Lab assignment 4

Due date: Tuesday, June 7

40 points

Team of 4 students

Divide the team into two subgroups with two team members per group.

The group works on the lab assignment and can discuss the code with another group. Each group demonstrates the lab to each other. The team then upload the solution from one of the subgroups.

Write the contribution to the lab assignment, the contribution percentage (maximum of 100%) on the comment box for each team member.

All the team members must agree on the percentage. If the team members have 50% and the lab assignment grade is 40/40, they get only 20 points. The instructor will review the paper summarizing and validate the percentage of each team member. If you have problems with team members, please let me know.

Problem

In this lab assignment you will build a Tic-Tac-Toe app, and create the GUI and handle the events programmatically. We leave the automatically generated activity_main.xml file as it is, and do everything by code.

Design

Model

The Model is comprised of the `TicTacToe` class, which encapsulates the functionality of the TicTacToe game. The `TicTacToe` constructor calls `reset-Game` after instantiating the two-dimensional array `game` as a 3 × 3 array. Having a separate `resetGame` method enables us to reuse the same object when playing back-to-back games. `ResetGame` clears the board by setting all the cells in `game` to 0; it also sets `turn` to 1 so that player 1 starts.

The method `play` first checks if the play is legal. If it is not, it returns 0. If it is, it updates `game` and `turn`. It then returns the value of `currentTurn` , which holds the old value of `turn`.

The method `whoWon` checks if a player has won the game and returns the player number if that is true; otherwise, it returns 0. We break down that logic using three `protected` methods: `checkRows` , `checkColumns`, and `checkDiagonals`. There is no reason for a client of that class to access these methods so we declare them `protected`.

API for the TicTacToe class

Instance Variables

private int[ ][ ] game 3 × 3 two-dimensional array representing the board

private int turn 1 = player 1's turn to play; 2 = player 2's turn to play

Constructor

TicTacToe( ) Constructs a TicTacToe object; clears the board and prepares to play by setting turn to 1

Public Methods

int play(int row, int column) If the play is legal and the board cell is not taken, plays and returns the old value of turn

boolean canStillPlay( ) Returns true if there is still at least one cell not taken on the board; otherwise, returns false

int whoWon( ) Returns i if player i won, 0 if nobody has won yet

boolean gameOver( ) Returns true if the game is over, false otherwise

void resetGame( ) Clears the board and sets turn to 1

```
public class TicTacToe {
    public static final int SIDE = 3;
    private int turn;
    private int [][] game;

    public TicTacToe( ) {
        game = new int[SIDE][SIDE];
        resetGame( );
    }

    public int play( int row, int col ) {
        int currentTurn = turn;
        if( row >= 0 && col >= 0 && row < SIDE && col < SIDE
                && game[row][col] == 0 ) {
            game[row][col] = turn;
            if( turn == 1 )
                turn = 2;
            else
                turn = 1;
```

```
            return currentTurn;
        }
        else
            return 0;
    }

    public int whoWon( ) {
        int rows = checkRows( );
        if ( rows > 0 )
            return rows;
        int columns = checkColumns( );
        if( columns > 0 )
            return columns;
        int diagonals = checkDiagonals( );
        if( diagonals > 0 )
            return diagonals;
        return 0;
    }

    protected int checkRows( ) {
        for( int row = 0; row < SIDE; row++ )
            if ( game[row][0] != 0 && game[row][0] == game[row][1]
                    && game[row][1] == game[row][2] )
                return game[row][0];
        return 0;
    }

    protected int checkColumns( ) {
        for( int col = 0; col < SIDE; col++ )
            if ( game[0][col] != 0 && game[0][col] == game[1][col]
                    && game[1][col] == game[2][col] )
                return game[0][col];
        return 0;
    }

    protected int checkDiagonals( ) {
        if ( game[0][0] != 0 && game[0][0] == game[1][1]
                && game[1][1] == game[2][2] )
            return game[0][0];
        if ( game[0][2] != 0 && game[0][2] == game[1][1]
                && game[1][1] == game[2][0] )
            return game[2][0];
        return 0;
    }

    public boolean canNotPlay( ) {
        boolean result = true;
        for (int row = 0; row < SIDE; row++)
            for( int col = 0; col < SIDE; col++ )
                if ( game[row][col] == 0 )
                    result = false;
```

```
        return result;
    }

    public boolean isGameOver( ) {
        return canNotPlay( ) || ( whoWon( ) > 0 );
    }

    public void resetGame( ) {
        for (int row = 0; row < SIDE; row++)
            for( int col = 0; col < SIDE; col++ )
                game[row][col] = 0;
        turn = 1;
    }
}
```

