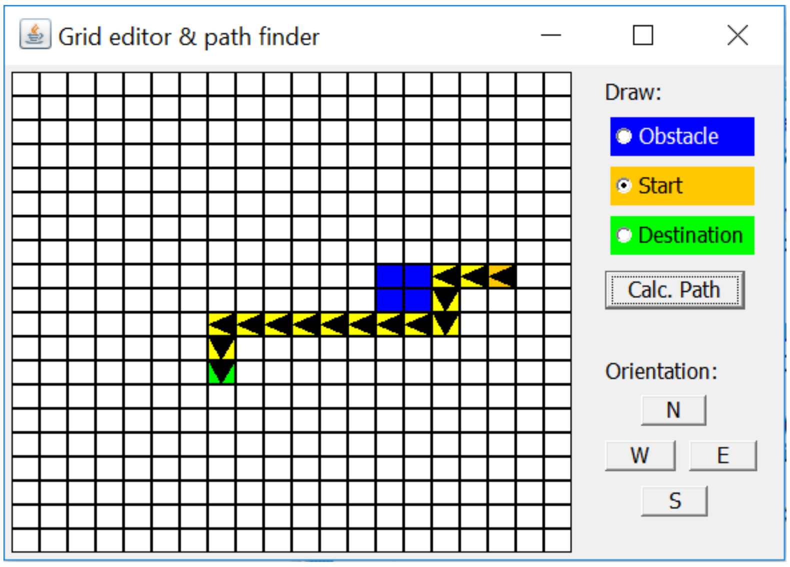


Assignment 7

Content Application Development Day 7
Lecture
In the lecture animation and timers were introduced. This assignment covers several skills learned in previous lectures and assignments, but this time with less detailed explanations.
Tutorial
In the tutorial you will practice extending the interactive map editor with path-finding capabilities. It will calculate turns and distances to travel and includes and appendix with help on how to send this to the car build in previous assignments.

At the end of day 7 you will be able to
- Create a userinterface with advanced interaction
- Use and apply libraries and code imported from online examples
- Apply A* pathfinding algorithm in a Java application.
- Realize a map editor based on tiles (panels).

Reading
Book: *Head First Java*: chapters 12-13,15,16 or from '*Aan de slag met Java*': chapters 9.1-9.4 + 10.1-10.9

Extra exercise
The topics in lesson 7 are provided with examples that can be used in this assignment. You can go through the exercises from the chapters mentioned above to increase your understanding of the subject matter.

Expand the interactive map editor with path-finding capabilities

This assignment will extend the previous built application to make it suitable to generate driving instructions for the previously built Rover car based on the path found. It will:

- Consider the orientation of the car: which direction is it facing?
- Calculate turns and distances to travel to generate driving instructions
- Show the car drive the path as an animation.

The calculated turns and distances to travel should be written in such a way that they can be easily transferred to for instance the previous built Rover car. The instructions will look like:

```
s90d100s-90d30s180d60
```

Where "s" means *steer* and "d" means *drive*. In human language these example instructions are:

Steer 90 degrees clockwise.
Drive 100 cm.
Steer 90 degrees counter clockwise.
Drive 30 cm.
Steer 180 degrees clockwise.
Drive 60 cm.

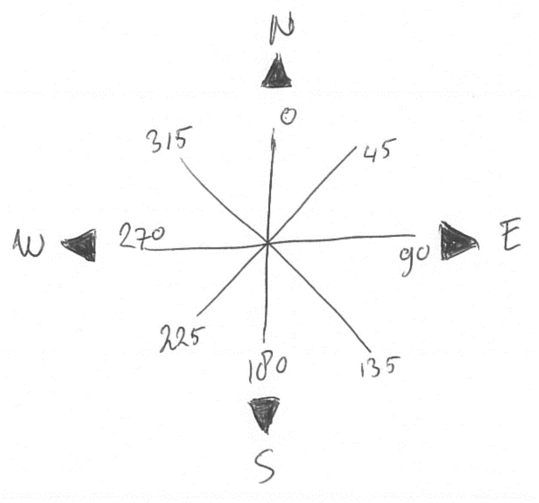
Please note that to make this not too complex, we assume (for the time being) that our car can make point turns, so steer 90 means it will make a point turn of 90 degrees. In reality, your car might not be able to make this kind of turns.

