**1.**

Привет! Надеюсь ты уже успел устать от обычных задач и с нетерпением ждешь большую!  
Сегодня напишем java реализацию игры **2048**. Вкратце, если ты о ней ничего не слышал, целью игры является получение плитки номиналом **2048** на игровом поле **4х4**. Подробнее можешь прочитать в википедии *https://ru.wikipedia.org/wiki/2048\_(%D0%B8%D0%B3%D1%80%D0%B0)*

**Для начала нам понадобятся такие классы:**  
1. **Controller** - будет следить за нажатием клавиш во время игры.  
2. **Model** - будет содержать игровую логику и хранить игровое поле.  
3. **View** - обеспечит отображение текущего состояния игры на экран.  
4. **Main** - будет содержать только метод main и служить точкой входа в наше приложение.

Создай их!

**Требования:**

* •

Должен быть создан класс Controller.

* •

Должен быть создан класс Model.

* •

Должен быть создан класс View.

* •

Должен быть создан класс Main с публичным статическим методом main (String[] args).

**2.**

В игре **2048** поле состоит из **16** **плиток**, каждая из которых имеет определенный вес.  
Кроме веса у плитки еще будет собственный цвет и цвет текста которым будет отображаться вес плитки.  
Цвета плиток находятся в диапазоне от светло-серого до красного, а цвет текста будет зависеть от цвета плитки.

Создадим класс **Tile** описывающий одну плитку.  
**В нем нам понадобятся:**  
1. Поле value типа **int**, уровень доступа по умолчанию.  
2. Конструктор с параметром, инициализирующий поле value.  
3. Конструктор без параметров (*значение поля***value***должно быть равно нулю*).  
4. Метод isEmpty, возвращающий **true** в случае, если значение поля **value** равно **0**, иначе - **false**.  
5. Метод getFontColor, возвращающий новый цвет(*объект типа Color*) (**0x776e65**) в случае, если вес плитки меньше **16**, иначе - **0xf9f6f2**.  
6. Метод getTileColor, возвращающий цвет плитки в зависимости от ее веса в соответствии с нижеприведенными значениями:  
0: (0xcdc1b4);  
2: (0xeee4da);  
4: (0xede0c8);  
8: (0xf2b179);  
16: (0xf59563);  
32: (0xf67c5f);  
64: (0xf65e3b);  
128: (0xedcf72);  
256: (0xedcc61);  
512: (0xedc850);  
1024: (0xedc53f);  
2048: (0xedc22e);

для любых других значений: (**0xff0000**).

Вышеперечисленные методы не должны быть приватными.

**Требования:**

* •

В классе Tile должно присутствовать поле value типа int с уровнем доступа по умолчанию.

* •

Конструктор класса Tile с одним параметром типа int должен инициализировать поле value.

* •

После создания объекта типа Tile с помощью конструктора без параметров, значение поля value должно быть равно нулю.

* •

Метод isEmpty должен возвращать true в случае, если значение поля value равно 0, иначе - false.

* •

Метод getFontColor должен быть реализован в соответствии с условием задачи.

* •

Метод getTileColor должен возвращать цвет плитки в зависимости от ее веса.

**3.**

Займемся реализацией класса **Model**. Он будет ответственен за все манипуляции производимые с игровым полем.

Но чтобы как-то манипулировать игровым полем, неплохо было бы для начала его создать!

**Нам понадобятся:**  
1. Приватная константа **FIELD\_WIDTH** **= 4**, определяющая ширину игрового поля.  
2. Приватный двумерный массив **gameTiles** состоящий из объектов класса **Tile**.  
3. Конструктор без параметров инициализирующий игровое поле и заполняющий его пустыми плитками.

**Требования:**

* •

В классе Model должно быть создано private static final поле FIELD\_WIDTH со значением равным четырем.

* •

В классе Model должно быть создано private поле gameTiles типа Tile[][].

* •

Конструктор без параметров класса Model должен заполнять массив gameTiles новыми объектами типа Tile.

* •

Массив gameTiles должен иметь размерность FIELD\_WIDTH x FIELD\_WIDTH.