Part I – Version 1

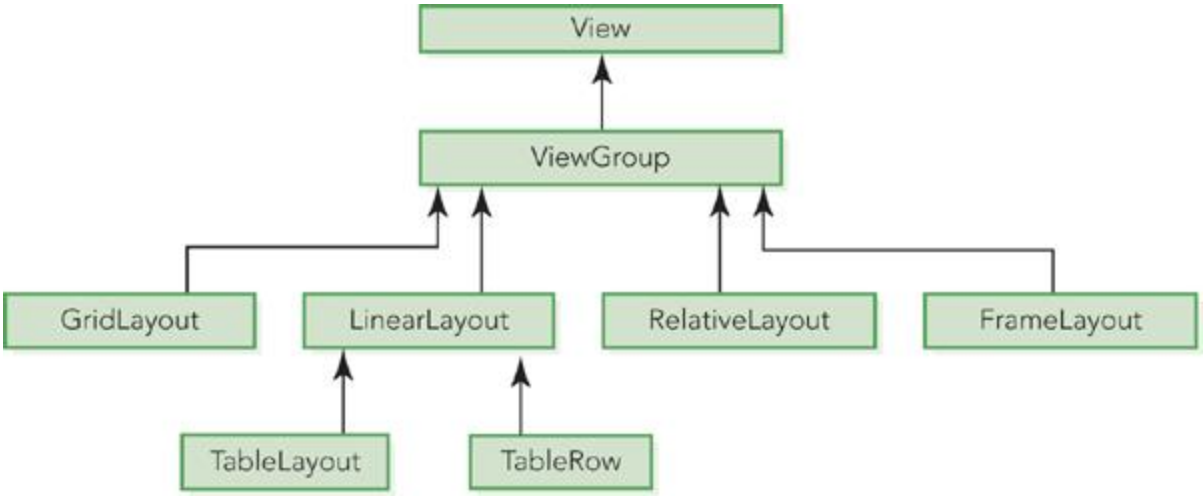
Creating the UI Programmatically

In order to display the nine buttons in a 3 × 3 grid, we use a GridLayout. The layout classes are subclasses of the abstract class ViewGroup, itself a subclass of View. A ViewGroup is a special View that can contain other Views, called its children.

`ViewGroup` and selected subclasses

Class	Description
ViewGroup	A View that contains other Views.
LinearLayout	A layout that arranges its children in a single direction (horizontally or vertically).
GridLayout	A layout that places its children in a rectangular grid.
FrameLayout	A layout designed to block out an area of the screen to display a single item.
RelativeLayout	A layout where the positions of the GUI components can be described in relation to each other or to their parent.
TableLayout	A layout that arranges its children in rows and columns.

Class	Description
TableRow	A layout that arranges its children horizontally. A TableRow should always be used as a child of a TableLayout.



There are many types of Android phones and tablets, and they all can have different screen dimensions. Thus, it is bad practice to hardcode widget dimensions and coordinates because a user interface could look good on some Android devices but poorly on others. In order to keep this example simple, we will assume that the user will only play in vertical orientation; thus, we assume that the width of the device is smaller than its height.

Resources involved in retrieving the width of the screen

Class or Interface	Package	Method and fields
Activity	android.app	WindowManager getWindowManager( )
WindowManager	android.view	Display getDefaultDisplay( )
Display	android.view	void getSize(Point)
Point	android.graphics	x and y public instance variables

We can dynamically retrieve, by code, the width and height of the screen of the device that the app is currently running on. Using that information, we can then size the GUI components so that they fit

within the device's screen no matter what the brand and model of the device are. In setting up the View, we perform the following steps:

- Retrieve the width of the screen
- Define and instantiate a GridLayout with three rows and three columns
- Instantiate the 3 × 3 array of Buttons
- Add the nine Buttons to the layout
- Set the GridLayout as the layout manager of the view managed by this activity

We chain three method calls, successively calling getWindowManager, getDefaultDisplay, and getSize. GetWindowManager, from the Activity class, returns a WindowManager object that encapsulates the current window. With it, we call the method getDefaultDisplay of the WindowManager interface; it returns a Display object, which encapsulates the current display and can provide information about its size and its density. With it, we call the getSize method of the Display class; getSize is a void method but takes a Point object reference as a parameter. When that method executes, it modifies the Point parameter object and assigns to it the width and height of the display as its x and y instance variables. We retrieve the width of the screen (size.x) and assign one third of it to the variable w.

GridLayout constructor and methods

Constructor

GridLayout(Context context) Constructs a GridLayout within the app environment defined by context.

Public Methods

setRowCount(int rows)Sets the number of rows in the grid to rows.

setColumnCount(int cols) Sets the number of columns in the grid to cols.

addView(View child, int w, int h) Method inherited from ViewGroup; adds child to this ViewGroup using width w and height h.

