**Нам, понадобятся:**

* Настроенная рабочая среда, т.е. [JDK](http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html) и IDE (например, [Eclipse](https://www.eclipse.org/downloads/packages/eclipse-ide-java-developers/mars2" \t "_blank));
* Библиотека LWJGL (версии 2.x.x) для работы с графикой (опционально). Обратите внимание, что для LWJGL версий выше 3 потребуется написать код, отличающийся от того, что приведён в статье;
* Спрайты, т.е. картинки плиток всех возможных состояний (пустая, и со степенями двойки до 2048). Можно нарисовать самому, или скачать

**С чего начать?**

Начать стоит с главного управляющего класса, который в нашем проекте находится выше остальных по уровню абстракции. Вообще отличный совет — в начале работы всегда пишите код вида if(getKeyPressed()) doSomething(), так вы быстро определите фронт работ.

/\*\*

\* Точка входа. Содержит все необходимые действия для одного игрового цикла.

\*/

public static void main(String[] args) {

initFields();

createInitialCells();

while(!endOfGame){

input();

logic();

graphicsModule.draw(gameField);

}

graphicsModule.destroy();

}

Это наш main(). Что тут происходит, понять несложно — мы инициализируем поля, потом создаём первые две ячейки и, пока игра не закончится, осуществляем по очереди: ввод пользовательских данных (input()), основные игровые действия (logic()) и вызов метода отрисовки у графического модуля (graphicsModule.draw()), в который передаём текущее игровое поле (gameField).

Так как пока мы не знаем, какие поля инициировать, постараемся написать createInitialCells(). Но так как создавать клетки нам пока просто-напросто не в чем, то создадим класс игрового поля.

**Создаём игровое поле**

Всё наше поле — матрица чисел и методы, позволяющие их изменять (геттеры и сеттеры). Договоримся только, что пустую ячейку мы будем обозначать числом 0. Выглядеть этот класс будет так:

public class GameField {

/\*\*

\* Состояние всех ячеек поля.

\*/

private int[][] theField;

/\*\*

\* Инициализирует поле и заполняет его нулями

\*/

public GameField(){

theField = new int[COUNT\_CELLS\_X][Constants.COUNT\_CELLS\_Y];

for(int i=0; i<theField.length;i++){

for(int j=0; j<theField[i].length; j++){

theField[i][j]=0;

}

}

}

/\*\*

\* Возвращает состояние ячейки поля по координатам

\*

\* @param x Координата ячейки X

\* @param y Координата ячейки Y

\* @return Состояние выбранной ячейки

\*/

public int getState(int x, int y){

return theField[x][y];

}

/\*\*

\* Изменяет состояние ячейки поля по координатам

\*

\* @param x Координата ячейки X

\* @param y Координата ячейки Y

\* @param state Новое состояние для этой ячейки

\*/

public void setState(int x, int y, int state){

//TODO check input maybe?

theField[x][y] = state;

}

/\*\*

\* Изменяет столбец под номером i

\*

\* @param i Номер изменяемого столбца

\* @param newColumn Массив новых состояний ячеек столбца

\*/

public void setColumn(int i, int[] newColumn) {

theField[i] = newColumn;

}

/\*\*

\* Возвращает массив состояний ячеек столбца под номером i

\*

\* @param i Номер запрашиваемого столбца

\* @return Массив состояний ячеек столбца

\*/

public int[] getColumn(int i) {

return theField[i];

}

/\*\*

\* Изменяет строку под номером i

\*

\* @param i Номер изменяемой строки

\* @param newLine Массив новых состояний ячеек строки

\*/

public void setLine(int i, int[] newLine) {

for(int j = 0; j< COUNT\_CELLS\_X; j++){

theField[j][i] = newLine[j];

}

}

/\*\*

\* Возвращает массив состояний ячеек строки под номером i

\*

\* @param i Номер запрашиваемой строки

\* @return Массив состояний ячеек строки

\*/

public int[] getLine(int i) {

int[] ret = new int[COUNT\_CELLS\_X];

for(int j = 0; j< COUNT\_CELLS\_X; j++){

ret[j] = theField[j][i];

}

return ret;

}

}

Возможно, пока не совсем очевидно, почему нужны именно такие геттеры и сеттеры, это станет ясно в процессе дальнейшей работы (при разработке с нуля следовало бы создать только getState() и setState(), а остальное дописывать потом).

**Создаём в поле первые две ячейки**

Совсем очевидно, что нам нужно просто вызвать два раза метод создания одной ячейки.