**4.**

Игра 2048 начинается на поле, где две плитки уже имеют какие-то начальные значения. А наше поле пока пусто :(.

Прежде чем бросаться писать код, давай подумаем как это можно было бы реализовать.

Предлагаю создать приватный метод addTile, который будет смотреть какие плитки пустуют и, если такие имеются,

менять вес одной из них, выбранной случайным образом, на 2 или 4 (на 9 двоек должна приходиться 1 четверка).

Получить случайный объект из списка можешь использовав следующее выражение:

(размерСписка \* случайноеЧислоОтНуляДоЕдиницы).

Также получение свободных плиток можно вынести в отдельный приватный метод getEmptyTiles, возвращающий список

свободных плиток в массиве gameTiles.

После реализации функционала добавления новых плиток, добавим в конструктор два вызова метода addTile,

выполняя начальное условие задачи.

P.S. Пожалуй стоит весь код из конструктора переместить в метод resetGameTiles, для того, чтобы при необходимости

начать новую игру, не приходилось создавать новую модель, а можно было бы просто вернуться в начальное состояние

вызвав его. Уровень доступа должен быть шире приватного.

P.P.S. Для вычисления веса новой плитки используй выражение (Math.random() < 0.9 ? 2 : 4).

**Требования:**

* •

Метод getEmptyTiles должен возвращать список пустых плиток в массиве gameTiles.

* •

Метод addTile должен изменять значение случайной пустой плитки в массиве gameTiles на 2 или 4 с вероятностью 0.9 и 0.1 соответственно.

* •

Метод resetGameTiles должен заполнять массив gameTiles новыми плитками и менять значение двух из них с помощью двух вызовов метода addTile.

* •

В конструкторе класса Model должен содержаться вызов метода resetGameTiles.

**5.**

Основными возможностями, которые мы должны реализовать, являются перемещения влево, вправо, вверх и вниз.  
Если ты раньше уже играл в **2048**, то знаешь, что при перемещении в одну из сторон, происходит перемещение плиток со значениями на место пустых, а также объединение плиток одного номинала.

В качестве базового сценария рассмотрим движение влево и подумаем что же происходит, когда мы хотим выполнить это действие.

Для каждого ряда или столбца, происходят на самом деле две вещи:  
а) Сжатие плиток, таким образом, чтобы все пустые плитки были справа, т.е. ряд {4, 2, 0, 4} становится рядом {4, 2, 4, 0}  
б) Слияние плиток одного номинала, т.е. ряд {4, 4, 2, 0} становится рядом {8, 2, 0, 0}.  
Обрати внимание, что ряд {4, 4, 4, 4} превратится в {8, 8, 0, 0}, а {4, 4, 4, 0} в {8, 4, 0, 0}.

Создай методы compressTiles(Tile[] **tiles**) и mergeTiles(Tile[] **tiles**), которые будут реализовывать пункты **а)** и б) соответственно. Использовать мы их будем только внутри класса **Model**, поэтому уровень доступа сделай максимально узким.

Также добавь поля score и maxTile типа int, которые должны хранить текущий счет и максимальный вес плитки на игровом поле. Счет увеличивается после каждого слияния, например если текущий счет **20** и было выполнено слияние ряда {4, 4, 4, 0}, счет должен увеличиться на **8**. Уровень доступа к полям должен быть шире приватного.  
Проще всего организовать обновление значений этих полей в методе mergeTiles, например так:  
1. Если выполняется условие слияния плиток, проверяем является ли новое значения больше максимального и при необходимости меняем значение поля maxTile.  
2. Увеличиваем значение поля score на величину веса плитки образовавшейся в результате слияния.

**P.S.** *Когда мы будем реализовывать методы движения, сжатие будет всегда выполнено перед слиянием, таким образом можешь считать, что в метод mergeTiles всегда передается массив плиток без пустых в середине.*

**Требования:**

* •

Метод mergeTiles должен быть реализован в соответствии с условием задачи.

* •

Метод compressTiles должен быть реализован в соответствии с условием задачи.

* •

Метод compressTiles должен быть приватным.