Table of Contents

1. Show orientation of the car in the userinterface

The application must know the orientation of the car, to be able to generate driving instructions.

The application will show the orientation of the car as a triangle (a simple pointer). To accomplish this, change the drawing method of class PixelPanel. The class should already have a class-variable **`orientation`**. If not, add it, and give it an initial value of -1 (= no orientation).



The values of class-variable **`orientation`** will be the amount of degrees: for N 0, E 90, S 180 and W 270 degrees. See also the picture for the 'design' of how to handle orientation.
If for example the orientation is North (N), the variable will have the value 0 and the *paintComponent()* method of class PixelPanel will draw a triangle pointing North.

To draw a triangle facing the proper orientation (angle), use the example below: (add it to the *paintComponent()* method)

```
Graphics2D g2 = (Graphics2D) g;
g2.setColor(Color.BLACK);

// create 2 arrays x,y to hold position for 3 points:
int x[] = new int[3], y[] = new int[3], n = 3;

// create points for triangle facing north:
x[0]=getWidth()/2; y[0]=0; // first point
x[1]=getWidth(); y[1]=getHeight(); // second point
x[2]=0; y[2]=getHeight();  // third point

// Create polygon which represents a triangle with the above vertices:
Polygon triangle = new Polygon(x, y, n);

// Rotate the triangle around it's center according to its orientation:
if (orientation>0)
        g2.rotate(Math.toRadians(orientation), getWidth()/2, getHeight()/2);
// Draw the filled triangle:
g2.fill(triangle);
```

This entire piece of code only has to be executed if the orientation is >= 0. Add an if-statement that prevents execution of this part of the method, by adding an if statement before it.
The if-statement checks if orientation is smaller than 0 and return if that is the case (use **`return;`**).

Next, in the *setStart()* method of the userinterface set the orientation of the panel to 0. We will use this orientation as the default, but it should be possible (from the userinterface) to change it (this is done in the next step). Also, in *setWalkable()*, set the orientation of the panel to -1.

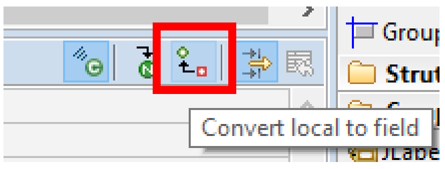
Test the application. With 'Start' selected, and you click a tile, is the north-facing triangle drawn when you click a tile? Click multiple tiles, to check if only one triangle is drawn.

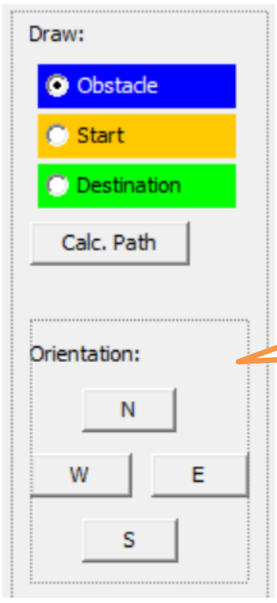
Change the orientation from the userinterface
Add buttons* to the userinterface which allow to change the orientation.

These buttons should only be visible when the "Start" radio button is selected. To accomplish this, put them in a panel, make this panel a field** and show/hide the panel on click of the radio buttons (this is explained later).
To hide the panel at startup, under Advanced properties, remove the check at field 'visible'.

* you are free to choose other userinterface components if you like.

