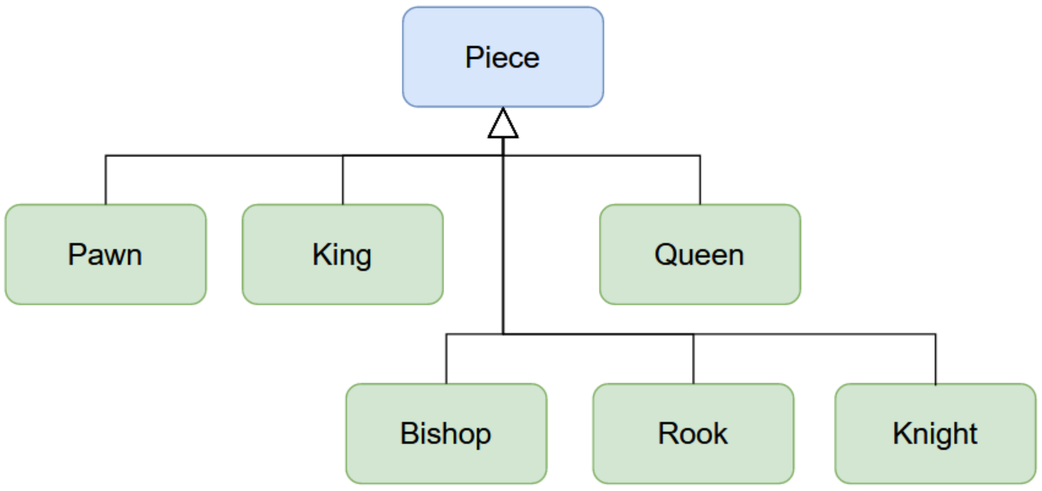
The package *core* will contain all classes related to game dynamics.

The package *gui* will contain all classes related to the Graphical User Interface.

Package Core

One of the key aspects of the chess game dynamics is related to the different movement and capture capabilities of each piece.
As such, it is inevitable that we implement a class to represent each piece type, that will focus on its rules and dynamics and hide complexity to the outside world.

Let's define a base class *Piece*, and a set of sub-classes that extend from it. Each sub-class will represent a given piece type:



We will benefit from having two dedicated enum, *Color* and *PieceType*, that will act as labels for us to distinguish among values.

We will need an entity representing a single square of the board, let's call it *Tile*.

Further on we will also need to have a grid, or matrix, of tiles, let's call this class *Board*.

The overall game session will be driven by the user's move: the class that processes the moves will be called, no surprise, *Game*.

We will also need a class to store the concept of a move, in terms of source and target: *Move* sounds good.

Finally, we will rely on a custom exception to handle invalid moves made by our distracted player: let's call it *InvalidMoveException*.

The package Core will hence consist of the following classes:

- *Piece*
- *Pawn*
- *King*
- *Queen*
- *Bishop*
- *Rook*
- *Knight*
- *Tile*
- *Game*
- *Board*
- *Move*
- *InvalidMoveException*

And by two enums:

- *Color*

- *PieceType*

Let’s create all these as placeholders, each in its own .java file, within the newly created *core* package!

Package Gui

In the package *gui* we will define all classes needed to handle the graphical user interface.

This includes the game graphics, the user interaction through the mouse, and the overall main application.

In terms of graphics this project relies on the swing libraries: in this framework the main graphical object is a *JFrame*.

As a start then let’s define a class, let’s call it *GameFrame*, that extends *JFrame*.

The main frame will then host the chess board in what the swing libraries call a *JPanel*; let’s hence create a class named *BoardPanel* that extends *JPanel*.

In order to react to mouse events, we will need a *listener*. A *listener* is an object that monitors a given target, waiting for special events to occur. When the events occur, certain registered given functions will be executed by the main graphical thread. These functions are often also referred to as *callbacks*.

Our listener will be called *GameDynamicsListener* and shall implement two interfaces, *MouseListener* and *MouseMotionListener*. We will come back later to the implementation details.

Finally, we need our main function, that shall host the main *JFrame*. Let’s create a class named *GuiLauncher* with the main function, and within this method let’s declare and instantiate our *GameFrame* object. Once the frame is spawned, the graphical application will stay open until manually closed (as any other normal window).

Icons

Our chess game will need icons for each piece, respectively for black and white army. The icons are provided as part of the project on classroom, and shall be unpacked in the NetBeans project folder, as follows:

- chess
    - target

- src
    - core
    - gui
    - icons
        - WR.png
        - WQ.png
        - WP.png
        - …

It is crucial to preserve this tree structure in the project, for the icons to be correctly retrieved at run-time.

Classes

core.Tile

The *Tile* class is relatively simple. It needs:

- an attribute representing the row
- an attribute representing the column
- an attribute containing the *Piece* that may (or may not) occupy the tile
- an attribute stating the color

Hint: each of these attributes shall have private visibility, and be initialized in the constructor. By default, the *Piece* attribute shall be set to *null*.

The *Tile* object is mutable in terms of the piece that occupies it, while it shall not change in terms of row, column and color. As such, we will also need a getter and setter method for the *Piece* attribute.

---

core.Color

Very straightforward enum, it shall contain simply the two labels BLACK and WHITE.

---

core.PieceType

Very straightforward enum, it shall contain labels for all types:

- PAWN
- KING
- QUEEN
- KNIGHT
- BISHOP
- ROOK

core.InvalidMoveException

This is our specific exception, designed to validate our game dynamics. Let's simply extend *Exception* and redefine the constructor that takes a *String* as parameter.

core.Board

We have defined what a *Tile* is, it's time to put a set of these together and have our game board ready!

