ENGINEERING TRIPOS PART IIA

DPO (CLUSTER 3) MODULE EXPERIMENT 3F6

OBJECT ORIENTED PROGRAMMING: ANDROID APP DEVELOPMENT

You should read this handout and understand the background material before you arrive for the laboratory session. Otherwise, you will find it very difficult to complete the laboratory in the allotted time.

1 Introduction

The continued rise in the popularity, power and range of technologies built into mobile devices presents a wealth of entrepreneurial opportunities for the budding software engineer. In this lab, we will cover the basics of application ("app") development for the Android operating system. This will involve learning about the principles of object oriented programming and the Java language, since this forms the foundation of an Android app. As a case-study, we consider developing a simple Sudoku puzzle solver.

Aims and objectives

- To introduce object-oriented programming and to gain experience in object-oriented programming (in Java) for a real-world application
- To introduce Android mobile phone application development including graphical user interfaces (GUIs) and interfacing with Java
- To teach good programming practice
- To compile and execute programs from the command line in a Unix environment

2 Background

This section lays out the theory required for the lab. We will begin with a brief introduction to Sudoku puzzles: the App we ultimately produce will solve such puzzles entered by a user. Consideration of the design of a Sudoku puzzle solver will be used to motivate an object oriented approach. We will then cover the basics of object oriented programming in Java. Finally we will discuss the organisation of an Android mobile phone application, leaving the detailed structure to be explained as we progress through the lab.

2.1 Sudoku as a running example

Sudoku is a logic puzzle consisting of a nine-by-nine grid (see figure 1). The goal is to fill each row, column, and sub-square with the digits 1 to 9. The puzzle is partially filled in to begin with in such a way that it has a unique solution. One way of solving the puzzle is to maintain a list of candidates for each element and to eliminate candidates using the constraints. For example, consider the element labelled in figure 1A. The following candidates can be eliminated by simple application of the row, column and sub-square constraints: $[1, 3, 6, 8, 9]$. When one candidate remains for an element, the value can be filled in, and this imposes new constraints on the other elements, which may elimate further candidates.

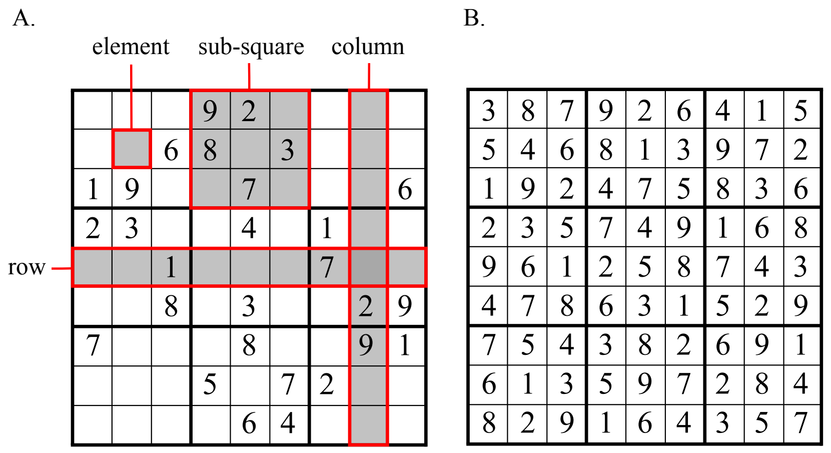


Figure 1: Sudoku terminology. A) A Sudoku puzzle consists of a grid of *elements*. The goal is to fill each element with an integer from the set $\{1, 2, \ldots, 9\}$ in such a way that the constraints are satisfied. The constraints are that there can be no repeated values in the elements belonging to each *row*, *column*, and *sub-square*. A number of values are given at the start of the problem. For newspaper Sudokus, the given values are usually arranged in a symmetric way (here rotationally symmetric). B) The solution to the 'very hard' Sudoku shown in A satisfies the row, column and sub-square constraints.

2.2 Motivating object-oriented design

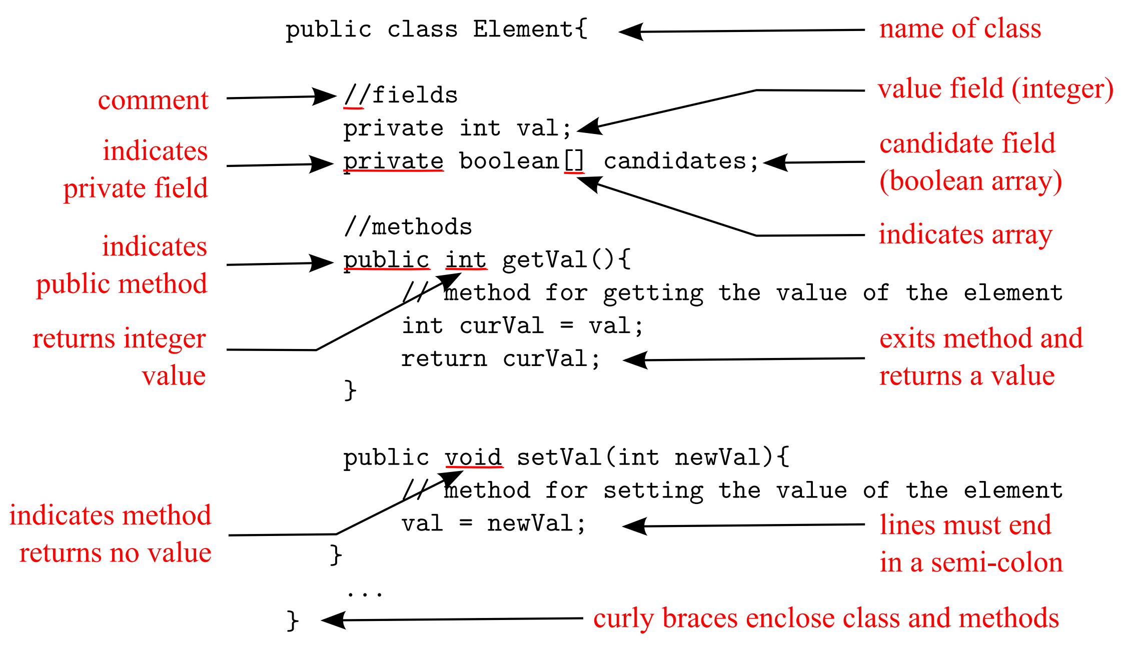
We could dive right in and start coding up a Sudoku solver. But it's always good when programming to spend some time planning the approach first. One of the first considerations is how to structure the data.