**





The panel with the buttons to change the orientation could look like this

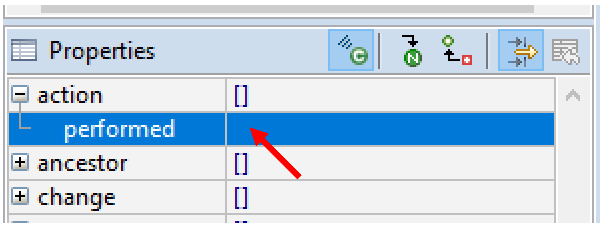
Next, add eventhandlers to the N-E-S-W buttons. Example of code for eventhandler of button N:

```
if (start==null) return; // if no start set yet, do noting
start.setOrientation(0); // north
```

The method setOrientation() is not there yet. Go to the class PixelPanel and generate it (via *Source > Generate Getters and Setters*). Make sure it includes a line with a call to repaint() at the last line of code.

Now add eventhandlers for the E, S and W buttons too. For each call of setOrientation(), change the value of the orientation accordingly (N 0, E 90, S 180 and W 270).

Because we want to show the panel only when radio button 'Start' is active, we must add an eventhandler: On change of radio buttons, add an 'action performed' eventhandler:



Which, for radio button 'Start', makes the panel which contains the N-E-S-W buttons visible:

```
panelOrientation.setVisible(true);
```

(Make sure you made `panelOrientation` a field as described in the beginning of this step. You may have chosen a different name for this panel)

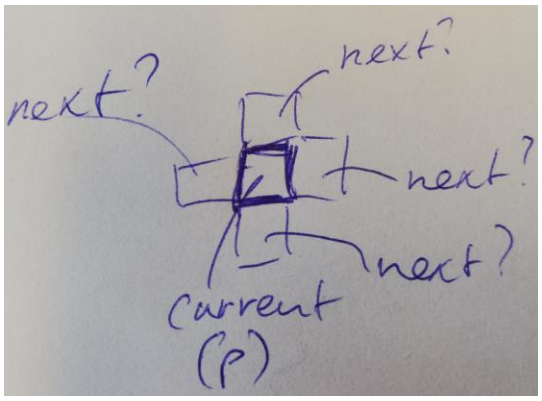
The other two buttons make the panel not-visible.

Run the application and check if the panel is shown only when 'Start' is selected. Also check if the orientation of the triangle changes if you press the N-E-S-W buttons.

2. Keep track of the orientation of the car in the calculated path

We will add code to the method that calculates the path.
First, in the part which clears a previous path (in a for-loop), set the orientation of the panel **p** to -1.

To properly set the orientation for each tile in the path, we must check the location of the next tile in the path and find from the current tile, were it lies. An example:



To find the next panel, use the code below. Add it <u>inside the loop</u> that shows the path, before the call to the *setBackground()* method.

```
if (i==0) { // if we are starting
   // compare orientation of start with panel p
}
if((i+1)<path.size()) {
  PixelPanel next = pixelPanels[path.get(i+1).getxPosition()][path.get(i+1).getyPosition()];
  // compare orientation of panel p with panel next
}
```

The comments are pseudo-code, which must become a call to a new method which compares the orientation with two parameters, the current tile and the next tile.

Add a new method to the class which will do this comparison of the orientations, it should have two PixelPanels as parameters, use variable names **p** and **next** for these (as used in the example code below).

The new method must check for 4 possibilities: the next tile is to the North of the current, or to the East, South or West. We will show you how to check the first option (if it is to the North):

```
// where is the next tile?
if (p.y==next.y+1 && p.x==next.x) // N?
    next.setOrientation(0);
```

In this example we also determine the degrees the car must turn to change its orientation to face the next tile. We will use this to generate driving instructions later.

Add **`else if () ...`** code for the other 3 conditions (E, S or W). For each if-statement the condition is different, and also the parameter for **`next.setOrientation(...)`** will get different values!

After these four if-statements, we can determine a value for the angle we must use to steer to the next tile. We use a new variable 'steer' for this. The value for **`steer`** can be determined as follows:

```
int steer = next.orientation - p.orientation;
if (steer<-180) steer = steer + 360;
if (steer>180) steer = steer - 360;
```

When the method is ready, call it twice in the loop at the 2 spots where you added the code (in the method that calculates the path): it is at the 2 spots with the comment "compare orientation of …". In the first call, we call that method with the start-tile as first parameter, which is the variable **`start`**.