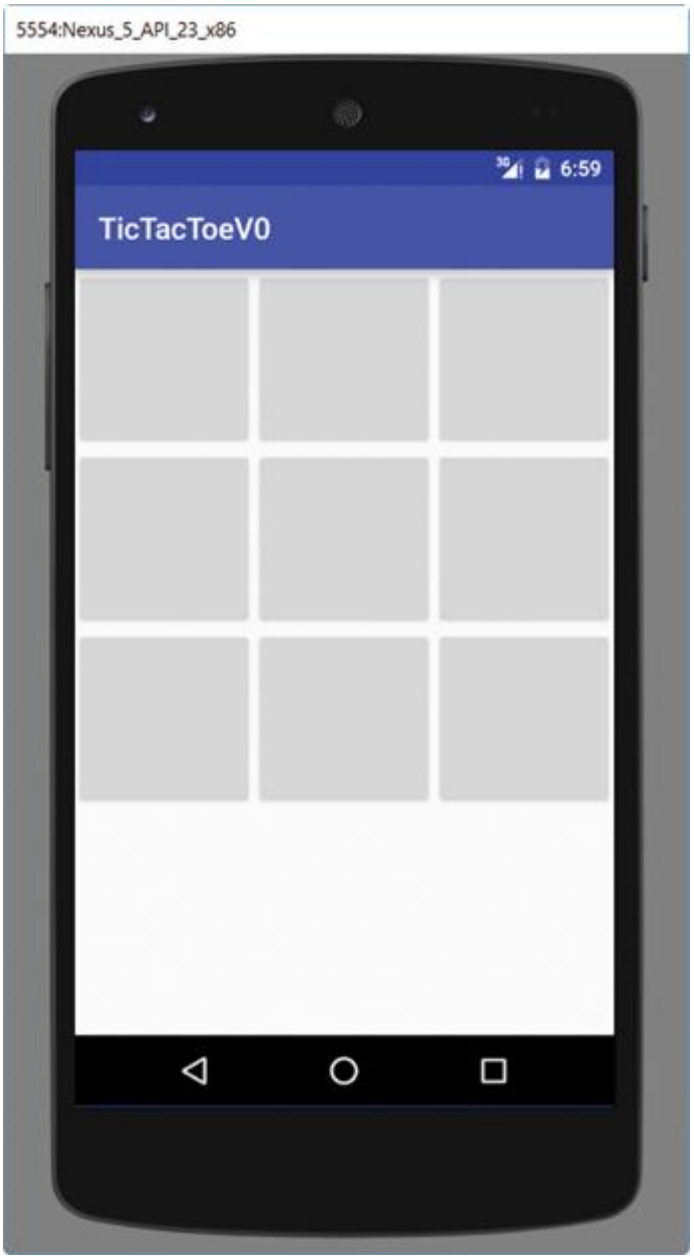
`MainActivity class`

```
public class MainActivity extends AppCompatActivity {
    private Button [][] buttons;

    @Override
    protected void onCreate( Bundle savedInstanceState ) {
        super.onCreate( savedInstanceState );
        // setContentView( R.layout.activity_main );
        buildGuiByCode( );
    }

    public void buildGuiByCode( ) {
        // Get width of the screen
        Point size = new Point( );
        //YOUR CODE – Retrieve the width of the screen
        _________________________
        //YOUR CODE – Assign one third of the width of the screen to a variable w
        ________________________
```

```
        // Create the layout manager as a GridLayout
        GridLayout gridLayout = new GridLayout( this );
        gridLayout.setColumnCount( _________________ );
        gridLayout.setRowCount( _________________ );

        // Create the buttons and add them to gridLayout
        buttons = new Button[TicTacToe.SIDE][TicTacToe.SIDE];
        for( int row = 0; row < TicTacToe.SIDE; row++ ) {
            for( int col = 0; col < TicTacToe.SIDE; col++ ) {
                buttons[row][col] = new _________________(this );
                gridLayout._____________________________;
            }
        }

        // Set gridLayout as the View of this Activity
        setContentView( gridLayout );
    }
}
```

In order to keep things simple, we only allow the app to work in vertical orientation. Inside the AndroidManifest.xml file, we assign the value `portrait` to the `android:screenOrientation` attribute of the `activity` tag.

Run the app inside the emulator.



Part II Event Handling – Version 2

we add code to capture a click on any button, identify what button was clicked, and we place an X inside the button that was clicked.

In order to capture and process an event, we need to:

1. Write an event handler (a class extending a listener interface)
2. Instantiate an object of that class
3. Register that object listener on one or more GUI components

The type of event that we want to capture determines the listener interface that we implement. View.OnClickListener is the listener interface that we should implement in order to capture and handle a click event on a View. Since it is defined inside the View class, importing the View class automatically imports View.OnClickListener.

View.OnClickListener interface

public abstract void onClick(View v) Called when a View has been clicked; the parameter v is a reference to that View.

Our event handler class, ButtonHandler, implements OnClickListener and overrides onClick. ButtonHandler, implemented as a private class. we add an output statement that gives feedback on the View parameter of the onClick method; we expect that it is a Button. We loop through the array buttons in order to identify the row and column values of the button that was clicked. We then call the update method, passing these row and column values.

The update method outputs some feedback on the row and column of the button that was clicked (for debugging purposes), and writes an X inside it.

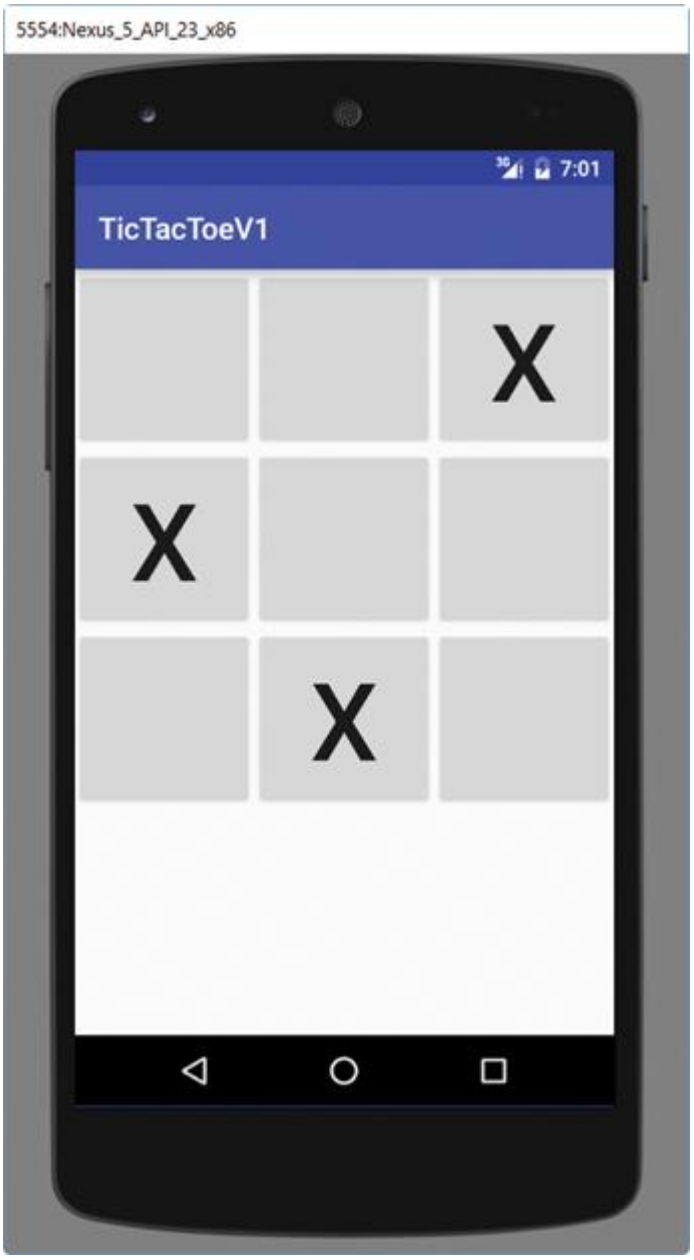
We declare and instantiate a ButtonHandler object and we register it on each element of buttons as we loop through the array. Additionally, we set the text size of each button so that it is relative to the size of each button, which we have sized relatively to the screen. In this way, we try to make our app as much device independent as possible. It is important to test an app on as many devices as possible and of various sizes to validate sizes and font sizes.