/\*\*

\* Создаёт на поле начальные ячейки

\*/

private static void createInitialCells() {

for(int i = 0; i < COUNT\_INITITAL\_CELLS; i++){

generateNewCell();

}

}

Заметьте, я не пишу вызов одного метода два раза. Для программистов существует одна максима: «Существует только два числа: один и много». Чаще всего, если что-то нужно сделать 2 раза, то со временем может возникнуть задача сделать это и 3, и 4 и куда больше раз. Например, если вы решите сделать поле не 4х4, а 10х10, то разумно будет создавать не 2, а 10 ячеек.

Вы могли заметить, что в коде использована константа COUNT\_INITIAL\_CELLS. Все константы удобно определять в классе с public static final полями. Полный список констант, который нам потребуется в ходе разработки, можно посмотреть в классе [Constants](https://github.com/DoKel/tprogers2048game/" \t "_blank) на GitHub.

Теперь постараемся решить вопрос — как в матрице создать ячейку вместо одного из нулей? Я решил пойти по такому пути: мы выбираем случайные координаты, и если там находится пустая ячейка, то создаём новую плитку там. Если там уже есть плитка с числом, то пытаемся создать в следующей клетке (двигаемся вправо и вниз). Обратите внимание, что после хода не может не быть пустых клеток, т.к. ход считается сделанным, когда клетки либо переместились (т.е. освободили какое-то место), либо соединились (т.е. клеток стало меньше, и место снова высвободилось).

private static void generateNewCell() {

int state = (new Random().nextInt(100) <= Constants.CHANCE\_OF\_LUCKY\_SPAWN)

? LUCKY\_INITIAL\_CELL\_STATE

: INITIAL\_CELL\_STATE;

int randomX, randomY;

randomX = new Random().nextInt(Constants.COUNT\_CELLS\_X);

int currentX = randomX;

randomY = new Random().nextInt(Constants.COUNT\_CELLS\_Y);

int currentY = randomY;

boolean placed = false;

while(!placed){

if(gameField.getState(currentX, currentY) == 0) {

gameField.setState(currentX, currentY, state);

placed = true;

}else{

if(currentX+1 < Constants.COUNT\_CELLS\_X) {

currentX++;

}else{

currentX = 0;

if(currentY+1 < Constants.COUNT\_CELLS\_Y) {

currentY++;

}else{

currentY = 0;

}

}

if ((currentX == randomX) && (currentY==randomY) ) { //No place -> Something went wrong

ErrorCatcher.cellCreationFailure();

}

}

}

score += state;

}

Немного более затратен по времени и памяти другой метод, который тоже имеет право на жизнь. Мы складываем в какую-либо коллекцию (например, ArrayList) координаты всех ячеек с нулевым значением (простым перебором). Затем делаем new Random().nextInt(X), где X — размер это коллекции, и создаём ячейку по координатам, указанным в члене коллекции с номером, соответствующем результату.

**Реализуем пользовательский ввод**

Следующим по очереди у нас идёт метод input(). Займёмся им.

private static void input() {

keyboardModule.update();

/\* Определяем направление, в котором нужно будет произвести сдвиг \*/

direction = keyboardModule.lastDirectionKeyPressed();

endOfGame = endOfGame || graphicsModule.isCloseRequested() || keyboardModule.wasEscPressed();

}

Отсюда нам нужно запомнить только, какие интерфейсы (графический и клавиатурный модули) нам нужно создать и какие методы в них определить. Если не запомнили — не волнуйтесь, ворнинги вашей IDE особо забыть не дадут.

**Интерфейсы для клавиатурного и графического модулей**

Так как многим не нравится, что я пишу эти модули на LWJGL, я решил в статье уделить время только интерфейсам этих классов. Каждый может написать их с помощью той GUI-библиотеки, которая ему нравится (или вообще сделать консольный вариант). Я же по старинке реализовал их на LWJGL, код можно посмотреть [здесь](https://github.com/DoKel/tprogersTetris/tree/master/src/ru/tproger/graphics/lwjglmodule) в папках graphics/lwjglmodule и keyboard/lwjglmodule.

Интерфейсы же, после добавления в них всех упомянутых выше методов, будут выглядеть следующим образом:

**Графический модуль**

public interface GraphicsModule {

/\*\*

\* Отрисовывает переданное игровое поле

\*

\* @param field Игровое поле, которое необходимо отрисовать

\*/

void draw(GameField field);

/\*\*

\* @return Возвращает true, если в окне нажат "крестик"

\*/

boolean isCloseRequested();

/\*\*

\* Заключительные действия, на случай, если модулю нужно подчистить за собой.