Following the discussion above, a sensible description of a Sudoku puzzle might be as a nine by nine grid in which each element contains two types of data: The first being the value (a value of 0 can be used to indicate that it is currently unassigned). The second being a representation of the possible values the element may take. This set of candidates could be a Boolean array with nine entries. If, for example, the fourth entry of the Boolean array is `true`, then that denotes that the value 4 has not been ruled out for that element. We could handle these two types of data separately, but in this tutorial we are bundling them together and send them as one composite data type. This idea of creating your own data structures to represent semantically-meaningful groups of data is good programming practice, and is fundamental to object-oriented programming.

The value of an element and the set of candidates are coupled variables: if there is only one candidate remaining in an element, the value should be equal to that candidate and *vice versa*. And if several candidates remain, the value should be unassigned and therefore set to zero. Because these two variables are yoked, it is potentially dangerous to permit arbitrary changes to the two fields. After all, a naïve user might accidentally alter one of the variables and forget to update the other. Instead, it would be safer and certainly simpler for a user, if the data were not directly accessible, but instead packaged with functions for accessing and altering the content that maintained the dependencies between the variables. The strategy of hiding the raw data from the user is called **encapsulation**. Providing a package containing the encapsulated data along with functions for accessing and altering them is called **data abstraction**. Encapsulation and data abstraction are two core motivations for object oriented programming.

2.3 Object-oriented programming: Basic Java

Let's take a look at how to define an Element in Java which incorporates the design considerations above. Java allows us to define the blueprint for an Element called a **class**. The general name for an instantiation of a class is an **object**. Classes contain **fields** (which are variables) and **methods** (which are functions), thereby separating the data from the behaviour. In our case, the Element class has two fields: the value the element takes (`val`) and the candidate values it might take (`candidates`). Both fields are set as `private` meaning that they can only be accessed by the methods which are specified within the Element class (so called member functions). That is, they are encapsulated. The class must therefore provide `public` methods, accessible outside of the class, for accessing the data (e.g. `getVal`) and modifying the data (e.g. `setVal`).

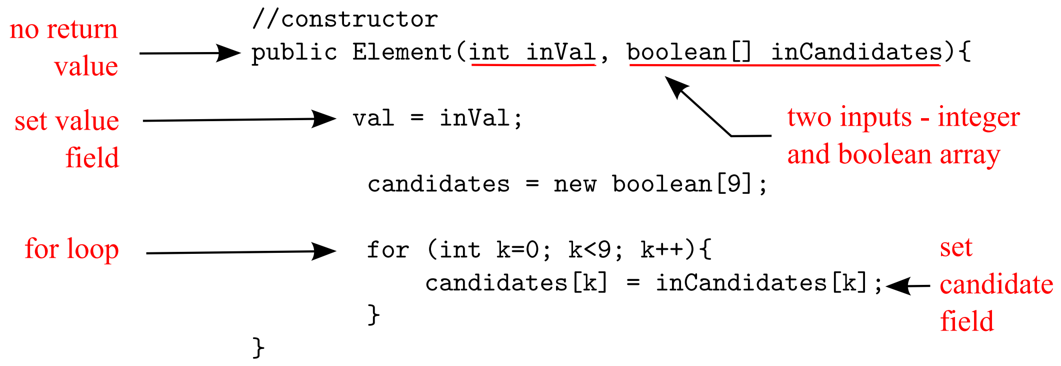


```
public class Element{

    //fields
    private int val;
    private boolean[] candidates;

    //methods
    public int getVal(){
        // method for getting the value of the element
        int curVal = val;
        return curVal;
    }

    public void setVal(int newVal){
        // method for setting the value of the element
        val = newVal;
    }
    ...
}
```

name of class
comment
value field (integer)
indicates private field
candidate field (boolean array)
indicates array
indicates public method
returns integer value
exits method and returns a value
indicates method returns no value
lines must end in a semi-colon
curly braces enclose class and methods

This organisation means that the methods can perform checks to make sure that the user-request is sensible, guaranteeing that the data is accessed correctly. The use of an explicit method also makes the data structure easier to use. So, it is usually wise to make member methods the *only* means of accessing the class's data, so that the class can guarantee that it is always in a correct state.

A class contains constructors that are invoked to create objects from the class blueprint. Constructor definitions look like method definitions — except that they use the name of the class and have no return type.



```
//constructor
public Element(int inVal, boolean[] inCandidates){

       val = inVal;

        candidates = new boolean[9];

        for (int k=0; k<9; k++){
            candidates[k] = inCandidates[k];
        }
}
```

To create a new Element we call the constructor using the `new` operator:

```
newElement = new Element(val,candidates);
```

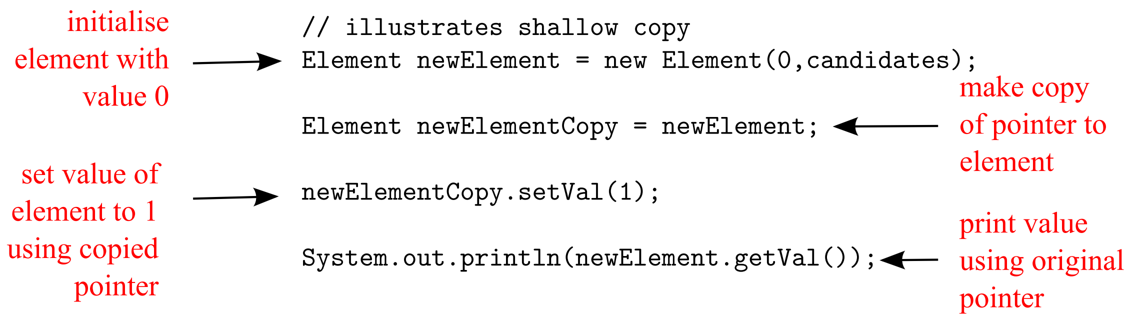
We now know how to specify classes in Java and how to create objects from a class. But how do we access an object's fields and methods? This is simple. Fields and methods of a particular object instance can be accessed by writing the object name followed by a dot (.) followed by the field or method name. For example, the method `getVal()` above accesses the `val` field in this way. Outside of a class, public fields and methods can be accessed using the dot (.) operator. For example, to call the method which gets the value of the `newElement` object above, we use:

```
int newVal = newElement.getVal();
```

If the method requires input arguments, we place these in the parentheses, e.g. to set the value field of the `newElement` object to nine, we use

```
newElement.setVal(9);
```

How do we copy objects? In Java, objects are kept in the background leaving the user to interact with pointers that are a reference to the memory space of the object (see figure 2). We can make a copy of the reference pointer to the object simply, like so:



```
// illustrates shallow copy
Element newElement = new Element(0,candidates);

Element newElementCopy = newElement;

newElementCopy.setVal(1);

System.out.println(newElement.getVal());
```

initialise element with value 0
make copy of pointer to element
set value of element to 1 using copied pointer
print value using original pointer

Is the output printed to the screen here 0 or 1? Since we have instantiated two pointers which both point to the same object, we may alter the value-field of the object using either pointer, and we may query that object using either pointer, and in each case we will get the same result. The output printed to screen is therefore 1. In order to make a copy of an object itself, rather than its reference, we need to use the constructor:

copy by getting values and using constructor

```
// illustrates deep copy
Element newElementCopy = new Element(newElement.getVal(),
                                     newElement.getCandidates());

newElementCopy.setVal(1);

System.out.println(newElement.getVal());
```

set value field of copy to 1

print value of original element

In this case the value of the original object will not have changed. Figure 2 demonstrates these examples schematically. In fact, we have already seen an example of 'copy by value'. If you cast your eye back to the Element constructor above, you will notice that we do not set the candidate field using `candidates = inCandidates`, but instead use a `for` loop in order to make a copy of the value of the variables.