Modify the class MainActivity

```
public class MainActivity extends AppCompatActivity {
    private Button[][] buttons;

            // Create the buttons and add them to gridLayout
        buttons = new Button[TicTacToe.SIDE][TicTacToe.SIDE];
        //Instantiate a ButtonHandler object
          ____________________________________
        for( int row = 0; row < TicTacToe.SIDE; row++ ) {
            for( int col = 0; col < TicTacToe.SIDE; col++ ) {
                buttons[row][col] = new __________( this );
                //Set the textsize for each button to w * 0.2
                ____________________________________
                //Register the event for each button
                ____________________________________
                gridLayout._____________________________;
            }
        }

        // Set gridLayout as the View of this Activity
        setContentView(gridLayout);
    }

    public void update(int row, int col) {
        Log.w( "MainActivity", "Inside update: " + row + ", " + col );
        buttons[row][col].____________________;
    }
```

```
    //Implement the ButtonHandler event
    private class ButtonHandler implements _____________________ {
        public void onClick( _______________ ) {
            Log.w( "MainActivity", "Inside onClick, v = " + v );
            for( int row = 0; row < TicTacToe.SIDE; row++ )
                for( int column = 0; column < TicTacToe.SIDE; column++ )
                    if ( v == _________________ )
                        ___________________________
        }
    }
}
```

Run the app inside the emulator. The figure below shows the app after the user clicked successively on three buttons.



Integrating the Model to Enable Game Play: TicTacToe

We are assuming that two users will be playing on the same device against each other. Enabling game play does not just mean placing an X or an O on the grid of buttons at each turn. It also means enforcing the rules, such as not allowing someone to play twice at the same position on the grid, checking if one player won, indicating if the game is over. Our Model, the `TicTacToe` class, provides that functionality. In order to enable play, we add a `TicTacToe` object as an instance variable of our `Activity` class, and we call the methods of the `TicTacToe` class as and when needed. Play is happening inside the `update` method so we have to modify it. We also need to check if the game is over and, in that case, disable all the buttons.

We declare a TicTacToe object, tttGame. Inside the update method, we first call the play method of the TicTacToe class with tttGame . We know that it returns the player number (1 or 2) if the play is legal, 0 if the play is not legal. If the play is legal and the player is player 1, we write X inside the button. If the play is legal and the player is player 2, we write O inside the button. If the play is not legal, we do not do anything.

When the game is over, we want to disable all the buttons. The enableButtons method enables all the buttons if its parameter, enabled, is true. If it is false, it disables all the buttons. We can enable or disable a Button by calling the setEnabled method of the Button class using the argument true to enable the Button, false to disable it. We test if the game is over and disable all the buttons if it is

```
public class MainActivity extends AppCompatActivity {
    _____________________________________
    private Button [][] buttons;

    @Override
    protected void onCreate( Bundle savedInstanceState ) {
        super.onCreate( savedInstanceState );
         ________________________
         buildGuiByCode( );
    }

    public void buildGuiByCode( ) {
    }

    public void update( int row, int col ) {
        //YOUR CODE
    }

    public void enableButtons( boolean enabled ) {
        for( int row = 0; row < TicTacToe.SIDE; row++ )
            for( int col = 0; col < TicTacToe.SIDE; col++ )
                buttons[row][col].setEnabled( enabled );
    }

    private class ButtonHandler implements View.OnClickListener {
        public void onClick( View v ) {
            for( int row = 0; row < TicTacToe.SIDE; row ++ )
                for( int column = 0; column < TicTacToe.SIDE; column++ )
                    if( v == _____________________________ )
                                ____________________________________
        }
    }
```

Run the app inside the emulator. The figure below shows player 1 just won and the buttons are disabled.

Part III – Version 3

Layout parameters

In Version 3, we use a GridLayout with four rows and three columns. We place the buttons in the first three rows, and use the fourth row to place a TextView across the three columns to display the game status.