Now run the application and test if the orientation throughout the entire path is shown correctly. Test with multiple turns in the path, so see if all four options are handled correctly.

To show the orientation of the car also in the destination, change the condition of the for-loop back to **`size()`** instead of **`size()-1`** and add this if statement in front of the line that changes the color to yellow:

```
if (p!=destination)
```

Which prevents the destination tile from having its color changed.

3. Generate driving 'instructions'

We will generate a set of instructions which the car which was built in previous assignments 'understands'. The Rover class has methods like *drive()* and *steer()* which can be used for driving. They both have an integer as a parameter, and can be called like these examples:

```
drive(100); // drive one meter
steer(-20); // steer to the left
steer(20); // steer to the right
...
```

To simplify the instructions, we could use a String to transfer the instructions from the Path finding app (the one we are working on now) to the Rover app (the Arduino app previously made).
'Commands' sent from the Path finding app to the Rover app could look like:

```
d100
s-20
d30
```

or packed

```
d100s-20d30
```

together:

Add two class variables which we will use to store the instructions and [a constant] which represents the size of a tile in the real world:

```
private final int tileSizeCM = 30; // size of a tile in cm (in the 'real world')
private int drive = 0; // keep track of distances to drive (straight)
private String driveInstruction = ""; // save the drive instruction
```

Make sure the variables **`driveInstruction`** and **`drive`** are set to their default values (same as above) every time a new path is calculated. Add two lines of code to the method that calculates the path for this.

Variable **`drive`** will be increased for each tile the car drives: if the car for example drives straight for 3 tiles, we simply add 30 (**`tileSizeCM`**) for each tile, so the instruction will become "**`d90`**".

In the method you just created (which does the comparison), we will determine the driving instructions and add them to the String **`driveInstruction`**. In pseudo code, the things that must be done are:

```
if no steering necessary (steer==0) {
   increase drive by tileSizeCM
}
else { // a turn will be made
   // before making a turn, save driving distance (if there):
   if a previous distance to drive was set { // (drive>0):
       add it to the driving instructions (add "d"+drive to the driving instructions)
   }
   // after a turn, we always have to drive to the next tile:
   give drive a (start-)value of tileSizeCM
   add the angle to the driving instructions // (add "s"+steer to the driving instructions)
}
```

Some parts are already written as comments, as these do not lead to code, but are meant to clarify things.

Some hints: to add something to the driving instructions you could use something like:

```
driveInstruction = driveInstruction + "d"+drive;
```

In case of a turn, the last part would be **`"s"+steer`**.
The pseudo-code "give drive a (start-)value of tileSizeCM" simply means: **`drive = tileSizeCM`**.

The generated drive instructions should be almost complete now. Only the last drive is not added to the instructions. To fix that, in the method that calculates the path, at the end, add:

```
driveInstruction = driveInstruction + "d"+drive;
```

After this, print the instructions to the Console, using *System.out.println()*.

Now test if the instructions generated are Ok.

4. Animate the path of the car

To show the movements of the car in a little bit more realistic way, we could animate the path. This means the for-loop which shows the path in the method that calculates the path would have to be replaced by something that shows it in a time based fashion, eg. every step after a short time.

We will realize this with a timer. Add a timer as a class-variable to the Userinterface class:

```
private Timer timer;
```

Import the Timer from the javax.swing library. Initialize it at the end of the constructor:

```
// Create new timer which calls method showPath() every 200ms
timer = new Timer(200, (e)->showPath());
```

Add the method *showPath()* to the class.

Find the for-loop which shows the path in the method that calculates the path. Move (cut) most of the lines inside the loop, except the line with *System.out.println(),* to the method *showPath()* (paste).