A *Board* is hence simply enough a collection of *Tile*, in computer science how can we model a grid of homogeneous types?

That's right, nothing fancy, a plain old matrix!

Our first attribute is hence a matrix of *Tiles*, let's call it *tiles*.

Let's now define a few static helper attributes that will come in handy pretty much everywhere in our project. We will need:

- a static final int attribute called *Rows*, initialized to 8
- a static final int attribute called *Columns*, initialized to 8
- a static final int attribute called *LastRow*, initialized to *Rows - 1*
- a static final int attribute called *LastColumn*, initialized to *Columns - 1*

Now, time to build our constructor. No parameters needed, it shall instantiate the matrix *tiles* of the right dimension (let' never hard-code the number 8, we have just defined our helper attributes about it).

Now we need the two usual loops to initialize the matrix; each tile object shall receive its row and column (*i* and *j*, simply enough), and a color (black or white). Can you think of a simple way to determine whether each tile is black or white, based on row and column?

That's it for the constructor, let's now add one helper method: *GetTile*. It shall return a tile, receiving the row and the column as parameter. Optionally, before accessing the matrix, it could also verify that rows and columns are within the acceptable range. The acceptable ranges are, respectively, [0, LastRow] and [0, LastColumn]. If the parameters are outside the valid range, we could for example throw an IllegalArgumentException.

core.Move

This is the class that represents a single given move, in terms of source and target.

As such it shall have the following 4 attributes:

- sourceRow
- sourceColumn
- targetRow
- targetColumn

All these are integers, and shall not be modified from the outside; what access modifier should we pick?

Even though we do not modify them from the outside, we still want to read them out; let's define getter methods for them.

Finally, later on we will see that it will help for a given instance of the *Move* class to have visibility of the overall *Board*; let's define hence an additional attribute called *board* of this type.

All of these attributes shall be received as parameters and initialized in the constructor.

Let's now code a few helper methods that will help us along the way.

1. Let's define a method *getPiece*, that returns a *Piece*, and receives the board's row and column as parameters. It then invokes on its board object the method *GetTile*, passing the row and column, and on the resulting Tile it invokes the method *getPiece*.
2. Similarly, let's have a method *setPiece* that receives the row, the column and the *Piece* to be set. Once again, it relies on the board object and the *GetTile* method to invoke the *setPiece* method.

Note: the two methods implemented at bullet 1 and 2 are examples of wrapper methods: they "wrap" (surround) an existing functionality, rather than implementing a new one from scratch.

3. Let's define a method *onTarget*, that returns a boolean and receives the usual two parameters, row and column. The method shall return true if the attributes targetRow and targetColumn are equivalent to the ones received as parameters.
4. Let's now add a method *checkPiecePresence*, that receives the usual two parameters (row and column), and returns true if the associated *Tile* contains a *Piece* or false otherwise. (Hint: a tile without a *Piece* will contain the *null* reference).
5. Let's add a method called *wouldEndInKingCheck*, with no parameters and no return value; for now let's leave it blank, we will come back later to it
6. Let's add a method *isTargetOccupiedByAlly*, that returns a boolean. It shall check whether there's a piece on the target tile, and if so if it has the same color of the moving piece (on the source tile). In such a case it shall return true, and it shall return false otherwise.
7. Let's add a method called *isKingInCheck*, that returns a boolean and receives as parameter:
    a. a row (let's call it *kingRow*)

   b. a column (let’s call it *kingColumn*)
   c. a color (let’s call it *kingColor*)

   For the time being, let’s simply return false, we will come back later to it

8. Finally, a very important method: *checkObstacles*. This method returns a boolean and receives no parameters.
   It shall implement the logic that checks whether there’s any obstacle in the depicted move. This will come in handy whenever we process a user’s move.
   This logic is fairly complex, we shall for example consider whether the move makes the row and column respectively increasing or decreasing, it shall proceed along the move sometimes only in a given direction (let’s imagine the rook’s move for example), and so on.
   A possible implementation, in pseudo code, would look like this:

```
row = sourceRow;
column = sourceColumn;

// is row increasing or decreasing?
rowGrows = sourceRow < targetRow

// is column increasing or decreasing?
columnGrows = sourceColumn < targetColumn

while we are not yet on target // let’s rely on the onTarget method
{
        //if row is not on target, let’s move it
        if row != targetRow
        {
                if rowGrows
                        row++
                else
                        row–;
        }

        // same logic is applied to the column

        // row and column are increased (if necessary);
        // have we reached the target? if so, return false

        if onTarget
                return false; // no obstacle met

        // we are not yet on target, let’s verify if there’s a piece
        // before proceeding in the cycle

        targetPiece = board.GetTile(row, column).getPiece();

        if targetPiece is not null
                return true; // obstacle met along the way
}

return false; // no obstacle met
```

---

core.Piece

This is the base class for all our pieces.

It needs to be abstract, in the sense that it can't be instantiated. It's good to have it, it allows us to treat pieces homogeneously, but we shall never create a generic *Piece*.

In terms of attributes, a *Piece* shall have:

- a type
- a color

We got enums covering the types of these two attributes, let's go ahead and use them!

In terms of access, we will need these attributes in the sub-classes (i.e. knight or bishop). What's the best modifier to be used on this occasion?

Let's start from the trivial stuff. It won't harm to have getter methods for type and color, and to have a custom toString method to represent our instances. We also need a constructor that receives and initializes all attributes. Let's code these.