The ViewGroup.LayoutParams class enables us to specify how a View child is to be positioned inside its parent layout. GridLayout.LayoutParams inherits from the ViewGroup. LayoutParams

class and we can use it to set the layout parameters of a GUI component before we add it to a GridLayout.

A GridLayout.LayoutParams object is defined by two components, rowSpec and columnSpec, both of type GridLayout.Spec. Spec is defined as a public static inner class of GridLayout. Thus, we refer to it using GridLayout.Spec. RowSpec defines a vertical span, starting at a row index and specifying a number of rows; columnSpec defines a horizontal span, starting at a column index and specifying a number of columns. Thus, by defining rowSpec, the vertical span, and columnSpec, the horizontal span, we define a rectangular area within the grid where we can place a View.

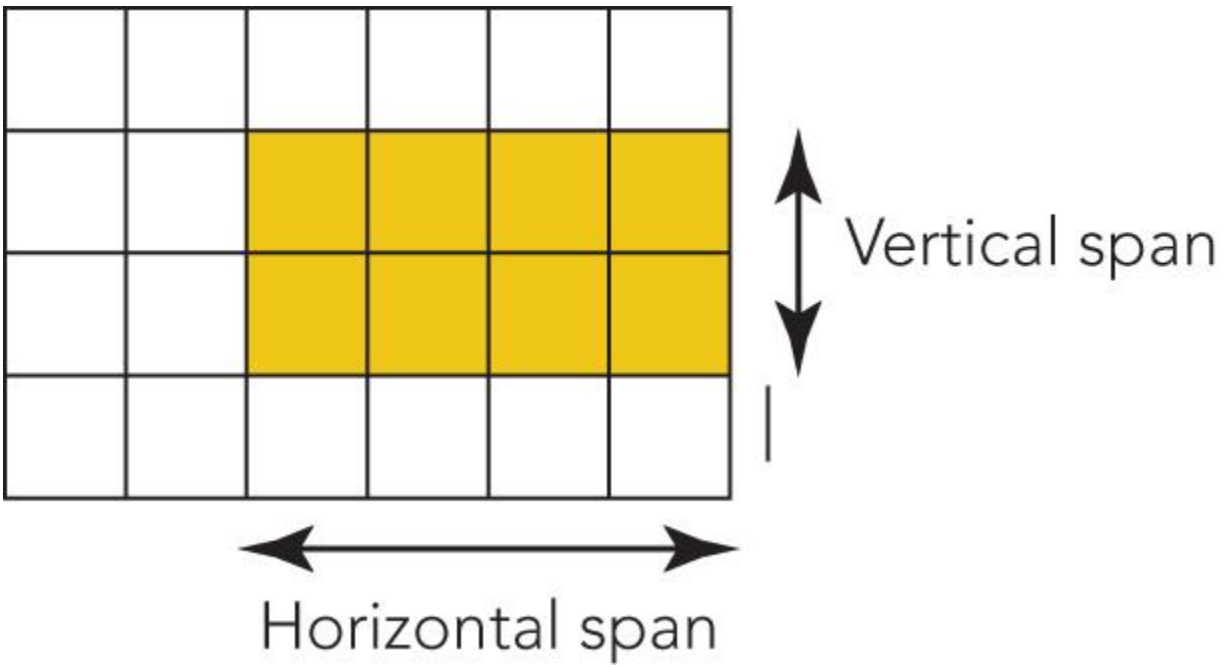
We can use the static spec methods of the GridLayout.Spec class to create and define a GridLayout.Spec object, which defines a span. Once we have defined two GridLayout.Spec objects, we can use them to define a GridLayout.LayoutParams object.

Methods of the GridLayout and GridLayout.LayoutParams classes

public static GridLayout.Spec spec(int start, int size)
Returns a GridLayout.Spec object where start is the starting index and size is the size.
public static GridLayout.Spec spec(int start, int size, GridLayout.Alignment alignment)
Returns a GridLayout.Spec object where start is the starting index and size is the size. Alignment is the alignment; common values include GridLayout.TOP,ridLayout,BOTTOM, GridLayout.LEFT, GridLayout.RIGHT, GridLayout.CENTER.

GridLayout.LayoutParams(Spec rowSpec, Spec columnSpec)
Constructs a LayoutParams object with rowSpec and columnSpec.

The figure below shows a 4 × 6 grid of cells.



An area of a 4 × 6 grid defined by a vertical span with start = 1 and size = 2, and a horizontal span with start = 2 and size = 4

The shaded area can be defined as follows:

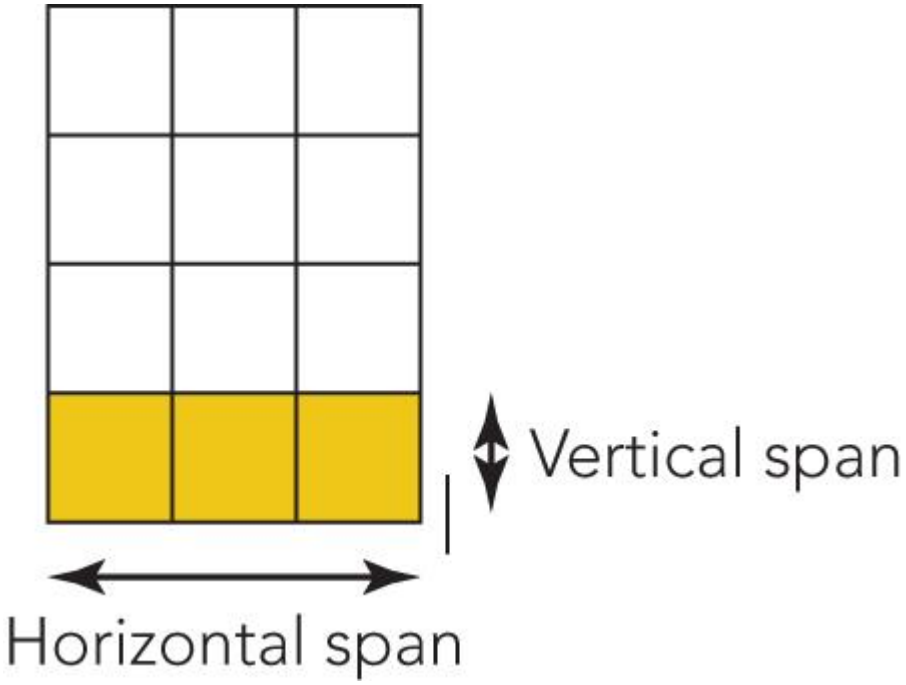
The vertical span starts at index 1 and has size 2
The horizontal span starts at index 2 and has size 4
We could define a GridLayout.LayoutParams object for the shaded area as follows:

```
// vertical span
GridLayout.Spec rowSpec = GridLayout.spec( 1, 2 );
// horizontal span
GridLayout.Spec columnSpec = GridLayout.spec( 2, 4 );
GridLayout.LayoutParams lp
= new GridLayout.LayoutParams( rowSpec, columnSpec );
```

Note that we could use Spec instead of GridLayout.Spec if we include the following import statement:

```
import android.widget.GridLayout.Spec;
```
in addition to:
```
import android.widget.GridLayout;
```

The figure below shows a 4 × 3 grid of cells like the one in our app.



An area of a 4 × 3 grid defined by a vertical span with start = 3 and size = 1, and a horizontal span with start = 0 and size = 3

The shaded area can be defined as follows:

The vertical span starts at index 3 and has size 1
The horizontal span starts at index 0 and has size 3

We define a GridLayout.LayoutParams object for the shaded area as follows:

GridLayout.Spec rowSpec = GridLayout.spec(3, 1);
GridLayout.Spec columnSpec = GridLayout.spec(0, 3);
GridLayout.LayoutParams lp
= new GridLayout.LayoutParams(rowSpec, columnSpec);

MainActivity class update

- Modify the method buildGuiCode

```
GridLayout gridLayout = new GridLayout( this );
gridLayout.setColumnCount( TicTacToe.SIDE );
gridLayout.setRowCount( TicTacToe.SIDE + __________ );

…….....
```

```
// set up layout parameters of 4th row of gridLayout
status = new TextView( this );
GridLayout.Spec rowSpec = GridLayout.spec( TicTacToe.SIDE, _____ );
GridLayout.Spec columnSpec = GridLayout.spec( _____, TicTacToe.SIDE );
GridLayout.LayoutParams lpStatus
        = new GridLayout.LayoutParams( ____, _______ );
status.setLayoutParams( ________ );

// set up status' characteristics
status.setWidth( TicTacToe.SIDE * _________ );
status.setHeight( ______ );
status.setGravity( Gravity.CENTER );
status.setBackgroundColor( Color.GREEN );
status.setTextSize( ( int ) ( w * .15 ) );
status.setText( ____________________ );

gridLayout.addView( ______ );
```

- **Modify the method update**

```
if( tttGame.isGameOver( ) ) // game over, disable buttons
{
    status.setBackgroundColor(Color.RED);
    enableButtons(_____________);
    status.setText(_________________________);

}
```

The figure below shows the app running inside the emulator, including the status of the game at the bottom of the screen.

Part IV – Version 4

In Version 4, we enable the player to play another game after the current one is over. When the game is over, we want a dialog box asking the user if he or she wants to play again to pop up. If the answer is yes, he or she can play again. If the answer is no, we exit the activity (in this case the app since there is only one activity).

The AlertDialog.Builder class, part of the android.app package, provides the functionality of a pop-up dialog box. It offers several choices to the user and captures the user's answer. A dialog box of type AlertDialog.Builder can contain up to three buttons: negative, neutral, and positive buttons. In this app,

we only use two of them: the positive and negative buttons. Typically, these two buttons correspond to yes or no answers from the user, although that is not required.

Selected constructor and methods of the AlertDialog.Builder class

Constructor of the AlertDialog.Builder class

public AlertDialog.Builder(Context context)

Constructs an AlertDialog.Builder dialog box object for the context parameter.

Methods of the AlertDialog.Builder class

public AlertDialog.Builder setMessage(CharSequence message)
Sets the message to display to message; returns this AlertDialog.Builder object, which allows chaining if desired.

public AlertDialog.Builder setTitle(CharSequence title)
Sets the title of the alert box to title; returns this AlertDialog.Builder object, which allows method call chaining if desired.

public AlertDialog.Builder setPositiveButton(CharSequence text, DialogInterface.OnClickListener listener)
Sets listener to be invoked when the user clicks on the positive button; sets the text of the positive button to text.

public AlertDialog.Builder setNegativeButton(CharSequence text, DialogInterface.OnClickListener listener)
Sets listener to be invoked when the user clicks on the negative button; sets the text of the negative button to text.

public AlertDialog.Builder setNeutralButton(CharSequence text, DialogInterface.OnClickListener listener)
Sets listener to be invoked when the user clicks on the neutral button; sets the text of the neutral button to text.

public AlertDialog show( )
Creates, returns an AlertDialog box and shows it.