* •

Метод mergeTiles должен быть приватным.

* •

Метод mergeTiles должен корректно обновлять значение поля score.

* •

Метод mergeTiles должен корректно обновлять значение поля maxTile.

* •

Поля score и maxTile должны быть инициализированы нулем при создании новой модели.

**6.**

Итак, ты реализовал сжатие и слияние плиток, что в комбинации дает нам возможность осуществить движение влево.  
**Отлично!** Но нам нужно еще и добавлять новую плитку в случае, если после передвижения игровое поле изменилось.

**Давай сделаем так:**  
1. Изменим метод compressTiles, чтобы он возвращал true в случае, если он вносил изменения во входящий массив, иначе - **false**.  
2. То же самое сделаем и для метода mergeTiles.  
3. Реализуем метод left, который будет для каждой строки массива **gameTiles** вызывать методы compressTiles и mergeTiles и добавлять одну плитку с помощью метода addTile в том случае, если это необходимо.  
4. Метод left не должен быть приватным, т.к. вызваться он будет, помимо прочего, из класса **Controller**.

**Требования:**

* •

Метод compressTiles должен возвращать true в случае, если им были внесены изменения во входящий массив.

* •

Метод compressTiles должен возвращать false в случае, если им НЕ были внесены изменения во входящий массив.

* •

Метод mergeTiles должен возвращать true в случае, если им были внесены изменения во входящий массив.

* •

Метод mergeTiles должен возвращать false в случае, если им НЕ были внесены изменения во входящий массив.

* •

Метод left должен перемещать элементы массива gameTiles влево в соответствии с правилами игры и добавлять плитку с помощью метода addTile, если это необходимо.

* •

Метод left НЕ должен изменять массив gameTiles, если перемещение влево невозможно.

**7.**

Движение влево мы реализовали, теперь необходимо реализовать методы right, up, down. Уверен, что ты с этим справишься и без моей помощи, так что дам только одну подсказку.

Что будет, если повернуть двумерный массив на **90** градусов по часовой стрелке, сдвинуть влево, а потом еще трижды выполнить поворот?

**Требования:**

* •

Метод up должен перемещать элементы массива gameTiles вверх в соответствии с правилами игры и добавлять плитку с помощью метода addTile, если это необходимо.

* •

Метод up НЕ должен изменять массив gameTiles если перемещение вверх невозможно.

* •

Метод down должен перемещать элементы массива gameTiles вниз в соответствии с правилами игры и добавлять плитку с помощью метода addTile, если это необходимо.

* •

Метод down НЕ должен изменять массив gameTiles если перемещение вниз невозможно.

* •

Метод right должен перемещать элементы массива gameTiles вправо в соответствии с правилами игры и добавлять плитку с помощью метода addTile, если это необходимо.

* •

Метод right НЕ должен изменять массив gameTiles если перемещение вправо невозможно.

**8.**

Итак, модель почти готова, добавим еще пару простых методов и начнем реализацию контроллера.

В модели нам не хватает способа получить игровое поле, чтобы передать его представлению на отрисовку, а также метода, выполнив который, можно было бы определить возможен ли ход в текущей позиции, или нет.

**Контроллер**, в свою очередь, будет в основном использоваться для обработки пользовательского ввода с клавиатуры, поэтому сделаем его наследником класса **KeyAdapter**.

Нам понадобятся приватные поля model и view соответствующих типов и методы getGameTiles и getScore, возвращающие подходящие свойства модели.

**По пунктам:**  
1. Добавь в класс **Model** геттер для поля gameTiles.  
2. Добавь в класс **Model** метод canMove возвращающий **true** в случае, если в текущей позиции возможно сделать ход так, чтобы состояние игрового поля изменилось. Иначе - **false.**  
3. Сделай класс **Controller** потомком класса **KeyAdapter**.  
4. Добавь в класс **Controller** метод getGameTiles вызывающий такой же метод у модели.  
5. Добавь в класс **Controller** метод getScore возвращающий текущий счет (***model.score***).

**Требования:**

* •

В классе Model должен быть реализован корректный геттер для поля gameTiles.