As the game progresses we will need to manipulate our pieces, removing them from the armies and even adding them (pawn promotion, good times). Let's hence override the equals method, returning a boolean, receiving as parameter an Object, and implementing a comparison based on:

- the getClass() method (different object won't be equal, clearly)
- the type
- the color

Now it's time for actual game dynamics content. The main idea is to have three default methods, that will be invoked from the outside, to determine whether:

- A move is valid
- A move has been executed
- A piece capture is valid

These three methods will be crucial in accepting a user's move and to take into account that a move has been processed (for example a king cannot castle if it has moved already).

The three methods shall be, respectively:

- void validateMove
- void executeMove
- boolean validateCapture

All of them shall receive a *Move* as parameter.

Let's have *executeMove* completely blank; the default implementation won't do anything, it will be overridden by those pieces who need it. For example a *Pawn* can move forward two tiles only on the first move, and this method offers the chance to store the logic status of *firstMove*.

validateMove is really a key method: each piece will overwrite it and define its own game dynamics. However there's still something that can be generalized and applied to all pieces, and hence be defined here. Here is how the pseudo-code will look like:

```
if move.sourceRow is equivalent to move.targetRow and same applies
to column
{
        // user specified a move with same source and destination
        throw an InvalidMoveException
}

if move.isTargetOccupiedByAlly
{
        // user specified a move that would end on another piece of its army!
        throw an InvalidMoveException
}
```

By implementing these base checks, we discovered the usage of the *InvalidMoveException* exception. The idea behind it it's that there could be so many reasons that invalidate a given move, that instead of returning true or false it is much much easier to throw an exception and stop as soon as a problem is detected.

Finally, let's provide a base implementation for the *validateCapture* method. This method verifies if a capture is valid, and comes in handy when checking for example king's checks. Most of the pieces capture simply by moving following their dynamics, but some of them don't (did you say pawns?). This method provides a base implementation hence for most of the pieces, and its logic looks something like this:

```
try
{
        validate move

        // no exception raised, move is valid, and so would be the capture
        return true;
}
catch InvalidMoveException
{
        // the move is invalid, exception was raised, and so is the capture
        return false;
    }
```

Great, we now have the basic concept of a chess Piece!

However the devil is in the details, as they say, so we will need to get our hands dirty with each single piece…

---

core.Game

We've got all the basic components in place, it's time to model the actual game session. The *Game* class shall have the following attributes:

- The current game *turn*, in terms of which color
- The two armies (*blackArmy* and *whiteArmy*), declared as ArrayList<Piece>
- The chess *Board* object, let's call it *board*

The constructor will initialize *turn* to the right color (which color starts first?), and then instantiate the two armies.

The armies are ArrayList and not array simply because it's easier to manipulate

It will also instantiate the board, and then invoke a class method called *setup*, that we are going to implement next.

The *setup* method does not take parameters, nor does it return anything; its role is to spawn all the right pieces on the board. For example, cell [0,0] shall contain a black rook, while [1,0] a black pawn. We hence need to create the right pieces, and set them on the proper tiles. Also, let's not forget to add them to the right armies.

Let's now add a *clear* method, responsible for nuking away the board's content. It shall iterate through the board tiles (nested for-loops), and invoke setPiece(null) on each.

It will so help later on to have the following methods:

- a getter method for the board
- a getter method for the turn
- a *removePieceFromArmy* method, that receives as parameter a given Piece and a color and does not return anything. Based on the received color it shall select which army to remove from, iterate the army and as soon as an element matches the specified piece type, it shall remove it and terminate (return)
- a *addPieceToArmy* method, that once again receives as parameter a given Piece and a color, and does not return anything. Once the army is selected, it shall add the given piece. This method will help us when pawns get promoted.

Let's add another method, called *isMoveSourceValid*. Given a row and a column (as parameter), it shall:

- return false if there's no piece on the source tile (what a distracted user!)
- return false if there's a piece on the source tile, but the color does not match the current turn (again, what a distracted user!)
- return true otherwise

Now on with the actual dynamics!

We need a *processMove* method, that receives as parameter

- a source row
- a source column
- a target row
- a target column

And return a boolean.

First, it shall create a *Move* instance, based on the received parameter.

It shall then try validating and executing a move, and look something like this:

```
try {

        validateMove()
        move.wouldEndInKingCheck()
        executeMove()

        current turn is over, let's switch color
        return true;
}
catch(InvalidMoveException)
{
        // something did not work quite as expected!
        print out exception content
}
return true;
```

Well done! The method *validateMove* is next. It receives a *Move*, and simply relies on the moving *Piece* to determine whether the move is valid or not. Let's then get the moving *Piece* from the right *Tile*, and on it invoke *validateMove*.
Finally, the *executeMove* method; it receives the *Move* and tries executing it. First, let's retrieve the source and target tiles. On these, let's retrieve the *Piece*: we shall hence obtain two variables, *sourcePiece* and *targetPiece*.
Now we invoke *executeMove* on the *sourcePiece*, and determine whether this move is capturing an opponent *Piece* or not. Let's simply check if *targetPiece* is not null, and remove it from its army.

Whether we are capturing or not, we need to invoke the *setPiece* method on the *targetTile* to set *sourcePiece*, and symmetrically set *null* on the *sourceTile* (that now is empty).

---

gui.GameDynamicsListener

Game dynamics are looking good, but it's also time to focus on the user interface. Let's start with one of the key aspects: how are we going to move our pieces?

We expect our chess game to be played with the mouse, hence we shall somehow react to the user's mouse clicks and movements.

That's where our *GameDynamicsListener* class comes into play! It implements (not extends) the interfaces *MouseListener* and *MouseMotionListener*. These are the two interfaces of the *swing* framework that provide *events* for exactly what we are looking for.