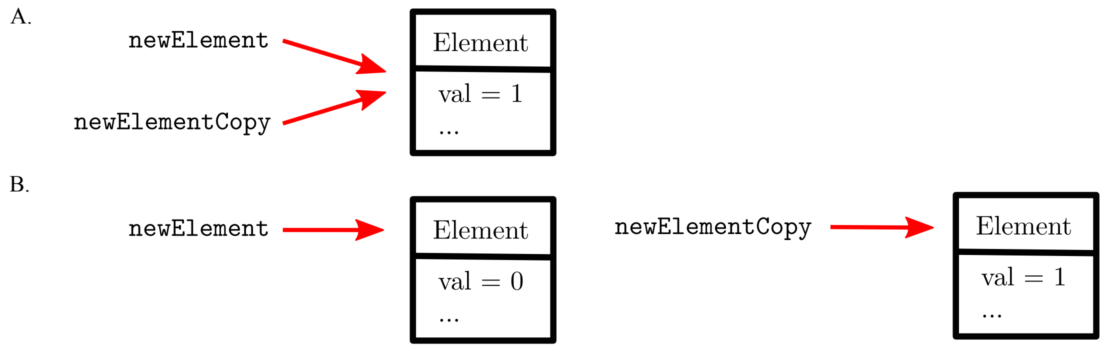


Figure 2: Diagrams of the state of the programmes at the end of the two pieces of code above. A) A copy of the reference has been made and both point to the same values. B) A copy of the values has been made and the references point to different values.

We have covered how to copy objects in Java, but it is worth bearing in mind that there are a number of primitive data types which are not objects: For example, `int` and `boolean` which we have met above. These primitive types are handled, just as we would expect, "by value" rather than "by reference". Consider the following example:

i contains the value 3

```
int i = 3;

int j = i;

i = 2;
```

j contains a copy of the value in i

i contains the value 2, j unchanged

Some object-oriented languages require that you keep track of all the objects you create and that you explicitly destroy them when they are no longer needed (object oriented C++ is one example). Managing memory explicitly is tedious and error-prone. Therefore the Java runtime environment deletes objects when it determines that they are no longer being used. This process is called garbage collection.

Java provides a library of many pre-defined classes. A complete reference of all these classes can be found here: http://docs.oracle.com/javase/7/docs/api/.

2.4 Contrasting procedural and object oriented programming

An entire program can be viewed in terms of the relationships between different objects, an approach known as *object-oriented* programming. Early programming, both historically and in your Part I C++ course, was *procedural.* In this case, data is something owned by the whole program, and is passed around between different functions which act upon it. Object-oriented programming is a different way of looking at a problem, where the data is divided up and owned by the different objects. The objects represent the different conceptual elements in the program, and these objects provide all of the functions and services needed for the program. For example, in the Sudoku example we will be using two types of objects: `elements`, which we have already discussed, and `sudokus`, which are objects comprising a nine by nine array of elements and a set of methods for solving and displaying the puzzle.

We have already noted some specific strengths of object oriented approach that relate to the problem we are considering here. A more general strength of the object oriented approach is that the structure encourages us to write code in small modules. In general, regardless of the programming language and style you are adopting, it is best to avoid writing huge, monolithic programs. There are two main reasons for this. The first is that it is very hard to debug such programs. It is much better to break down the program into the smallest testable units and methodically test each one as you go along. Unit testing suites make this process even simpler and we'll make use of one such testing suite in this lab. The second reason is that it is often efficient and more reliable to reuse code and whilst it is very likely that you can reuse small units, it is highly unlikely that you will reuse large monolithic programs. Code reuse is important because rewriting the same code is time consuming and bugs can get introduced - we're much better off using trusted code that we've tested in many situations before.

2.5 The structure of an Android App

Here we briefly describe the organisation of an Android App. Through the course of the lab you will learn more about some of the components described below, but it is useful to have a broad understanding of the organisation.

- **AndroidManifest.xml** The manifest file describes the fundamental characteristics of the app and defines each of its components.

- **src/** Directory containing the app's main source files. By default, it includes an *Activity class* that runs when your app is launched using the app icon (in our case this is **src/com/example/myfirstapp/MainActivity.java**). The Java code for the app will be placed in here too.

- **res/** Contains several sub-directories containing app resources including

  - **layout/** Directory for files that define the app's user interface, for example **main.xml** which specifies the GUI.
  - **values/** Directory for other various XML files that contain a collection of resources, such as **strings.xml** which defines the strings used in the app.

- **bin/** Directory containing the .apk binaries that can be installed on an Android device. The binaries are compiled by the Ant build script. For example, the binary for our app will be called **MyFirstApp.apk**

In Android Terminology an `activity` is the name given to a class describing a single, focused thing that the user can do.

For more information about Android App development, see the Android Developer site `http://developer.android.com/index.html`. The Android API can be found here: `http://developer.android.com/reference/packages.html`

In addition to the directories above, we provide you with two more: **java/**, which contains incomplete Java code for a stand-alone Sudoku solver, and **docs/**, which contains a copy of this document (**3F6_lab.pdf**) and a text file containing unsolved Sudoku puzzles (**selection-puzzles.txt**).

2.6 Linux command line reference

Through the course of the lab, you will have to use several different commands. Here we summarise these commands for your reference:

command	description	example
`ls`	list the contents of directory	`ls` *directory*
`cd`	change into a directory	`cd` *directory*
`cp`	copy file to another location	`cp` *file destination*
`emacs`	open file in text editor	`emacs` *file*
`man`	view manual for the command	`man` *command*

3 Method

Experimental technique and apparatus

You are provided with a computer running a variant of the Linux operating system and a software development environment (the emacs text editor, a Java compiler and the Android software development kit).