* •

Метод canMove в классе Model должен быть реализован в соответствии с условием задачи.

* •

Класс Controller должен быть потомком класса KeyAdapter.

* •

Метод getGameTiles в классе Controller должен возвращать результат вызова метода getGameTiles у модели.

* •

Метод getScore в классе Controller должен возвращать значение поля score модели.

**9.**

Ты отлично справляешься! Так хорошо, что я решил тебе немного помочь и уже реализовал класс **View**.  
Он достаточно прост. Наследуемся от класса **JPanel**, переопределяем метод paint и выводим на экран  
текущее состояние модели, полученное через контроллер.

Тебе же, предстоит закончить реализацию класса **Controller**.

Для начала нам понадобится конструктор, он будет принимать один параметр типа **Model**, инициализировать поле model, а также сохранять в поле view новый объект типа **View** с текущим контроллером(**this**) в качестве параметра конструктора.

Далее, нам нужен метод resetGame, который позволит вернуть игровое поле в начальное состояние. Необходимо обнулить счет, установить флаги **isGameWon** и **isGameLost** у представления в **false** и вызывать метод resetGameTiles у модели.

Примечание: устанавливай значение полей напрямую, без использования сеттеров.

Добавим приватную константу int **WINNING\_TILE** = 2048. Она будет определять вес плитки при достижении которого игра будет считаться выигранной.

Ну а теперь, самое главное! Для того чтобы иметь возможность обрабатывать пользовательский ввод, необходимо переопределить метод keyPressed с одним параметром типа **KeyEvent**.

**Логика метода должна быть следующей:**  
1. Если была нажата клавиша **ESC** - вызови метод resetGame.  
2. Если метод canMove модели возвращает **false** - установи флаг **isGameLost** в **true**.  
3. Если оба флага **isGameLost** и **isGameWon** равны **false** - обработай варианты движения:  
а) для клавиши **KeyEvent.VK\_LEFT** вызови метод left у модели;  
б) для клавиши **KeyEvent.VK\_RIGHT** вызови метод right у модели;  
в) для клавиши **KeyEvent.VK\_UP** вызови метод up у модели;  
г) для клавиши **KeyEvent.VK\_DOWN** вызови метод down у модели.  
4. Если поле maxTile у модели стало равно **WINNING\_TILE**, установи флаг **isGameWon** в **true**.  
5. В самом конце, вызови метод repaint у view.

**P.S.** *Для получения кода нажатой клавиши используй метод* **getKeyCode** *класса* **KeyEvent**.

**Требования:**

* •

В классе Controller должна быть создана приватная статическая константа int WINNING\_TILE = 2048.

* •

Конструктор класса Controller с одним параметром типа Model должен быть реализован в соответствии с условием задачи.

* •

Метод resetGame должен возвращать игру в начальное состояние, как описано в условии задачи.

* •

Метод keyPressed должен вызывать метод resetGame в случае, если была нажата клавиша ESC.

* •

Метод keyPressed должен устанавливать флаг isGameLost в true в случае, если ход невозможен.

* •

Метод keyPressed должен вызывать корректные методы перемещения игрового поля, в случае если была нажата подходящая клавиша и оба флага isGameLost и isGameWon равны false.

* •

Метод keyPressed должен устанавливать флаг isGameWon равным true в случае, если значения полей model.maxTile и WINNING\_TILE стали равны после передвижения.

* •

Метод keyPressed должен вызывать метод repaint у объекта сохраненного в поле view.

**10.**

Пора приступить к реализации метода main в классе **Main**, чтобы иметь возможность наконец-то запустить игру и отдохнуть!

Метод main нам нужен только для того чтобы запустить приложение, все внутренности мы уже реализовали.  
Для этого мы создадим в нем модель и контроллер, а также объект типа JFrame. Для примера я назову его **game**, но ты можешь выбрать любое другое имя.