Before going further, we need to define a constructor that receives the BoardPanel object as a parameter, and stores it in a local private attribute.

In order to implement the interfaces, we need to provide the following methods:

- mouseClicked
- mousePressed
- mouseReleased
- mouseEntered
- mouseExited
- mouseDragged
- mouseMoved

All these methods receive a *MouseEvent* as parameter, and have return type void. Let's not forget the @Override annotation.

All these are plain place-holders, with blank implementations, except for one: *mousePressed*.

The method (or better, the callback) *mousePressed* will be triggered every time we click on our game window, and will provide us coordinates relative to the frame origin.

To retrieve the click coordinates, simply invoke *getX()* and *getY()* on the *MouseEvent* parameter.
Let's keep in mind that the X coordinate gives us the column, while the Y coordinate gives us the row.

Also, we will be drawing a board that is 800x800, with each tile being 100x100. Let's jump to the *BoardPanel* class and a static attribute called *TILE_DIMENSION* initialized to 100.

Now, whenever we click in the frame, we can retrieve the coordinates and divide by the tile dimension, respectively to determine the row and the column.

Once we have determined the row and the column, let's invoke the function *onMove* provided by *BoardPanel*, passing the row and the column. If the method does not exist just yet, let's create it and leave it empty for the time being.

---

gui.GameFrame

This is as anticipated the main *JFrame* graphical object. Every java *swing* application shall own one of these objects.

So let's go ahead and extend JFrame. We are going to need two attributes:

- Our *BoardPanel*
- Our *core.Game*; this will be the contact point with between the game dynamics (package *core*) and the UI (package *gui*)

Now on with the constructor, no parameters needed, and:

- it shall create the *Game* instance
- it shall create the *BoardPanel* instance (*game* will be a parameter)
- it shall invoke add(*boardPanel*), to register graphically the boardPanel as graphical child of the main frame
- finally, some graphical cosmetics:
    - setResizable(false) //let's not stretch or shrink our pieces
    - pack(); //let's fix size based on board panel content
    - setVisible(true); // time to show up

---

gui.GuiLauncher

Very easy class, here we define our main method as usual, and within it let's declare and instantiate our *GameFrame*. We are finally ready to launch our application!

---

gui.BoardPanel

This class is the main graphical content of our game, and so it requires some effort. Let's get started!

First let's define the attributes:

- a *JPanel* called *main*
- a matrix of *JPanel* called *tiles*
- a *Game* called *game*
- a *GameDynamicsListener* called *listener*
- two integers, *sourceRow* and *sourceColumn* (helper attributes to track user's movements)
- a boolean, called *moveIsOnGoing*
- a static integer *TILE_DIMENSION*, equal to 100 (pixels)

Good, it's time to build our constructor.

- First it shall invoke the *super* constructor, passing a *new BorderLayout()* as parameter.
- It receives the *Game* object as a parameter, let's initialize the class attribute based on it
- It initializes the *moveIsOnGoing* flag to false
- It invokes *initializeLayout and initializeGame* functions (not yet available, let's go ahead and define them)

Now we need to define the following methods:

- *determineTileColor*, that returns a *java.awt.Color* and takes as input a row and a column. The full package is specified for color to distinguish with respect to our own enum Color. The method shall determine and return either jawa.awt.Color.DARK_GRAY or java.awt.Color.WHITE, based on the received row and column
- *highlightSourceTile*, that does not return anything, and receives as parameter a row, a column and a java.awt.Color. It then retrieves the right tile and apply the color, invoking the *setBackground function*
- *clearPiece*, that does not return anything, and receives a row and a column. It retrieves the right tile panel, and on it it invokes *removeAll and updateUI*.
- *drawPiece*, that does not return anything, and receives a row, a column, a *PieceType* and a *Color*. This method is responsible for drawing the right image on the tile based on color and type. It shall compose the icon name as follows:

*icons/**iconName**.png*

iconName starts with W for whites, and B for blacks, and then picks the following letter according to the type:

- P for pawns
- N for knights
- B for bishops
- R for rooks
- K for kings
- Q for queens

for example a black bishop shall be: icons/BB.png.

Once the icon name is ready, let's copy this snippet to draw the icon on the tile:

```
URL url = getClass().getClassLoader().getResource(iconName);

Image image = new ImageIcon(url).getImage();
image = image.getScaledInstance(TILE_DIMENSION, TILE_DIMENSION,
Image.SCALE_SMOOTH);
JLabel label = new JLabel(new ImageIcon(image));
tiles[row][column].removeAll();
tiles[row][column].add(label);
tiles[row][column].updateUI();
```

- *initializeGame*, a method with no parameters or return value. It shall simply iterate through the tiles of the *game board* (through the method *getBoard().GetTile()),* retrieve the piece on each tile (*getPiece()*), and if not null draw it using the *drawPiece* method (type and color can be retrieved from the piece itself)
- *initializeLayout*, whose content is reported here below, since it's very specific to the graphic framework and not really relevant for us. The only very key aspect to note is how the mouse listeners are registered (that is our GameDynamicsListener)

```
private void initializeLayout() {

        main = new JPanel(new GridLayout(Board.Rows, Board.Columns));
        main.setBounds(0, 0, TILE_DIMENSION * Board.Rows, TILE_DIMENSION *
        Board.Columns);

        setPreferredSize(new Dimension(800, 800));
        add(main, BorderLayout.CENTER);

        listener = new GameDynamicsListener(this);

        addMouseListener(listener);
        addMouseMotionListener(listener);

        tiles = new JPanel[Board.Rows][Board.Columns];
        for (int i = 0; i < Board.Rows; i++) {
                for (int j = 0; j < Board.Columns; j++) {
                        tiles[i][j] = new JPanel(new GridLayout(1, 1));
                        tiles[i][j].setPreferredSize(new Dimension(TILE_DIMENSION,
                        TILE_DIMENSION));
                        tiles[i][j].setSize(new Dimension(TILE_DIMENSION,
                        TILE_DIMENSION));
                        tiles[i][j].setBackground(determineTileColor(i, j));
                        tiles[i][j].setVisible(true);

                        main.add(tiles[i][j]);
                }
        }
}
```