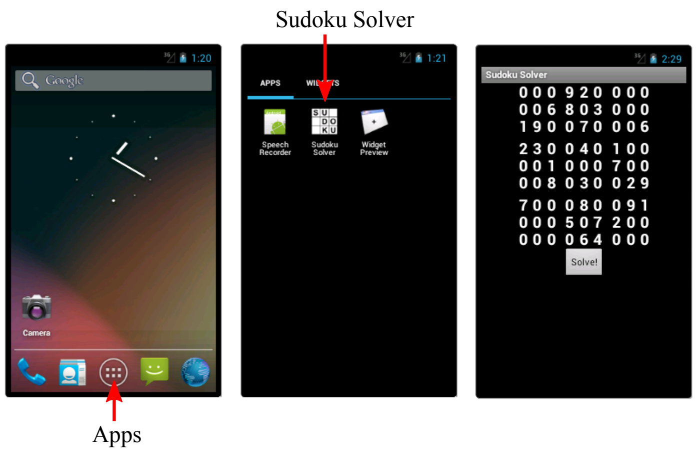
To start the experiment, log on to the computer and select 3F6 from the menu (Applications>All CUED Applications>3rd Year>Start 3F6). This starts the android emulator (more on this in a moment), creates a directory called `myfirstapp` containing all the pre-written programs and opens a command window in that directory ready to start the laboratory. You can see a list of all the directories provided to you by typing `ls` in the command window.

We provide you with an incomplete Sudoku solver application. The lab will involve three steps:

1. modify the app GUI to include a Solve! button
2. complete an object oriented Java Sudoku solver
3. hook up the app to the Java Sudoku solver to turn it into a working application

Compiling Apps and running them on an emulator [10 minutes]

First we need to learn how to run applications on an Android Virtual Device (AVD). The AVD is an emulator that enables you develop and test applications without using a physical device. The emulator will have automatically started when you selected the lab, but it takes some time to initialise. If the emulator stops at any point, you can restart it from the command line:



```
0 0 0 9 2 0 0 0 0
0 0 6 8 0 3 0 0 0
1 9 0 0 7 0 0 0 6
2 3 0 0 4 0 1 0 0
0 0 1 0 0 0 7 0 0
0 0 8 0 3 0 0 2 9
7 0 0 0 8 0 0 9 1
0 0 0 5 0 7 2 0 0
0 0 0 0 6 4 0 0 0
```

Figure 3: The android emulator. To unlock the emulator click on the lock symbol and drag to the unlocked symbol. To open the Sudoku solver app, select the app menu as shown above, scroll to the right, and double click. In the next part of the lab, we'll add the solve button shown in the final image above.

```
>> emulator @avd1.1 &
```

(The `&` character lets the emulator run in the background while we install/uninstall apps to it.) Now let's install the incomplete Sudoku solver application to the virtual device. From the command line, type:

```
>> adb install bin/MyFirstApp-debug.apk
```

This will install the Sudoku solver to the device. Find the new application and run it (see figure 3 if further information is required). You will see that the current version of the application is just a GUI for entering the values of a Sudoku grid. You can enter values by clicking on an element and using the numeric key-pad, but the real functionality needs to be added. Next we're going to build the app in debug mode. Let's make a simple change to the app so that we can verify the build and installation has worked. Change the name of the app by editing the string-resources using `emacs`,

```
>> emacs res/values/strings.xml &
```

Alter the `app_name` field entry, which corresponds to the text shown at the top of the app once it is opened, and save the changes. Now build the app by running the following command from the root directory of your project:

```
>> ant debug
```

The build script writes out information which is useful for debugging purposes if the programme fails to compile.

In order to update the app on the emulator, first uninstall the old version. You may do this either by dragging the app icon into the uninstall waste bin, or from the command line using:

```
>> adb uninstall com.example.myfirstapp
```

Then install the new version as before. Verify that the name of the app has changed as expected. Now we know how to build an app and install it on an emulator we'll start to add the functionality.

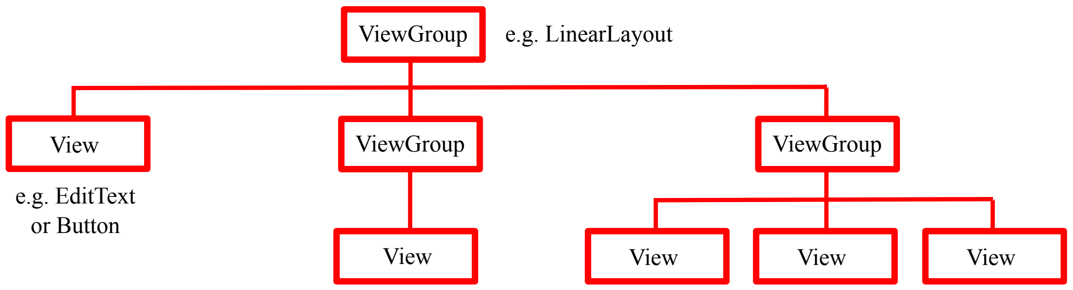


Figure 4: The XML tree: Illustration of how ViewGroup objects form branches in the layout and contain other View objects.

Adding a button [10 minutes]

We're going to begin by adding a button to the app GUI. Ultimately, when the user clicks on this button the app will provide a solution to the current Sudoku grid, but for now we'll just add the button.

The GUI is specified in the XML file `res/layout/main.xml`. Open up this file. The graphical user interface for an Android app is built using a hierarchy of View and ViewGroup objects organised into a tree (see figure 4). View objects are usually user interface 'widgets' such as buttons or text fields whilst ViewGroup objects are invisible view containers that define how the child views are laid out. Draw the hierarchy of objects used in the Sudoku app GUI. Note whether each object is a View or ViewGroup object and the relationship to the Sudoku puzzle. Use short-hand to denote repeated elements.

Now that we understand the structure of the GUI add a `<Button>` to the layout, beneath the Sudoku board:

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/button_solve"
    android:onClick="solveSudoku" />
```

Here the `layout` attributes are set to `"wrap_content"` so that the button size is determined by the text which appears on it. The text itself is defined in the `text` attribute. This is specified as a resource string in the same file we defined the name of the app, `res/values/strings.xml`. We need to add this new string to the resource file. Open it and add to it the button text: `<string name="button_solve">Solve!</string>` Why do you think it is useful to define all of the text used in an app in a resource file? The final `onClick` attribute of the button is the name of the Java method in your activity that the system calls when the user clicks the button. We'll add this method later in the lab. For now, build the app, check that it compiles correctly and that it runs on the emulator. Of course, clicking the button will crash the app since we have yet to write the `solveSudoku` method. Finally, add the button to the schematic showing the hierarchy of the Sudoku GUI.

Adding a method to the Element class [15mins]

The long-term goal is to write the `solveSudoku` Java method which is called when we press the Solve! button. We are going to break this task into pieces and initially concentrate on writing a stand alone object oriented Java Sudoku solver. Developing the Java solver in isolation will make it easy to test the solver, before integrating it into the app.

The incomplete Java code for the solver lives in the directory `myfirstapp/java/`. Change into this directory,