\*/

void destroy();

}

**Клавиатурный модуль**

public interface KeyboardHandleModule {

/\*\*

\* Считывание последних данных из стека событий, если можулю это необходимо

\*/

void update();

/\*\*

\* @return Возвращает направление последней нажатой "стрелочки",

\* либо AWAITING, если не было нажато ни одной

\*/

ru.tproger.main.Direction lastDirectionKeyPressed();

/\*\*

\* @return Возвращает информацию о том, был ли нажат ESCAPE за последнюю итерацию

\*/

boolean wasEscPressed();

}

**Метод логики**

Вполне понятно, что если было определено направление сдвига, то нужно произвести в этом направлении сдвиг — в этом вся суть игры. Также, если сдвиг произвести удалось, необходимо создать новую ячейку. Направление для нового сдвига должно снова стать неопределённым — до следующего пользовательского ввода.

private static void logic() {

if(direction!=Direction.AWAITING){

if(shift(direction)) generateNewCell();

direction=Direction.AWAITING;

}

}

Вы могли заметить, что мы часто используем enum Direction для определения направления. Т.к. его используют различные классы, он вынесен в отдельный файл и выглядит так:

public enum Direction {

AWAITING, UP, DOWN, LEFT, RIGHT

}

**Давай уже серьёзно. Как нам сдвинуть это чёртово поле?**

Самое ядро нашего кода! Вот самое-самое. К слову, спорный вопрос, куда поместить этот метод — в Main.java или в GameField.java? Я выбрал первое, но это решение нельзя назвать слишком обдуманным. Жду ваше мнение в комментариях.

Очевидно, что должен быть какой-то алгоритм сдвига линии, который должен применяться к каждому столбцу (или строке, зависит от направления) по очереди и менять значения необходимым нам образом. К этому алгоритму мы и будем обращаться из Main.shift(). Так же такой алгоритм (вынесенный в метод) должен определять, изменил он что-то или не изменил, чтобы метод shift() это значение мог вернуть.

/\*\*

\* Изменяет gameField, сдвигая все ячейки в указанном направлении,

\* вызывая shiftRow() для каждой строки/столбца (в зависимости от направления)

\*

\* @param direction Направление, в котором необходимо совершить сдвиг

\* @return Возвращает true, если сдвиг прошёл успешно (поле изменилось)

\*/

private static boolean shift(Direction direction) {

boolean ret = false;

switch(direction) {

case UP:

case DOWN:

/\*По очереди сдвигаем числа всех столбцов в нужном направлении\*/

for(int i = 0; i< Constants.COUNT\_CELLS\_X; i++){

/\*Запрашиваем очередной столбец\*/

int[] arg = gameField.getColumn(i);

/\*В зависимости от направления сдвига, меняем или не меняем порядок чисел на противоположный\*/

if(direction==Direction.UP){

int[] tmp = new int[arg.length];

for(int e = 0; e < tmp.length; e++){

tmp[e] = arg[tmp.length-e-1];

}

arg = tmp;

}

/\*Пытаемся сдвинуть числа в этом столбце\*/

ShiftRowResult result = shiftRow (arg);

/\*Возвращаем линию в исходный порядок\*/

if(direction==Direction.UP){

int[] tmp = new int[result.shiftedRow.length];

for(int e = 0; e < tmp.length; e++){

tmp[e] = result.shiftedRow[tmp.length-e-1];

}

result.shiftedRow = tmp;

}

/\*Записываем изменённый столбец\*/

gameField.setColumn(i, result.shiftedRow);

/\*Если хоть одна линия была изменена, значит было изменено всё поле\*/

ret = ret || result.didAnythingMove;

}

break;

case LEFT:

case RIGHT:

/\*По очереди сдвигаем числа всех строк в нужном направлении\*/

for(int i = 0; i< Constants.COUNT\_CELLS\_Y; i++){

/\*Запрашиваем очередную строку\*/

int[] arg = gameField.getLine(i);

/\*В зависимости от направления сдвига, меняем или не меняем порядок чисел на противоположный\*/

if(direction==Direction.RIGHT){

int[] tmp = new int[arg.length];

for(int e = 0; e < tmp.length; e++){

tmp[e] = arg[tmp.length-e-1];

}

arg = tmp;

}

/\*Пытаемся сдвинуть числа в этом столбце\*/

ShiftRowResult result = shiftRow (arg);

/\*Возвращаем линию в исходный порядок\*/

if(direction==Direction.RIGHT){

int[] tmp = new int[result.shiftedRow.length];

for(int e = 0; e < tmp.length; e++){

tmp[e] = result.shiftedRow[tmp.length-e-1];

}

result.shiftedRow = tmp;

}

/\*Записываем изменённую строку\*/

gameField.setLine(i, result.shiftedRow);

/\*Если хоть одна линия была изменена, значит было изменено всё поле\*/

ret = ret || result.didAnythingMove;

}

break;

default:

ErrorCatcher.shiftFailureWrongParam();

break;

}

return ret;

}

Так как этот магический метод с алгоритмом должен будет по сути вернуть два объекта (новую линию и boolean, который будет говорить о наличии изменений в ней), создадим в начале класса Main для такого результата обёртку:

/\*\*

\* Результат работы метода сдвига shiftRow().