- *onMove*, that returns nothing and receives a row and a column. This method is responsible for handling the user's move, stating whether the move starts or ends, highlighting source tile and so on. The pseudo-code looks as follows:

  ```
  if( not moveIsOnGoing) {
          if( game.isMoveSourceValid) { //let's check if move is valid
                  moveIsOnGoing = true
                  highlightSourceTile(java.awt.Color.Green)
                  sourceRow = row
                  sourceColumn = column
          }
  } else {
          moveIsOnGoing =  false
          processMove(sourceRow, sourceColumn, row, column)
          highlightSourceTile( determineTileColor)
  }
  ```

  This method is the callback attached to the user's click. Doesn't it feel like we are getting closer?
- *processMove*, that does not return anything and receives sourceRow, sourceColumn, targetRow and targetColumn. It shall invoke *processMove* on *game*, and store the result, that states if the piece has been moved correctly or not. If it has moved, we shall:

  - retrieve the moved piece from the game target tile
  - *clearPiece* on the tile identified by the source row and column
  - *drawPiece* on the tile identified by the target row and column

That was a long shot, well done! Time to breathe, catch some air, pour some coffee and stretch for a little while.

---

core.Rook

Let’s start concentrating on the piece dynamics, and let’s start by the good old rook. Up and down, seems pretty **straight**-forward (pun intended).

After redefining the constructor, it’s time to override the *validateMove* method.

Its implementation shall appear pretty much as follows:

```
super.validateMove

validMoveDetected = orthogonalMove(move);

if(!validMoveDetected)
{
        throw InvalidMoveException
}
```

orthogonalMove is the other method that we need to implement. It is worth capturing the rooks’ moves in this static method because we will completely reuse them for the Queen!

Let’s then add the static method *orthogonalMove* that returns a boolean and receives the *Move.* It shall

1. determine if a move seems legit for a rook (hence either row or column shall not change). If not, return false
2. if it seems legit, check if there are obstacles along the way (relying on *Move checkObstacles* method). If obstacles are detected, return false; if not , return true

By now we should be convinced that from now on we will only really focus on each *Piece* detail, because all the main reusable logic is already implemented and in place!

---

core.Bishop

The bishop is a bit like the rook, it follows its own path being diagonal rather than horizontal and vertical.

Let’s start by adding an attribute, a boolean flag that states if the bishop is on a white tile or not. *onWhite* seems like a good name.

Once the constructor method is sorted out, let’s override *validateMove* once again.

The implementation is very similar to the Rook’s one, but before invoking *diagonalMove* (the equivalent of *orthogonalMove*) it shall also check if the target tile is of the same color as the bishop’s one (remember the *onWhite* attribute?).

Let’s implement *diagonalMove* based on *orthogonalMove*. The only difference will be bullet 1, hence determining if the bishop’s move seems legit.

Time to knuckle down, and come up with an algorithm!

---

core.Queen

Queen is now very straightforward, since we have rooks and bishops in place, offering their own dynamics in static methods!

Let’s quickly sort out the constructor, and then override the usual *validateMove* method.

Its implementation shall appear as follows:

```
super.validateMove(move);

validMoveDetected = Rook.ortogonalMove OR Bishop.diagonalMove

if(!validMoveDetected)
{
        throw new InvalidMoveException("Invalid move detected");
}
```

---

core.Knight

Knight is a funny one! It literally jumps where it wants, skipping obstacles.

Wait a minute, checking obstacles is what we implemented explicitly in the *Move* class! As long as we don’t check then we are good. Overriding looks really appealing more than ever, doesn’t it?

Let’s quickly sort out the constructor, and then override the usual *validateMove* method.

Its implementation shall appear as follows:

```
super.validateMove(move);

// knight shall not check for obstacles

boolean validMoveDetected = false;

if (move seems legit) {
        validMoveDetected = true;
}
```

```
if(!validMoveDetected)
{
        throw InvalidMoveException;
}
```

Once again, time to sort out some simple math; can you think of a simple way to determine if a knight's move seems legit?

---

core.Pawn

Ah pawns.. Good old pawns. They seem so harmless and innocent. We will see that implementing them is an entirely different story!

They move forward by one step (direction depends on color), on the first movement they can advance two tiles, they capture diagonally (yeah that's right, the only piece that moves and captures differently), they can get promoted, and let's not even think for the time being about the en-passant!

Ok, one step at a time. We need two additional attributes:

- a boolean flag *firstMove*
- a reference to the *Game* object (en-passant is a pain, did I mention that?)

Constructor should then be sorted fairly easily, taking care of the two new attributes as well.