```
>> cd java
```

Open the `Element.java` class in `emacs`. Objects from this class represent one of the 81 elements in a Sudoku board. The class has two fields: `val` which is the integer value of the element (undetermined elements are assigned a value of '0') and `candidates` which is a Boolean array with nine entries indicating the possible values the element might take. The `candidates` array will be initialised with all entries equal to `true`. But as the solver proceeds, these candidates will be whittled down by ruling out the possible values each element might take. How can other classes modify the `candidate` field of an `Element` object? Argue why this might be a sensible design choice bearing in mind the principles of object oriented programming.

The solver will need a method for finding the possible values which an element can take on. For example, if `candidates = {true, false, true, false, false, false, true, true, false}` then the method should return the array of integers `{1,3,7,8}`. Write this method using the following skeleton code:

```
public int[] getCandidateVals(){
     // return the candidate values remaining for an element
     ...
}
```

The existing method `countCandidates()` should be be useful, both as a subroutine and because you might want to use a similar structure for the new method.

The new method can be tested using the unit tests that have already been written to test the `Element` class. The code can be compiled and tested from the command line:

```
>> javac -cp /usr/share/java/junit4.jar Element.java TestElement.java
>> java -cp /usr/share/java/junit4.jar:. org.junit.runner.JUnitCore TestElement
```

Initially, the final test will be failed since this is the one that tests the `getCandidateVals()` method (if you want to take a look at the tests, open `TestElement.java`). This is an example of **test-driven design** where the tests are written before the programs that will pass them. The output from an unsucessful test is useful for debugging purposes. A successful implementation will result in all the tests being passed:

```
JUnit version 4.8.2
......
Time: 0.006

OK (6 tests)
```

You are now ready to write the Sudoku solver.

Writing a Sudoku solver method in Java [55 minutes]

Open the `Sudoku.java` class. This class has a single field which is a 9 by 9 array of Element objects. Take a look at the method `solveSudoku()`. This initially applies the row, column, and sub-square constraints to eliminate candidates and fill in any values when one candidate remains. It then calls the `recursiveSolveSudoku()` method which is the heart of the Sudoku solver. In this section you'll have to write this method. One way of doing this is outlined using pseudo-code below.

```
function recursiveSolveSudoku( )
    locate an undetermined element
    compute possible values of undetermined element
    for all possible values of undetermined element do
        set boardOld to a copy of the board
        assign the undetermined element to the current possible value
        update board by imposing row/column/subsquare constraints
        if board is solved then
            return
        end if
        if board is consistent then
            call recursiveSolveSudoku()
        end if
        if board is solved then
            return
        end if
        restore board to the copy in boardOld
    end for
end function
```

Describe how the solver above works. Your answer should include: descriptions of sensible ways to choose the undetermined element; the logic at the end of the method; why the function calls itself; and, why the first copy operation must make a full copy of the object, but the restore operation can make a copy of the reference to the object.

Implement the solver - you are welcome to depart from the approach outlined in the pseudo-code if you wish. The `Sudoku` class has a number of methods that you will find very helpful, the key ones being:

- `findLeastCandidates()` returns the row and column of the undetermined element with fewest possible candidates remaining
- `cloneBoard()` makes a deep copy of the current board
- `removeCandidateAll(val,row,col)` eliminates candidates equal to `val` from all of the elements in each row `row`, column `col` and the sub-square defined by that location
- `isSudokuSolved()` checks whether the board is solved
- `isConsistent()` checks whether the current board is consistent (e.g. that an incomplete board does not violate the constraints)

Once again, the solver can be tested using the unit tests that have already been written to test the `Sudoku` class. If you want to take a look at the tests, open `TestSudoku.java`. These tests show you how to construct a Sudoku object and how to call various methods. (For debugging purposes you may find it useful to uncomment the marked lines to see the solution your solver provides to the test Sudokus.) The tests can be run from the command line:

```
>> javac -cp /usr/share/java/junit4.jar Element.java Sudoku.java TestSudoku.java
>> java -cp /usr/share/java/junit4.jar:. org.junit.runner.JUnitCore TestSudoku
```

These tests verify that your code successfully solves a selection of Sudoku problems of varying difficulty.

When your solver passes the unit-tests, bingo, you'll be ready to add the Java code into the Android App. To prepare for this step, first copy the Java source code for the two classes into the directory where the app Java code lives. (The `java` directory is not actually part of the app - it is just a place to hold the code whilst we test it.)

```
>> cp Element.java Sudoku.java ~/myfirstapp/src/com/example/myfirstapp/
```

Then add the line: `package com.example.myfirstapp;` to the very top of the `Element` and `Sudoku` class files to indicate they are part of the app's package. For example, when you're done, the top of the `Sudoku` class should read:

```
package com.example.myfirstapp;
import java.util.*;

public class Sudoku{
   ...
```

When that's done, we're ready to complete the app.

Connecting the Solve! button to the solver [30 minutes]

The final step involves hooking up the button to the Java solver and displaying the result. Open the `MainActivity.java` file which is located with the app's Java code. This class runs when your app is launched using the app icon. We must add the method which is called when the Solve! button is clicked. Remembering that we set the `android:onClick` signature to `solveSudoku`, we add a method of that name:

```
/** Called when the user clicks the Solve! button */
public void solveSudoku(View view) {
   ...
}
```

Notice that the method must be public, have a void return value, and have a View as the only parameter (this will be the View that was clicked).

Next, you must fill in this method to read the contents of the text fields, create a Sudoku object, solve the Sudoku, and deliver the solved board to another activity. Although this sounds like a lot of work, we have already written methods that solve the basic steps. First use the `getBoard()` method (see the bottom of the `MainActivity`) to return an integer array corresponding to the current board, then produce a new Sudoku object from this array using the `Sudoku` constructor. Solve the Sudoku using the method `solveSudoku()` and read the solution into a string using `getBoardString()` method. Remember to comment your code adding a brief description.

The final task is to pass the string to an activity which will display the board. This is handled by an `Intent`, which is an object that provides runtime binding between separate components (such as two activities). The `Intent` represents an app's "intent to do something". You can use intents for a wide variety of tasks, but most often they're used to start another activity. Inside the `solveSudoku()` method add the following intent to start an activity called `DisplaySolvedSudoku`:

```
// Do something in response to button
Intent intent = new Intent(this,
                           com.example.myfirstapp.DisplaySolvedSudoku.class);
```

An intent not only allows you to start another activity, but it can carry a bundle of data to the activity as well. In our case we want to pass the solved Sudoku board as a `String`. Assuming that the string is called `solvedBoardString`, add the lines:

```
intent.putExtra(SOLVED_BOARD, solvedBoardString);
startActivity(intent);
```

An Intent can carry a collection of various data types as key-value pairs called extras. The `putExtra()` method takes the key name in the first parameter and the value in the second parameter. The `startActivity` method then begins the new activity.

In order for the next activity to query the extra data, you should define your key using a public constant. To that end, add the `SOLVED_BOARD` definition to the top of the `MainActivity` class:

```
public class MainActivity extends Activity {
   public final static String SOLVED_BOARD = "com.example.myfirstapp.MESSAGE";
   ...
```

We now have to add a second activity called `DisplaySolvedSudoku.java` that will display the solution. Create a file with the following content:

```
package com.example.myfirstapp;
import android.app.Activity;
import android.content.Intent;
import android.widget.TextView;
import android.view.View;
import android.os.Bundle;
public class DisplaySolvedSudoku extends Activity {
    @Override
        public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // Block 1
        Intent intent = getIntent();
        String solvedBoardString = intent.getStringExtra(
                           com.example.myfirstapp.MainActivity.SOLVED_BOARD);

        // Block 2
        TextView textView = new TextView(this);
        textView.setTextSize(35);
        textView.setText(solvedBoardString);

        // Block 3
        setContentView(textView);
    }
}
```

Broadly, describe what do the three blocks of code do.

Finally, you must declare all activities in your manifest file, `AndroidManifest.xml`, using an `<activity>` element. Open the `AndroidManifest.xml` and add to it:

```
<application>
    ...
    <activity
       android:name=".DisplaySolvedSudoku"
       android:label="@string/title_activity_display_message" >
       <meta-data
           android:name="android.support.PARENT_ACTIVITY"
           android:value="com.example.myfirstapp.MainActivity" />
    </activity>
</application>
```

Notice that you need to add another string to the string resource file,

`<string name="title_activity_display_message">Sudoku Solution:</string>`.

Test the solver by entering in the Sudoku values shown in figure 1A and pressing the Solve! button. Verify that the solution matches that given in figure 1B. If you would like to try other Sudokus, a selection of puzzles of varying difficulty can be found in `java/selection-puzzles.txt`. What happens when you try and solve a Sudoku puzzle with no given values? Is this a sensible solution? What happens if the Sudoku puzzle contains given values which are contradictory? Why? How could you modify the code to provide more informative feedback?

If you have a device running the Android OS you may want to install the app by following the instructions at http://developer.android.com/training/basics/firstapp/running-app.html.

4 Writing Up

You should refer to the advice given in the *Guide to the Engineering Tripos Part IIA*, Sections 3.3.2 and 3.3.3.

The Laboratory Report [3 hours]

This should be no more than five sides in long-hand or three if word processed, excluding any diagrams or program listings. The report should describe the stages you went through while developing the Sudoku Solver App. You should illustrate each stage with the pieces of code you added. It is not necessary to include full program listings. You should describe and discuss any important results and findings, in particular by responding to all the questions underlined in Section 3.

The Full Technical Report [10 additional hours]

Guidance on the preparation of Full Technical Reports is provided both in Appendix I of the *General Instructions* document and in the CUED booklet *A Guide to Report Writing*, with which you were issued in the first year. Remember that the Full Technical Report should be a stand-alone report, restating the original experiment and results, as well as answering the extra points listed in this section.

In writing any report it is not necessary to stick rigidly to traditional section headings; instead, you are just trying to lead the reader logically through the story you are trying to tell. You might consider modifying the traditional report structure to the one given below:

Introduction Describe the main goals of the laboratory.

Theory: Object Oriented Programming Describe the basic principles underpinning object oriented programming. Use Sudoku as a running example to illustrate your points.

Theory: Android Mobile Phone App Summarise the organisation of an Android mobile phone application (use schematics where possible).

Object oriented Java: A Sudoku solver Describe this part of the laboratory and your findings, answering the various questions posed in the laboratory reports.

Android App Development: Adding functionality to a GUI Describe this part of the laboratory and answer the questions posed.

Conclusion Draw your discussion to an end.

If you are offering a Full Technical Report for this experiment, you should discuss the following points in addition to those listed above for the Laboratory Report:

- Describe extensions to the Java code that would be needed to:
  - **Produce new Sudoku puzzles using an object oriented approach**. Discuss how to modify the Java code to automatically generate random Sudoku puzzles. Puzzles that can only be solved by guessing are frowned upon - how can you ensure that your puzzles fulfil this requirement? You may like to discuss how a simple Sudoku solver, like the method `simpleSolveSudoku()`, could be used for such a purpose. How hard would the resulting puzzles be? How might you modify the method to control the difficulty puzzles? The given values in newspaper-ready problems are symmetrically arranged. Discuss how to generate symmetric puzzles. Concentrate on the main ideas and design choices - you do not need provide detailed working Java code.
  - **Solve more complex Sudoku problems with different constraint sets**. Newspaper Sudoku's come in many different shapes and sizes (see figure 5). Describe how you would modify the Java code to solve these complicated puzzles. The key is to find an elegant way to specify constraints in a flexible way. Describe any new objects or methods you might add for this purpose. Concentrate on the main ideas and design choices - you do not need provide detailed working Java code.

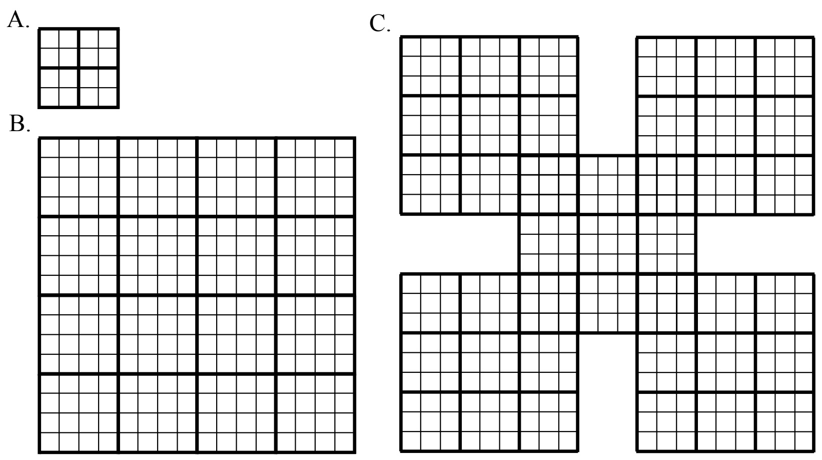


Figure 5: Sudoku variants. A) Quaduko grids contain values from the set $\{1, 2, 3, 4\}$ with the same constraints as Sudoku. B) Similarly, Hexdoku contain values $\{1, \ldots, 16\}$. C) Some puzzles involve more complicated sets of constrains, in this puzzle each of the 9 by 9 grids must be played like a normal Sudoku board. This introduces additional constraints on the sub-squares at the corners of the central Sudoku board.

- **Research software testing**. Describe why unit tests and integration tests might be useful in complex software engineering projects. Discuss test driven development. Do you think it is sensible? Discuss briefly why an automated testing framework such as JUnit is useful. Describe various principles for devising unit tests. Discuss the pros and cons of using random numbers in tests. Use examples from `TestSudoku.java` and `TestElement.java` to illustrate your points. Do you think it is possible to test every contingency in a complicated programme? What consequences does this hold for unit tests and integration tests?

You should include your Laboratory Report as an appendix and refer to it where appropriate.

A Java Program Listing

Element.java