Modify the MainActivity class

- **Modify the method update**

```
showNewGameDialog( );   // offer to play again
```

- **Define the method resetButtons**

```
public void resetButtons( ) {
    for( int row = 0; row < TicTacToe.SIDE; row++ )
```

```
        for( int col = 0; col < TicTacToe.SIDE; col++ )
            ____________________________________________
}
```

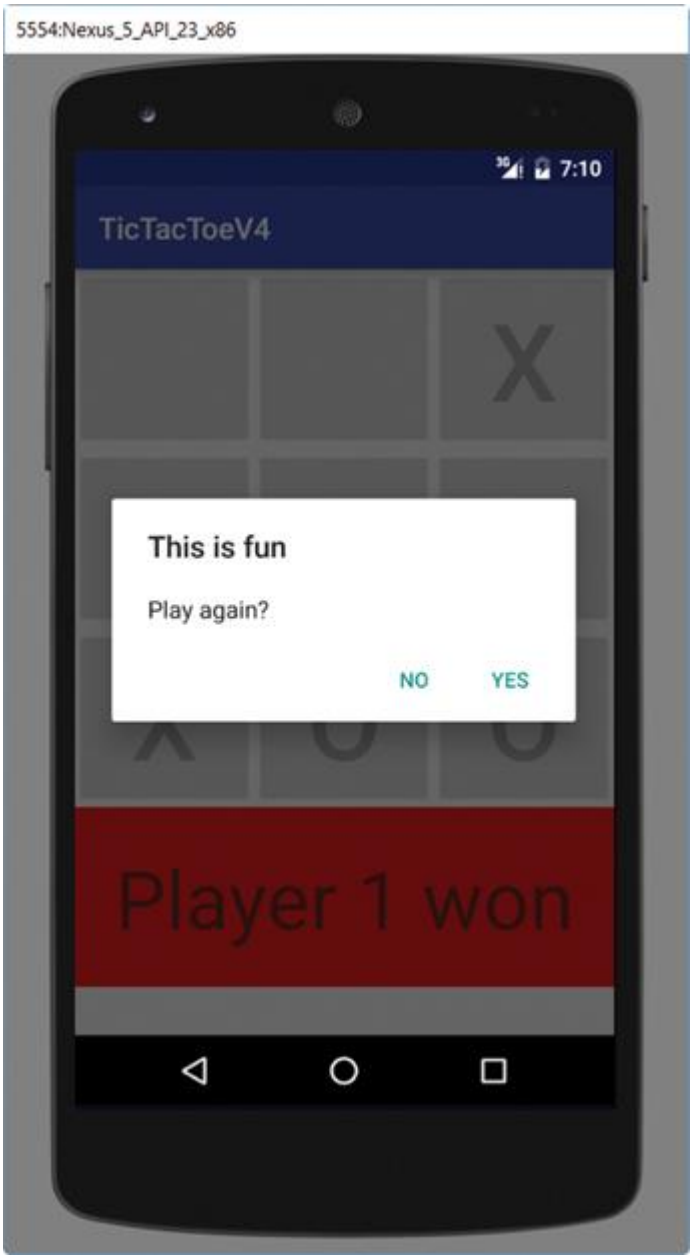
- **Define the method showNewGameDialog**

```
public void showNewGameDialog( ) {
    AlertDialog.Builder alert = new AlertDialog.Builder( _______ );
    alert.setTitle( "This is fun" );
    alert.setMessage( "Play again?" );
    PlayDialog playAgain = new PlayDialog( );
    alert.setPositiveButton( ______,_____ );
    alert.setNegativeButton( _____,_______ );
    _________
}
```

- **Define the class PlayDialog**

```
private class ButtonHandler implements _________________________________ {
    public void ____________________________ {
        Log.w( "MainActivity", "Inside onClick, v = " + v );
        for( int row = 0; row < TicTacToe.SIDE; row++ )
            for( int column = 0; column < TicTacToe.SIDE; column++ )
                if ( v == ____________________________ )
                    ____________________________
    }
}
```

The figure below shows the app at the end of the game with the status reflecting that player 1 won, and asking the user to play again.



Grading:

1. Submit the project in zip file (extension zip)
2. A pdf document that contains the content of MainActivity class.
3. A zoom link or YouTube that demonstrate the user of the TicTacToe app. Show player 1 as a winner (less than 3 minutes). Write the link on the comment section in the Dropbox.
4. Write full names of the team members
5. Write down the contribution of each team members. All the team members must agree to the contribution of each team member.

Note:
App is running correctly and met all the requirements is 20 points.
Incomplete item 1(-20 points)
Incomplete item 2 (-10 points)
Incomplete item 3 (-10 points)

Part V – Version 5

Splitting the View and the Controller

In the next version, we split the View and the Controller. In this way, we make the View reusable. The Controller is the middleman between the View and the Model, so we keep the View independent from the Model.

In the View, in addition to the code creating the View, we also provide methods to:

- Update the View.
- Get user input from the View.

This is similar to the Model class, which provides methods to retrieve its state and update it.

In the Controller, in addition to an instance variable from the Model, we add an instance variable from the View. With it, we can call the various methods of the View to update it and get user input from it.