Let's now override the usual *validateMove* method. Its implementation shall appear as follows:

```
super.validateMove

boolean validMoveDetected;

// check two rows up or down
validMoveDetected = tryDoubleAdvanceMove

if (validMoveDetected) {
        // updateEp(move);
        return;
}

// check one row up or down
validMoveDetected = trySingleAdvanceMove
if (validMoveDetected) {
        return;
}
```

```
// check one diagonal left or right, up or down
validMoveDetected = tryDiagonalCapture
if (validMoveDetected) {
        return;
}

throw new InvalidMoveException("Invalid move detected");
```

Advancing by one tile or two tiles is fairly similar; we could really benefit from having a common method *tryForwardMove*.

This method returns a boolean, receives a *Move* and an integer *offset*, to state whether we are advancing by one or two tiles.

The method shall:

- Check if there's a piece on the target tile: a pawn can never move on a tile that contains a piece, even if it belongs to the opponent. It captures in an entirely different way (diagonally). If there's a piece on a target tile, let's return false
- Check if the move is along the column (source column and target column shall be the same)
- If so, it shall check if the target row is equal to the source row plus the given *offset*; let's keep in mind that black pawns go down (hence row increase), while white pawns go up (hence decrease), so pay attention to the offset sign!
- If so, it shall return true
- In any other case, it shall return false

Good, now we can wrap it around and implement the method *trySingleAdvanceMove*: invoking *tryForwardMove* with *offset* set to 1 will do the job!

tryDoubleAdvanceMove is very similar, but before invoking *tryForwardMove* with *offset* set to 2:

- let's make sure that this is the *firstMove* for the pawn
- let's make sure that there are no obstacles in this double advance move (*move.checkObstacles*, what a handy method we've got!)

validateCapture method shall be overridden, since the pawn is the only piece that does not capture in the same way that it moves. Hence let's redefine it and invoke *tryDiagonalCapture*.

executeMove method shall also be overridden; whenever a pawn moves, it no longer can do the double-forward move, reserved for *firstMove* only. The associated flag shall hence be set to true.

Capturing for a pawn implies moving up or down by one, and either left or right; it is hence a diagonal move, by one tile.

Let’s create a common method called *checkDiagonalCapture*, that receives a *Move* and a boolean flag, called *left*.

This method shall:

- determine the column offset of the capture (+1 if towards right, -1 if towards left)
- determine the row offset (+1 if black, -1 if white)
- is it true that the move is targeting a tile with these offsets? (targetColumn == sourceColumn + columnOffset? and what about the row?)
- if so, let’s check if there’s a piece on the target tile to capture! we can rely on the method *move.checkPiecePresence*
- if so, we return true
- in any other case, return false

Good, now the method *tryDiagonalCapture* comes together quite easily. Its pseudocode resembles the following:

```
if (checkDiagonalCapture left) {
    return true
}

if (checkDiagonalCapture right) {
    return true
}

return false;
```

Let’s skip pawn’s promotion and en-passant for the time being, we can come back later to them.

---

core.King

Similarly to pawns, also kings need to distinguish whether they are moving for the first time.

In particular, kings on first move might try to castle.

Let’s start then by defining a boolean attribute, *firstMove*. The attribute shall be initialized to true in the constructor.

As soon as a move is performed, the flag shall be set to false. Let’s override the *executeMove* method, and implement such logic.

Leaving castling aside, kings are able to move in any direction by a single tile.

Let’s define a method called *trySingleStepMove*, that takes as parameter a *Move* object and returns a boolean, stating if the move is valid or not.

This method shall check that the moving offset is not greater than one for either the row or the column, and return true or false accordingly.
Let’s now add the method *tryCastling*, that takes as parameter a *Move* object and returns a boolean, stating if the castling is legit or not. For now let’s not focus on the implementation, let’s simply return false and come back later to it.

Now it’s time to override the usual *validateMove* method. It shall:

- start by relying on the super implementation
- define a boolean flag called *validMoveDetected*, initialized to false
- determine if king is currently in check, by relying on *isKingInCheck* method of the *Move* object
- if the king is not in check, *validMoveDetected* shall take the return value of the invoke of *tryCastling*
- in check or not, *validMoveDetected* shall be assigned using the operator **|=** to the return value of *trySingleStepMove*. The idea is to consider all possible valid outcomes and evaluate them in *OR*; any valid move will do the job!
- Finally, if *validMoveDetected* is still false, well, time to throw our handy *InvalidMoveException*

King’s checks

Now that the main dynamics are sorted, we can focus on the more specific rules.

Let’s start by defining the functionality that verifies if a given move would end with the king being in check. We will rely on this functionality whenever we validate a move, because players are not allowed to sacrifice their own king (that would be a suicide!).

Let’s define the method *wouldEndInKingCheck* in the class *Move*, with no parameters and no return value. The method shall:

- determine the moving piece, by relying as usual on the board and tile methods
- determine the target piece, as in previous bullet, if available
- try and apply the move, to update the chess board as if the move would be performed:
    - the source tile piece is set to null
    - the target tile piece is set to the source moving piece
- now let’s find out where our king is; we iterate through all the tiles and store the king’s row and column
- now that we found out where the king is, let’s invoke the method *isKingInCheck* to see if the move caused a king’s suicide attempt, and store the result in a variable
- whatever the outcome, let’s revert the move:
    - the source tile is set back to host the moving piece

    - the target tile is set back to host the target piece
- based on the outcome, let's throw an *InvalidMoveException*

Now we implement the method *isKingInCheck*. It receives as parameter the king's position (row and column), and the king's color to be checked (will come in handy later on).
It shall:

- determine the opponent's color, with respect to king's color
- now we need to ask ourselves if any opponent's piece has king under siege:
    - let's iterate through every tile of the board
    - let's get the piece on each tile
    - does the piece exist, and if so is it of the opponent's color?
    - also, its type shall not be king. A king cannot menace another's king!
    - if all previous checks are satisfied, we need to verify if the piece can perform a valid move towards the tile's target hosting the king itself
    - if the move is valid, it means that we found a piece that could legitimately menace the king, and put it under threat
    - let's return true and false accordingly

Castling

Castling is a special move that allows moving the king and a rook together, swapping them and putting them close to each other, in order to protect our most valuable piece.