```
import java.util.*;

public class Element{

    //fields
    private int val; // value taken by element (0=?)
    private boolean[] candidates; // candidate elements

    //constructor (takes in value and a list of candidates)
    public Element(int inVal, boolean[] inCandidates){
        val = inVal; // val = {0,...,9}
        // candidates = boolean array denoting possible values of
        // element remaining out of nine
        candidates = new boolean[9];

        for (int k=0; k<9; k++){
            candidates[k] = inCandidates[k];
        }
    }

    //methods
    public int getVal(){
        // method for getting the value of the element
        int curVal = val;
        return curVal;
    }

    public void setVal(int newVal){
        // method for setting the value of the element
        val = newVal;
    }

    public boolean[] getCandidates(){
        // method for getting the candidates
        boolean[] outCandidates = new boolean[9];

        for (int k=0; k<9; k++){
            outCandidates[k] = candidates[k];
        }
        return outCandidates;
    }

    public boolean excludeCandidate(int cand){
        // method for removing a candidate, and updating the value field if
        // only one candidate remains

        boolean updateVal = false;

        if (val !=0){
            // if val is not equal to zero, set all but one candidate to false
                    candidates = new boolean[9];
                candidates[val-1] = true;
        }
        else{
            // removed candidate always different from val
            candidates[cand-1] = false;

            // if there is just one candidate remaining and value is
            // unassigned update value
            int numTrue = countCandidates();
            if (numTrue==1 && val==0){
                for(int i=0; i<candidates.length; i++){
                    if(candidates[i]){
                        setVal(i+1);
                        updateVal = true;
                    }
                }
            }
        }
        return updateVal;
    }

    public int countCandidates(){
        // method for counting candidates
        int numTrue = 0;

        for(int i=0; i<candidates.length; i++){
            if(candidates[i]) numTrue++;
        }
        return numTrue;
    }

    public Element clone(){
        // clone the element - necessary to keep old values for back tracking
        Element outElement = new Element(val,candidates);

        return outElement;
    }


}
```

Sudoku.java

```
import java.util.*;

public class Sudoku{

    //fields
    private Element[][] board;

    //constructor
    public Sudoku(int[][] inBoard){

        board = new Element[9][9]; // board = array of elements

        // initialise board using given values and set all candidates
        // to possible - no constraints are imposed yet
        boolean[] candidates = new boolean[9];
        Arrays.fill(candidates, true);

        for (int i=0; i<9; i++){
            for (int j=0; j<9; j++){
                board[i][j] = new Element(inBoard[i][j],candidates);
            }
        }
    }

    //methods
    public int[][] getBoard(){
        // function for getting the board field
        int[][] boardOut = new int[9][9];
            for (int i=0; i<9; i++){
            for (int j=0; j<9; j++){
                boardOut[i][j] = board[i][j].getVal();
            }
        }
        return boardOut;
    }

    public void removeCandidateCol(int val,int col){
        // removes a candidate from a column
        boolean updateVal;

        for (int i=0; i<9; i++){
            updateVal = board[i][col].excludeCandidate(val);

            if (updateVal==true){
                removeCandidateAll(board[i][col].getVal(),i,col);
                    }
        }
    }

    public void removeCandidateRow(int val,int row){
        // removes a candidate from a row
        boolean updateVal;

        for (int i=0; i<9; i++){
            updateVal = board[row][i].excludeCandidate(val);
            if (updateVal==true){
                removeCandidateAll(board[row][i].getVal(),row,i);
                    }
        }
    }

    public void removeCandidateSubsq(int val,int subsq){
        // removes a candidate from a subsquare
        boolean updateVal;

        int curRow;
        int curCol;
        for (int i=0; i<3; i++){
            for (int j=0; j<3; j++){

                if (subsq<3){
                    curRow = i;
                }
                else if (subsq<6){
                    curRow = i+3;
                }
                else{
                    curRow = i+6;
                }

                curCol = j+3*(subsq%3);

                updateVal = board[curRow][curCol].excludeCandidate(val);

                if (updateVal==true){
                    removeCandidateAll(board[curRow][curCol].getVal(),
                                       curRow,curCol);
                }

            }
        }
    }

    public void removeCandidateAll(int val,int row,int col){
        // removes a candidate from a row, col, subsq. This is a
        // recursive function - the methods called below themselves
        // call removeCandidateAll if another elements value has been
        // determined

        if (val>0){
                int subsq = rowCol2Subsq(row,col);
                removeCandidateRow(val,row);
                removeCandidateCol(val,col);
                removeCandidateSubsq(val,subsq);
        }
    }

    public void simpleSolveSudoku(){
    // solve as far as possible by excluding given values from the
    // row/col/subsq
        for (int i=0; i<9; i++){
            for (int j=0; j<9; j++){
                removeCandidateAll(board[i][j].getVal(),i,j);
            }
        }
    }

    private Element[][] cloneBoard(){
        // copy the board - necessary to keep old values for back tracking
```

```
        Element[][] outBoard = new Element[9][9];

        for (int k=0; k<9; k++){
            for (int l=0; l<9; l++){
                outBoard[k][l] = board[k][l].clone();
            }
        }
        return outBoard;
    }

    public int[] findLeastCandidates(){
        // find the element with the least number of candidates
        // output is an array {row, column}
        int[] leastEle = new int[2];
        int numCandidates = 10;
        int curNumCandidates;

        for (int i=0; i<9; i++){
            for (int j=0; j<9; j++){

                curNumCandidates = board[i][j].countCandidates();

                if (curNumCandidates<numCandidates && curNumCandidates>1){

                    leastEle[0] = i ; // row
                    leastEle[1] = j ; // col
                    numCandidates = curNumCandidates;
                }
            }
        }
        return leastEle;
    }

    public void solveSudoku(){
    // solve the sudoku board

            // eliminate candidates by applying row/col/subsq constraints
            // to the values initially give
            simpleSolveSudoku();