The array of buttons and the TextView status are now inside the View. Updating the View means updating the buttons and the TextView status. To that end, we provide the following methods:

- A method to set the text of a particular button
- A method to set the text of the TextView status
- A method to set the background color of the TextView status
- A method to reset the text of all the buttons to the empty String (we need it when we start a new game)
- A method to enable or disable all the buttons (we also need it when we start a new game)

User input for this View is clicking on one of the buttons. Typically, event handling is performed in the Controller. Thus, we provide a method to check if a particular button was clicked.

The ButtonGridAndTextView class, the View for this version of this app. In the previous version, our View was a GridLayout, thus, the ButtonGridAndTextView class extends GridLayout, and therefore, it "is" a GridLayout. We have three instance variables:

- buttons, a two-dimensional array of Buttons
- status, a TextView
- side, an int, the number of rows and columns in buttons

Because we want to keep the View independent from the model, we do not use the SIDE constant of the TicTacToe class to determine the number of rows and columns in buttons. Instead, the side instance variable stores that value. The constructor includes a parameter, newSide, that is assigned to side . When we create the View from the Controller, we have access to the Model, thus, we will pass the SIDE constant of the TicTacToe class so that it is assigned to side.

The ButtonGridAndTextView constructor includes three more parameters: a Context, an int, and a View.OnClickListener. The Context parameter is needed to instantiate the widgets of the View (the Buttons and the TextView). Since the Activity class inherits from Context, an Activity "is a" Context. Thus, when we create the ButtonGridAndTextView from the Controller, we can pass this for the Context parameter. We pass that Context parameter to the Button and TextView constructors. The int parameter represents the width of the View. By having the width as a parameter, we let the Activity client determine the dimensions of the View. We assign the newSide parameter to side . Finally, the View.OnClickListener parameter enables us to set up event handling. We want to handle events in the Controller but the Buttons are in the View, so event handling needs to be set up in the View. The constructor code can be made more robust by testing if newSide and width are positive. This is left as an exercise.

We add each Button and the TextView to this ButtonGridAndTextView.
The setStatusText, setStatusBackgroundColor, setButtonText, resetButtons, and enableButtons methods provide the ability to a client of the View (the Controller) to update the View. The isButton method enables a client of the View (the Controller) to compare a Button with a Button from the array buttons identified by its row and column. From the Controller, we will call that method to identify the row and the column of the Button that was clicked.

ButtonGridAndTextView class

```
public class ButtonGridAndTextView extends GridLayout {
    private int side;
    private Button [][] buttons;
    private TextView status;

    public ButtonGridAndTextView( Context context, int width, int newSide,
                                  OnClickListener listener ) {
        super( context );
        side = newSide;
        // Set # of rows and columns of this GridLayout
        setColumnCount( side );
        setRowCount( side + 1 );

        // Create the buttons and add them to this GridLayout
       //YOUR CODE



        // set up layout parameters of 4th row of gridLayout
        //YOUR CODE
```

```
    public void setStatusText( String text ) {
        status.setText( text );
    }

    public void setStatusBackgroundColor( int color ) {
        status.setBackgroundColor( color );
    }

    public void setButtonText( int row, int column, String text ) {
        buttons[row][column].setText( text );
    }

    public boolean isButton( Button b, int row, int column ) {
        return ( b == buttons[row][column] );
    }

    public void resetButtons( ) {
        for( int row = 0; row < side; row++ )
            for( int col = 0; col < side; col++ )
                buttons[row][col].setText( "" );
    }

    public void enableButtons( boolean enabled ) {
        for( int row = 0; row < side; row++ )
            for( int col = 0; col < side; col++ )
                buttons[row][col].setEnabled( enabled );
    }
}
```

MainActivity class

A ButtonGridAndTextView instance variable, tttView, is declared and instantiated. Inside onClick , we first identify which button was clicked , and call setButtonText to update the View depending on whose turn it is to play. If the game is over, we update the View accordingly by calling the setStatusBackgroundColor and setStatusText methods. We also disable the buttons.

```
public class MainActivity extends AppCompatActivity {
    private TicTacToe game;
    private ____________________ tttView;

    @Override
    protected void onCreate( Bundle savedInstanceState ) {
         ……..
        ButtonHandler bh = new ButtonHandler( );
        tttView = ____________________________________________________
        tttView._____________________( ____________________ );
        setContentView( ____________________ );
    }
```

```
    private class ButtonHandler implements View.OnClickListener {
        public void onClick( View v ) {
            for( int row = 0; row < TicTacToe.SIDE; row++ ) {
                for( int column = 0; column < TicTacToe.SIDE; column++ ) {
                    if( tttView.isButton( ____________________________________ ) {
                        int play = game.play( row, column );
                        if( play == 1 )
                            tttView.___________________________________;
                        else if( ___________________ )
                            tttView._____________________________________;
                        if( game._________________ ) {
                            tttView.setStatusBackgroundColor_____________ );
                            tttView.___________________________;
                            tttView.setStatusText( _____________________________ );
                            showNewGameDialog( );    // offer to play again
                        }
                    }
                }
            }
        }
    }

}
```

Grading:

1. Submit the project in zip file (extension zip)
2. A pdf document that contains the content of MainActivity class, and ButtonGridAndTextView class.
3. A zoom link or YouTube that demonstrate the user of the TicTacToe app. Show player 2 as a winner (less than 3 minutes). Write the link on the comment section in the Dropbox.
4. Write full names of the team members
5. Write down the contribution of each team members. All the team members must agree to the contribution of each team member.

Note:
App is running correctly and met all the requirements is 20 points.
Incomplete item 1(-20 points)
Incomplete item 2 (-10 points)
Incomplete item 3 (-10 points)