У нашей игры (о*бъекта типа****JFrame***) мы должны будем вызвать некоторые методы для того чтобы все корректно отображалось на экране:

game.setTitle("2048");  
game.setDefaultCloseOperation(WindowConstants.EXIT\_ON\_CLOSE);  
game.setSize(450, 500);  
game.setResizable(false);  
  
game.add(controller.getView());  
  
  
game.setLocationRelativeTo(null);  
game.setVisible(true);

Обрати внимание на метод add в который мы передаем представление из контроллера. У нас еще нет геттера для поля view в классе **Controller**. Не забудь его добавить.

**P.S.** *Результатом выполнения этого задания будет рабочая версия игры***2048***, если у тебя вдруг что-то не работает, или работает не так как ожидалось, обязательно разберись и исправь прежде чем переходить к следующим задачам.*

**Требования:**

* •

В классе Controller должен быть создан корректный геттер для поля view.

* •

В методе main класса Main должна быть создана новая модель и контроллер на базе этой модели.

* •

В методе main класса Main должен быть создан объект типа JFrame.

* •

В методе main класса Main на объекте JFrame должны быть выполнены методы перечисленные в условии задачи.

**11.**

Отличная работа! На этом этапе у нас уже есть полнофункциональное приложение, но ведь нет предела совершенству, давай еще поработаем.

Если ты успел какое-то время поиграть в **2048**, то заметил, что порой очень хочется иметь возможность отменить последний ход.

Давай создадим в классе **Model** два стека, в одном будем хранить предыдущие состояния игрового поля, а в другом предыдущие счета. Назовем их **previousStates** и **previousScores**. Инициализировать можешь прямо в строке объявления или в конструкторе. Используй стандартную реализацию стека (***java.util.Stack***).

Добавим **boolean** поле **isSaveNeeded** = true, оно нам понадобится в будущем.

Хранилище состояний у нас есть, теперь реализуем два метода для работы с ними.  
1. Приватный метод saveState с одним параметром типа **Tile[][]** будет сохранять текущее  
игровое состояние и счет в стеки с помощью метода push и устанавливать флаг **isSaveNeeded** равным **false**.  
2. Публичный метод rollback будет устанавливать текущее игровое состояние равным последнему находящемуся в стеках с помощью метода pop.

Обрати внимание на то, что при сохранении массива **gameTiles** необходимо создать новый массив и заполнить его новыми объектами типа **Tile** перед сохранением в стек.

В методе rollback достаточно просто выполнить присваивание (**gameTiles***=*previousStates.pop()) и то же для счета, нет необходимости в глубоком копировании.

Перед восстановлением игрового состояния с помощью метода rollback не забудь проверить что стеки не пусты, чтобы избежать возникновения исключения **EmptyStackException**.

**Требования:**

* •

В классе Model должны быть объявлены и инициализированы приватные поля previousStates, previousScores, isSaveNeeded.

* •

Метод saveState должен сохранять в стек previousStates новый объект типа Tile[][] с помощью метода push.

* •

После вызова метода saveState веса плиток в массиве который находится на вершине стека должны совпадать с весами плиток массива полученного в качестве параметра.

* •

Метод saveState должен сохранять в стек previousScores текущее значение поля score с помощью метода push.

* •

Метод saveState должен устанавливать флаг isSaveNeeded равным false.

* •

Метод rollback должен восстанавливать поля score и gameTiles из соответствующих стеков, если они не пусты.

* •

Метод rollback не должен модифицировать текущее игровое состояние в случае, если хотя бы один из стеков пуст.

* •

Каждый вызов метода saveState должен увеличивать количество элементов в стеках на единицу.

* •

Каждый вызов метода rollback должен уменьшать количество элементов в стеках на единицу, до тех пор пока это возможно.

**12.**

Ну что, попробуем наш алгоритм в действии? Осталось добавить сохранение игрового состояния в начало каждого метода движения, а также еще один кейс для обработки клавиши, которой будем выполнять отмену последнего хода.