After doing this, you will get errors, because the variable **i** which was used as the counter of the for-loop, is not in that method. Add a new variable to be used as a counter as a class-variable and replace all occurrences of variable **i** in *showPath()* with that variable. At the end of the method which calculates the path, initialize the new counter with 0 and start the timer. Increment the counter by 1 at the end of the method *showPath()*.

The timer must be stopped if the counter is equal to the size of the path. Add an if-statement at the start of *showPath()* which checks that and if *true*, stops the timer.

Test the application again, to see if the path is shown animated.

You might notice that the application no longer shows the proper driving instructions. That is because the instructions are now generated while the timer is running and must be shown after the animation is complete. Find the two statements that were added at the end of the method which calculates the path (one with the final addition to the instructions and one which prints the instructions) and put them at the spot where the timer is stopped, in *showPath()*.

Finish

If you did not do this, add a title to the application. Check if you have added sufficient comments in the code. If you have not done so, do it now. Make sure there is Javadoc formatted documentation for each method. In Eclipse you add Javadoc documentation by typing **/**** before the method and then pressing Enter.

Make sure the layout of code is neat. Eclipse can automatically do that for you via *Source > Format* (or CTRL-SHIFT-F).

Test the application and demonstrate it to an assistant or lecturer.

Summary

Today you have created an interactive editor and learned how to realize the logic and interaction for such an application.
In addition, you have learned the following.

- Integrate external code into your own application.
- Applying algorithms like the A* path-finding algorithms in a Java application.
- Using and applying event handlers for multiple components.
- Applying dynamic application logic using selection statements.

Appendix: send the driving instructions to the car

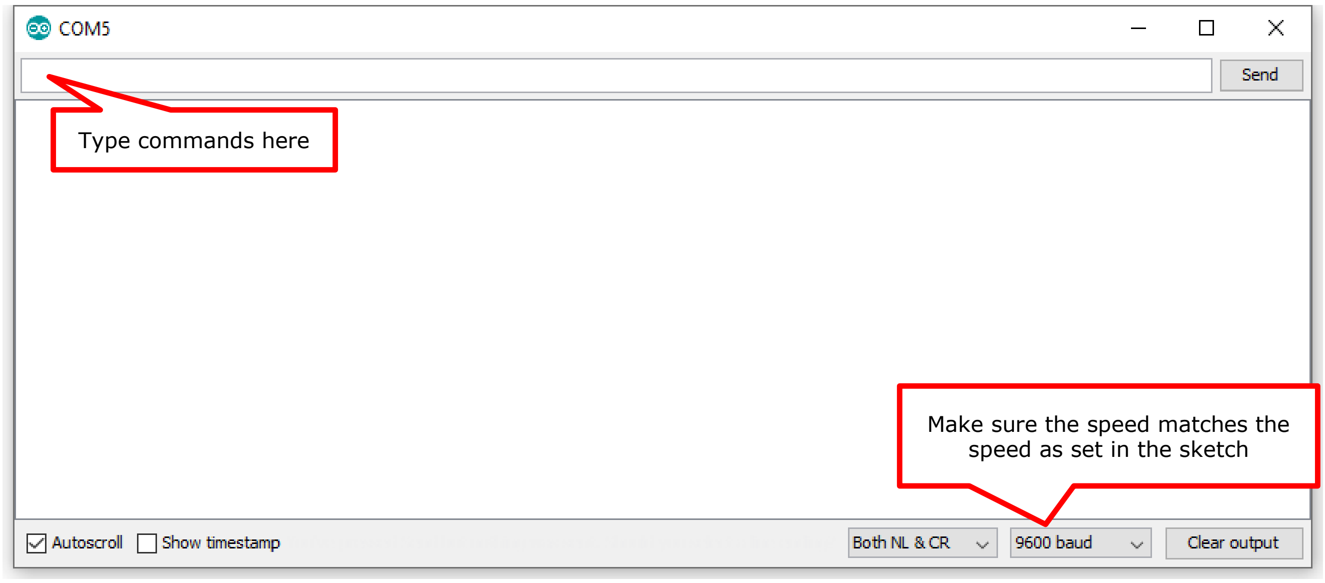
To send the driving instructions to the car, both apps must communicate with each other. In past lectures other examples were given which use communication:

- The weather station app was able to get temperature data from a connected Arduino with temperature sensor (details how to setup communication were given in the appendix of assignment 3).
- An example how to remote control the Rover car with your phone was given in practical assignment 1.
- Further examples of communication were given in practical assignment 3 and for instance these Blog posts.