It strictly requires the king not to have moved before (remember the *firstMove* flag in *King*?), and the rook to be on its initial tile.

There's clearly many ways to implement this dynamic, feel free to give it your best try, or alternatively keep on reading to find out a possible implementation.

Let's go back to the *Game* class, and add the following *static* boolean helper attributes:

- blackLongCastlingOnGoing
- blackShortCastlingOnGoing
- whiteLongCastlingOnGoing
- whiteShortCastlingOnGoing

As their name suggests, they state that a given castle move is currently on-going.

In *Game* class we need to define a private method, called *resetCastlingVariables*, that receives a *Move* and simply returns void. It shall:

- retrieve the moving piece from the move's source tile
- if the piece is black, reset the two castling helper variables *blackLongCastlingOnGoing* and *blackLongCastlingOnGoing*
- otherwise, reset the other two

Let’s head back to the *processMove* function; before processing the move (validate and execute), we shall always reset the castling variables (let’s invoke it). That is to say that every new move resets this state.

Now we focus on the method *tryCastling* in *King* class. It is rather long-winded, let’s summarize it in pseudo-code:

```
validMove = false;

if( this is king first move) {

    //determine castling row, based on king's color
    int row = ??

    if(move.getTargetRow() = row) {

        // try long castling
        if(move.getTargetColumn()  is 2) {

            // is rook in place?
            Piece p = move.getPiece(row, 0);

            if(p is valid and is rook and the color matches) {

                // are other pieces in the way?
                obstacleDetected = move.checkPiecePresence(row, 1);
                obstacleDetected &= move.checkPiecePresence(row, 2);
                obstacleDetected &= move.checkPiecePresence(row, 3);

                if(!obstacleDetected)
                {
                        if(color is BLACK)
                            blackLongCastlingOnGoing = true;
                        else
                            whiteLongCastlingOnGoing = true;

                        Piece rook = move.getPiece(row, 0);
                        move.setPiece(row, 3, rook);
                        move.setPiece(row, 0, null);
                        validMove = true;
                }
            }
        }
        else if(move.getTargetColumn() == 6) {
            // is rook in place?
            Piece p = move.getPiece(row, Board.LastColumn);
            if(p is valid and is rook and the color matches) {
```

```
                        // are other pieces in the way?
                        obstacleDetected = move.checkPiecePresence(row, 5);
                        obstacleDetected &= move.checkPiecePresence(row, 6);

                        if(!obstacleDetected) {
                            if(color is BLACK)
                                blackShortCastlingOnGoing = true;
                            else
                                whiteShortCastlingOnGoing = true;

                            Piece rook = move.getPiece(row, Board.LastColumn);
                            move.setPiece(row, 5, rook);
                            move.setPiece(row, Board.LastColumn, null);
                            validMove = true;
                        }
                    }
                }
            }
        }

        return validMove;
```

What this method does is checking if the king is trying to do a long castle or a short castle, verifying if it is the first move, if the rook is in place and no obstacles are met along the way.

Finally, the castling move is a bit special, in the sense that it is the only one that moves two pieces in one shot. Our *BoardPanel* class shall take this into account. Let's go back to the *processMove* method, and invoke a new private function, *checkCastling*, after *drawPiece*.

The *checkCastling* function (feel free to complete it) appears as follows:

```
    if(Game.blackLongCastlingOnGoing) {
        clearPiece(0, 0);
        drawPiece(0, 3, PieceType.ROOK, Color.BLACK);
    } else if(Game.blackShortCastlingOnGoing)  {
        clearPiece(0, Board.LastColumn);
        drawPiece(0, …, … );
    } else if(Game.whiteLongCastlingOnGoing) {
        clearPiece(...);
        drawPiece(...);
    } else if(Game.whiteShortCastlingOnGoing) {
        clearPiece(...);
        drawPiece(...);
    }
```

Now the king is entitled to move in this special way, and the graphic acts accordingly!

Promoting Pawns

Time to focus on pawns' promotion. As you know when a pawn gets to the opposite side of the board it gets promoted to a piece of much higher value, either being a rook, a bishop, a knight or a queen (clearly king is not an option).

Let's define few helper static attributes in the *Game* class:

- *public static int pawnPromotionColumn*
- *public static boolean blackPawnPromotionOnGoing*
- *public static boolean whitePawnPromotionOnGoing*

We need to reset the two boolean flags to false every move: let's implement a function *resetPawnPromotionVariables* to be invoked next to *resetCastlingVariables* for this purpose.

Now let's head back to the *Pawn* class. Whenever a move is executed (overridden method *executeMove*), let's check if the last row is reached.

Last row clearly depends on moving pawn's color. If last row is reached, we need to:

- save which column we are moving along (*Game.pawnPromotionColumn*)
- state that promotion is on-going (*Game.blackPawnPromotionOnGoing* or *Game.whitePawnPromotionOnGoing* set to true)

Finally graphics shall be taken into account as well. A pawn's promotion implies removing a pawn from the army, adding another piece chosen by the player, and redrawing the piece.

Let's implement a new method called *checkPawnPromotion*, to be invoked next to *checkCastling*.