\* Содержит изменённую строку и информацию о том, эквивалентна ли она начальной.

\*/

private static class ShiftRowResult{

boolean didAnythingMove;

int[] shiftedRow;

}

Можно, конечно. просто возвращать линию, а затем сравнивать её (не забываем, что это нужно делать через метод equals(), а не через ==), но на это будет уходит больше времени (из-за сравнение каждого элемента массива), но меньше памяти (на один boolean).

**Самое сердце программы. Метод shiftRow()**

Если подумать, то вам предстоит решить задачку — как за наименьшее (линейно зависящее от количества поступающих данных) время произвести с рядом чисел следующие последовательные операции: (1) если в ряде есть нули, их необходимо удалить, (2) если любые два соседних числа равны, то вместо них должно остаться одно число, равное сумме двух равных чисел. И (3) — если число получено через пункт (2), оно не может совмещаться с другими числами.

Если представить себе алгоритм таким образом, то придумать линейное решение будет гораздо легче. Вот какой алгоритм должен получиться:

* **Выкидываем все нули** — проходимся по всему массиву и копируем элемент в новый массив, только если он не равен нулю. Если вы попробуете удалять эти нули из середины того же массива, алгоритм будет работать за O(n^2).
* **Рассмотрим (поставим указатель на) первое число получившегося массива без нулей**.
  1. Если с ним можно совместить следующее за ним число (наш указатель +1), то переписываем в новый массив лишь их сумму, затем ставим указатель на третье число (второго уже нет).
  2. Иначе переписываем только первое, и ставим указатель на второе число.

При этом нам необходимо хранить место в возвращаемом массиве, на которое необходимо произвести запись.

А вот как он выглядит в виде кода:

private static ShiftRowResult shiftRow (int[] oldRow) {

ShiftRowResult ret = new ShiftRowResult();

int[] oldRowWithoutZeroes = new int[oldRow.length];

{

int q = 0;

for (int i = 0; i < oldRow.length; i++) {

if(oldRow[i] != 0){

if(q != i){

/\*

\* Это значит, что мы передвинули ячейку

\* на место какого-то нуля (пустой плитки)

\*/

ret.didAnythingMove = true;

}

oldRowWithoutZeroes[q] = oldRow[i];

q++;

}

}

}

ret.shiftedRow = new int[oldRowWithoutZeroes.length];

{

int q = 0;

{

int i = 0;

while (i < oldRowWithoutZeroes.length) {

if((i+1 < oldRowWithoutZeroes.length) && (oldRowWithoutZeroes[i] == oldRowWithoutZeroes[i + 1])

&& oldRowWithoutZeroes[i]!=0) { {

ret.didAnythingMove = true;

ret.shiftedRow[q] = oldRowWithoutZeroes[i] \* 2;

i++;

} else {

ret.shiftedRow[q] = oldRowWithoutZeroes[i];

}

q++;

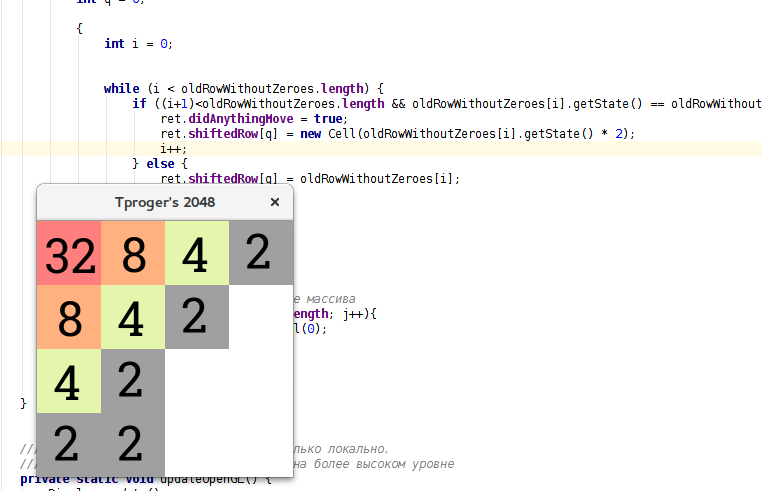
i++;

}

}

}

**Наслаждаемся результатом**

[](https://media.tproger.ru/uploads/2016/07/Screenshot-from-2016-07-05-05-53-57.png)

Работающая программа