        // recursively solve the board by guessing and checking for
        // consistency
        recursiveSolveSudoku();
    }

    public int countSolved(){
        // count how many elements are solved
        int numSolved = 0;

        for (int i=0; i<9; i++){
            for (int j=0; j<9; j++){

                if (board[i][j].countCandidates()==1){
```

```
                numSolved++;
                }
            }
        }
        return numSolved;
    }

    public int countNumCandidates(){
        // count how many candidates remain
        int numCand = 0;

        for (int i=0; i<9; i++){
            for (int j=0; j<9; j++){
                numCand = numCand + board[i][j].countCandidates();

            }
        }
        return numCand;
    }

    public boolean isSudokuSolved(){
        // figures out whether the board has been solved by:
        // 1) checking each element has only one candidate remaining
        // 2) checking that there are no duplicates in each row/col/subsq
        boolean solved = false;

        int numSolved;
        numSolved = countSolved();

        if (numSolved==81&&!checkDuplicates()){
            solved = true;
                }

        return solved;
    }

    public boolean checkDuplicates(){
        // figures out whether there are conflicts in sudoku grid and
        // returns true if there are
        boolean duplicates = false;

        if (checkDuplicatesRows()||checkDuplicatesCols()||
            checkDuplicatesSubsq()){
            duplicates = true;
        }

        return duplicates;
    }

    public boolean isConsistent(){
        // checks that the constraints are not violated and that each
        // element has at least one candidate
        boolean consistent = false;
        if (!checkDuplicates()&checkCandidates()){
            consistent = true;
        }
        return consistent;
    }

    public boolean checkDuplicatesRows(){
        // figures out whether there are duplicates in rows of a grid
        // returns true if there are
        int[] counts;
        boolean duplicates = false;

        for (int i=0; i<9; i++){
            counts = new int[10];
            for (int j=0; j<9; j++){
                counts[board[i][j].getVal()]++;
            }

            if (counts[1]>1 || counts[2]>1 || counts[3]>1 || counts[4]>1 ||
                counts[5]>1 || counts[6]>1 || counts[7]>1 || counts[8]>1 ||
                counts[9]>1){
                duplicates = true;
            }

        }

        return duplicates;
    }

    public boolean checkDuplicatesCols(){
        // figures out whether there are duplicates in columns of a grid
        // returns true if there are
        int[] counts;
        boolean duplicates = false;

        for (int i=0; i<9; i++){
            counts = new int[10];
            for (int j=0; j<9; j++){
                counts[board[j][i].getVal()]++;
            }

            if (counts[1]>1 || counts[2]>1 || counts[3]>1 || counts[4]>1 ||
                counts[5]>1 || counts[6]>1 || counts[7]>1 || counts[8]>1 ||
                counts[9]>1){
                duplicates = true;
            }

        }

        return duplicates;
    }
```

```
public boolean checkDuplicatesSubsq(){
    // figures out whether there are duplicates in subsquares of a board
    // returns true if there are
    int[] counts;
    int col;
    int row;
    boolean duplicates = false;

    for (int i=0; i<9; i++){
        counts = new int[10];

        for (int j=0; j<3; j++){
            for (int k=0; k<3; k++){
                if (i<3){
                    row = j;
                }
                else if (i<6){
                    row = j+3;
                }
                else{
                    row = j+6;
                }

                col = k+3*(i%3);
                counts[board[row][col].getVal()]++;
            }
        }
        if (counts[1]>1 || counts[2]>1 || counts[3]>1 || counts[4]>1 ||
            counts[5]>1 || counts[6]>1 || counts[7]>1 || counts[8]>1 ||
            counts[9]>1){
            duplicates = true;
        }

    }

    return duplicates;
}

public boolean checkCandidates(){
    // checks that every element has at least one candidate value
    boolean candidates;
    for (int i=0; i<9; i++){
        for (int j=0; j<9; j++){
            int cnt = board[i][j].countCandidates();
            if (cnt==0){
                return candidates = false;
            }
        }
    }
    return candidates = true;
}

int rowCol2Subsq(int row, int col){
    // method for getting the subsquare of the element (0-8)

    int xPos;
    int yPos;
    int subsqCur;

    if (col<3){
        xPos = 1;
    }
    else if (col<6) {
        xPos = 2;
    }
    else {
        xPos = 3;
    }

    if (row<3){
        yPos = 1;
    }
    else if (row<6) {
        yPos = 2;
    }
    else {
        yPos = 3;
    }

    subsqCur = xPos+3*(yPos-1)-1;
    return subsqCur;
}

public void printSudoku(){
    // prints out a board to the screen with information about the soln
    String str;
    System.out.println("\n");
    System.out.println("Number of elements solved: "+countSolved()+"/81");
    System.out.println("Number candidates: " + countNumCandidates());

    if (checkDuplicates()){
        System.out.println("There are duplicates in the board");
    }
    else{
        System.out.println("There are no duplicates in the board");
    }

    if (checkCandidates()){
        System.out.println("Every element has at least one candidate");
    }
    else{
        System.out.println("One or more elements have no candidates");
    }
    System.out.println("\n");
    String boardString = getBoardString();
    System.out.println(boardString);
    System.out.println("\n");
}

public String getBoardString(){
    // returns the board as a string with line breaks at the row
    // ends for display
    String boardString = "";
    for (int i=0; i<9; i++){
        for (int j=0; j<9; j++){
            boardString = boardString +
                Integer.toString(board[i][j].getVal()) + " ";
            if (j==2 || j ==5){
                boardString = boardString + "   ";
            }

            }
        boardString = boardString + "\n";

        if (i==2 || i ==5){
                boardString = boardString + "\n";
            }
    }
    return boardString;
}
}
```

---

main.xml

```
<?xml version="1.0" encoding="utf-8"?>

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:descendantFocusability="beforeDescendants"
    android:focusableInTouchMode="true"
>

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="12pt"
    android:gravity="center_horizontal"
>

<EditText android:id="@+id/cell_11"
          style="@style/sudoku_number"
/>

<EditText android:id="@+id/cell_12"
          style="@style/sudoku_number"
/>
```

```
<EditText android:id="@+id/cell_13"
          style="@style/sudoku_number"
/>


<TextView android:id="@+id/space_1"
          style="@style/h_space"
          android:text=""
/>

<EditText android:id="@+id/cell_14"
          style="@style/sudoku_number"
/>

<EditText android:id="@+id/cell_15"
          style="@style/sudoku_number"
/>

<EditText android:id="@+id/cell_16"
          style="@style/sudoku_number"
/>

<TextView android:id="@+id/space_1"
          style="@style/h_space"
          android:text=""
/>

<EditText android:id="@+id/cell_17"
          style="@style/sudoku_number"
/>

<EditText android:id="@+id/cell_18"
          style="@style/sudoku_number"
/>

<EditText android:id="@+id/cell_19"
          style="@style/sudoku_number"
/>

</LinearLayout>

<!-- End of row -->

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="12pt"
    android:gravity="center_horizontal"
>

...
</LinearLayout>
```