При сохранении текущего состояния в стек, обрати внимание на то, чтобы всегда сохранялось актуальное состояние и только однажды. Если ты послушал мой совет и реализовал методы right, up, down с помощью поворотов и вызова метода left, можешь использовать следующий подход:  
1. В самом начале методов right, up, down вызываем метод saveState с **gameTiles** в качестве параметра.  
2. В методе left организуем проверку того, вызывался ли уже метод saveState. За это у нас отвечает флаг **isSaveNeeded**, соответственно, если он равен **true**, выполняем сохранение. После выполнения сдвига влево устанавливаем флаг **isSaveNeeded** равным **true**.

Также добавим в метод keyPressed класса **Controller** вызов метода rollback по нажатию на клавишу **Z** (*код -* KeyEvent.VK\_Z).

**Требования:**

* •

Метод keyPressed класса Controller должен вызывать метод rollback у модели в случае, если была нажата клавиша с кодом KeyEvent.VK\_Z.

* •

Метод left должен один раз сохранять текущее игровое состояние и счет в соответствующие стеки.

* •

Метод right должен один раз сохранять текущее игровое состояние и счет в соответствующие стеки.

* •

Метод up должен один раз сохранять текущее игровое состояние и счет в соответствующие стеки.

* •

Метод down должен один раз сохранять текущее игровое состояние и счет в соответствующие стеки.

**13.**

Твой прогресс впечатляет! Для разнообразия, предлагаю дать игре возможность самостоятельно  
выбирать следующий ход.

Начнем с простого, реализуем метод randomMove в классе **Model**, который будет вызывать один из методов движения случайным образом. Можешь реализовать это вычислив целочисленное n = ((int) (Math.random() \* 100)) % 4.  
Это число будет содержать целое псевдослучайное число в диапазоне **[0..3]**, по каждому из которых можешь вызывать один из методов left, right, up, down.

Не забудь добавить в метод keyPressed класса **Controller** вызов метода randomMove по нажатию на клавишу **R**(*код -* KeyEvent.VK\_R).

**P.S.** *Проверку корректности работы метода*randomMove*оставляю полностью под твою ответственность, проверю только наличие вызова метода* Math.random().

**Требования:**

* •

У класса Model должен быть метод void randomMove().

* •

Метод keyPressed класса Controller должен вызывать метод randomMove у модели в случае, если была нажата клавиша с кодом KeyEvent.VK\_R.

* •

Метод randomMove должен использовать статический метод random класса Math.

**14.**

Случайный ход конечно неплох, но намного круче реализовать возможность умного хода. В дебри нейронных сетей мы заходить не будем, для начала сконцентрируемся на достаточно простой идее.

Очевидно, хороший ход должен в итоге приближать нас к победе, а именно к получению плитки **2048**.

**Предлагаю рассмотреть такой вариант сравнения эффективности хода:**  
1. Первый ход является лучше второго, если после его совершения на поле находится больше пустых плиток, чем в результате второго хода.  
2. Первый ход является лучше второго, если общий счет после его совершения больше, чем счет  
полученный в результате второго хода.

Для того, чтобы реализовать такое сравнение, мы можем совершить ход, оценить его эффективность и потом отменить совершенный ход, чтобы вернуть игру в начальное состояние. Применив эту последовательность действий ко всем четырем вариантам хода, сможем выбрать наиболее эффективный ход и выполнить его.

Концептуально, нам понадобятся два класса, один будет описывать ход, а другой эффективность хода.

Создай интерфейс **Move** с одним **void** методом move. Отметь интерфейс аннотацией ***@FunctionalInterface***, которая будет сигнализировать о том что в этом интерфейсе будет только один абстрактный метод.

Создай класс **MoveEfficiency**, описывающий эффективность хода. В нем нам понадобятся приватные поля numberOfEmptyTiles и score типа int, а также приватное поле поле move типа **Move**.  
В классе **MoveEfficiency** необходим конструктор с тремя параметрами (int numberOfEmptyTiles, int score, Move move) для инициализации полей класса и **геттер** для поля move.

**Требования:**

* •

В интерфейсе Move должен присутствовать один абстрактный void метод move.

* •

Интерфейс Move должен быть отмечен аннотацией @FunctionalInterface.

* •

Конструктор класса MoveEfficiency должен корректно инициализировать поля класса.