The function shall:

- verify that either black pawn or white pawn promotion is on-going
- if so, ask the user what piece he wants to promote the pawn to; this is achieved with the following code line:

  String promotedPieceType = (String)JOptionPane.showInputDialog(null, "Select promoted piece", "Pawn Promotion", JOptionPane.QUESTION_MESSAGE, null, new String[] {"Bishop", "Rook", "Knight", "Queen"}, "Bishop");

  where:

  - "Select promoted piece" and "Pawn promotion" are the strings displayed in the dialog UI
  - QUESTION_MESSAGE is the type of option dialog
  - the string array contains all possibles choice outcomes
  - "Bishop" is the default choice
  - the result is stored in the variable *promotedPieceType*

Now that we have determined the user's promotion choice, it's time to swap the pawn.

Let's determine the row and the column (based on color and on helper static attributes), and:

- create the new piece, of the right color and type, based on user's choice
- get the pawn from the given tile
- remove this piece from the army (*Game* method *removePieceFromArmy*)
- invoke *clearPiece* to nuke away the pawn graphically
- set the piece on the given tile
- add the new piece to the army (*Game* method *addPieceToArmy*)
- invoke *drawPiece* to draw the newly promoted piece

By doing so, we took care of both the graphical and dynamic aspects of the pawn's promotion rule, wicked!

En-Passant

One more special rule, en-passant!

It's used seldomly and it's fairly complex, but we will manage to sort it out without too much hassle, no worries.

First, as for the last two special moves, we need to head back to *Game* class and add some helper static attributes, visible all-around our project:

- *boolean epLastBlackMove*
- *boolean epLastWhiteMove*
- *boolean epBlackOnGoing*
- *boolean epWhiteOnGoing*
- *int epBlackPawnRow*
- *int epWhitePawnRow*
- *int epBlackPawnColumn*
- *int epWhitePawnColumn*

The first two flags state, respectively, whether the last black (white) move allows the white (black) player to proceed with en-passant.

The second two flags state, respectively, whether the black (white) player is currently doing the en-passant move (useful for the graphics to be aware of what's going on).

The last four attributes are fairly explanatory; they store the row and column of the piece that might be the target of the en-passant rule by the opponent.

Let's add a new method in *Game* called *resetEpVariables*, that takes a *Move* and does not return anything. It shall be invoked next to the other reset routines in this class, and it shall:

- always reset the ep onGoing flags to false
- reset the ep last move flag based on color and current turn

In this way every player will be eligible to apply the en-passant rule just in the right turn, and then these flags will reset to default values.

Let's jump back to the *Pawn* class, that's clearly where the magic happens.

Straight after the tryDoubleAdvanceMove invoke we left over a commented line, referring to function *updateEp*.

We now uncomment and implement this function, that has a void return value and takes a *Move* as parameter.

This function checks the current turn (color), and updates the following flags:

- *Game* ep last move, that shall be set to true
- *Game* ep pawn row, based on move target's row
- *Game* ep pawn column, based on move target's column

We now need to go back to the method *checkDiagonalCapture*.
We used to return true whenever the move row and column were validated, and the presence of the target piece was checked.

Now we need to take into account that, due to en-passant, the piece might not be present; after the *return true* statement we add an *else branch*, in which we:

- check if the target column of the move is equivalent to the ep pawn column stored previously (either *epBlackPawnColumn* or *epWhitePawnColumn*). If so:
  - add a boolean variable *enPassantCapture* that stores the result of the invoke of a new function, *tryEnPassantCapture*
  - if the value returned is *true*, we *return true* as well

We then need to implement this new method, *tryEnPassantCapture*, that returns a *boolean* and takes as parameter a *Move*.

Its pseudo-code resembles the following:

```
enPassantCapture = false;
capturedPiece = null;

if (color is Color.BLACK and epLastWhiteMove) {
    if (move.sourceRow == epWhitePawnRow
        && Math.abs(move.sourceColumn - epWhitePawnColumn) == 1)
    {
        // black en-passant capture
        epBlackOnGoing = true;
```

```
            capturedPiece = move.getPiece(epWhitePawnRow,
epWhitePawnColumn);
            move.setPiece(epWhitePawnRow, epWhitePawnColumn, null);
            enPassantCapture = true;
        }
    } else if (color is Color.WHITE and epLastBlackMove) {
        if (move.sourceRow == epBlackPawnRow
            && Math.abs(move.sourceColumn - epBlackPawnColumn) == 1)
        {
            // white en-passant capture
            epWhiteOnGoing = true;
            capturedPiece = move.getPiece(epBlackPawnRow,
epBlackPawnColumn);
        move.setPiece(epBlackPawnRow, epBlackPawnColumn, null);
        enPassantCapture = true;
        }
    }

    if (enPassantCapture) {
        game.removePieceFromArmy(capturedPiece, game.turn);
    }

    return enPassantCapture;
```

In a nutshell, we can now capture diagonally with pawns if the target pawn is eligible to en-passant capture (as if staying on intermediate tile).

The dynamic side of this rule is sorted, let's sort it out graphically as well.

In *BoardPanel* we need one additional method, *checkEnPassant*. The method returns void and takes no parameter. It simply:

- checks if en-passant capture is on-going (*Game epOnGoing* flags)
- if so, it graphically removes the target pawn, invoking *clearPiece* on target tile (identified by *Game epPawnRow* and *epPawnColumn* attributes)

This function is invoked next to the other *check* routines, within *processMove*.

Conclusion

If you have read and implemented so far, congratulations!

You got yourself a fully functional chess game, pour yourself a cup of tea or coffee, give yourself a nice pat on the shoulder, and go flex with all your friends and relatives!