* •

В классе MoveEfficiency должен быть создан корректный геттер для поля move.

**15.**

Для того, чтобы эффективности различных ходов можно было сравнивать, необходимо реализовать в классе **MoveEfficiency** поддержку интерфейса **Comparable**.

В методе compareTo первым делом сравни количество пустых плиток (**numberOfEmptyTiles**), потом счет (**score**), если количество пустых плиток равное. Если и счет окажется равным, будем считать эффективность ходов равной и вернем ноль.

**Далее перейдем в класс Model и реализуем два метода:**  
1) boolean hasBoardChanged - будет возвращать **true**, в случае, если вес плиток в массиве **gameTiles** отличается от веса плиток в верхнем массиве стека **previousStates**. Обрати внимание на то, что мы не должны удалять из стека верхний элемент, используй метод peek.  
2) MoveEfficiency getMoveEfficiency(Move **move**) - принимает один параметр типа move, и возвращает объект типа **MoveEfficiency** описывающий эффективность переданного хода. Несколько советов:  
а) не забудь вызывать метод rollback, чтобы восстановить корректное игровое состояние;  
б) в случае, если ход не меняет состояние игрового поля, количество пустых плиток и счет у объекта **MoveEfficiency** сделай равными **-1** и **0** соответственно;  
в) выполнить ход можно вызвав метод move на объекте полученном в качестве параметра.

**Требования:**

* •

Класс MoveEfficiency должен поддерживать интерфейс Comparable.

* •

Метод compareTo должен корректно сравнивать два объекта типа MoveEfficiency.

* •

Метод hasBoardChanged должен быть реализован в соответствии с условием задачи.

* •

Метод getMoveEfficiency должен возвращать эффективность хода полученного в качестве параметра.

* •

Если ход, переданный в метод getMoveEfficiency не меняет игровое поле, должен быть возвращен объект с количеством пустых клеток равным -1.

* •

Метод getMoveEfficiency не должен менять вес плиток в массиве gameTiles и счет.

**16.**

Осталось совсем немного! У нас есть способ вычислить эффективность любого хода, а также мы можем их сравнивать между собой.

Давай реализуем метод autoMove в классе **Model**, который будет выбирать лучший из возможных ходов и выполнять его.

**Сделаем так:**  
1) Создадим локальную PriorityQueue с параметром Collections.reverseOrder() (*для того, чтобы вверху очереди всегда был максимальный элемент*) и размером равным четырем.  
2) Заполним PriorityQueue четырьмя объектами типа **MoveEfficiency** (*по одному на каждый вариант хода)*.  
3) Возьмем верхний элемент и выполним ход связанный с ним.

После реализации метода autoMove добавим его вызов в метод keyPressed класса **Controller** по нажатию на клавишу **A** (код - KeyEvent.VK\_A).

**P.S.** *В качестве факультативного задания можешь почитать про указатели на методы и попробовать передать аргумент в метод getMoveEfficiency используя оператор "****::****". Для метода*left*должно получиться*getMoveEfficiency(this::left)*. Альтернативно можешь использовать внутренний анонимный класс.*

**Требования:**

* •

В методе autoMove должен быть создан объект типа PriorityQueue с размером равным четырем.

* •

В методе autoMove в PriorityQueue должно быть добавлено 4 объекта типа MoveEfficiency с помощью метода offer (по одному на каждый вариант хода).

* •

Метод keyPressed класса Controller должен вызывать метод autoMove у модели в случае, если была нажата клавиша с кодом KeyEvent.VK\_A.

* •

В методе autoMove должен быть выполнен метод move связанный с объектом MoveEfficiency полученном с помощью метода peek или poll.

**17.**

Поздравляю с реализацией своей собственной версии игры **2048**!

Помимо основного функционала ты также реализовал отмену последнего хода и автоматический выбор наилучшего хода с помощью оценки эффективности одиночного хода.

Из возможных улучшений можешь попробовать увеличить глубину анализа эффективности хода и проверить, сможет ли твой алгоритм набрать максимально возможный счет в **839,732** очков.

**Требования:**

* •

Поздравляю, ты отлично справился!