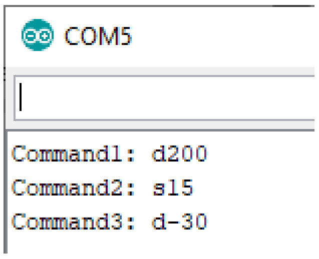
This appendix will show how to send the driving instructions from the Java app to the Arduino running the Explorer sketch via USB, with an addition to make it able to receive the instructions.

Create an Arduino app that can read the driving instructions

Test this example sketch **read_command_data_from_serial.ino**. This sketch can read instructions via the Serial port. To test it run it on an Arduino, leave the USB cable connected, start the Serial Monitor and type commands:



If you for instance type "d200s15d-30", it should give:



```
Command1: d200
Command2: s15
Command3: d-30
```

Now all we must do is add this to the Arduino sketch of the Rover car, and it can read commands.

Make Java app send instructions via Serial connection

How to setup a Serial connection for a Java project (in Eclipse) was detailed in appendix 1 of assignment 3. We will repeat it briefly.

First, add the jSerialComm library to the project. Download it as a .jar file here. In Windows Explorer, copy the .jar file. In Eclipse: right-click the main project folder and paste it into that folder. Next, add the library to the build path: right click it, and select *Build Path > Add to Build Path*.

Import the library by adding the following line at the top of the userinterface class:

```
import com.fazecast.jSerialComm.*;
```

Next, add the example code from this text file to the userinterface class.

Call the **`initializeSerialPort()`** method at the end of the constructor.

Before you start testing, set the proper values for the class-variables **`comPort`** and **`Baudrate`**. They must be the same as used in the Arduino IDE.

Call the send() method with the driving instructions at the same spot as where it is printed:

```
System.out.println(driveInstruction); // print driving instructions
send(driveInstruction); // send it to the Arduino
```

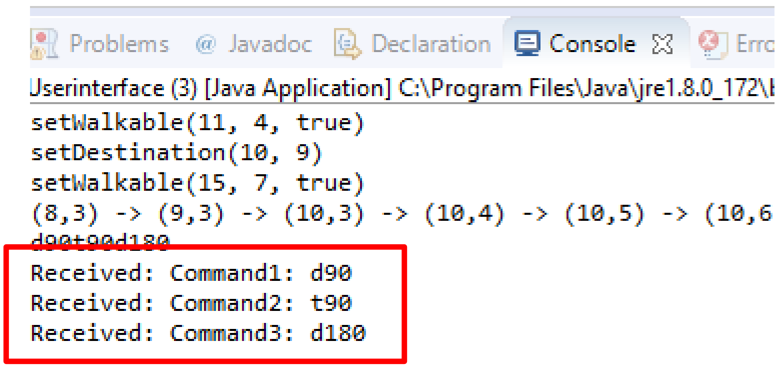
On a Mac, comports are looking different. It might be something like `/dev/tty.usbmodem*` or `/dev/tty.usbserial*`

There is a piece of code in comments (“`// get a list of available ports`”) which you might un-comment to see a list of ports.

When testing, make sure the Serial Monitor of the Arduino IDE is NOT running! (close it).

First test with the example sketch **read_command_data_from_serial.ino**.

You should see the response of the Arduino in the Console (the actual commands might differ):



```
Jserinterface (3) [Java Application] C:\Program Files\Java\jre1.8.0_172\
setWalkable(11, 4, true)
setDestination(10, 9)
setWalkable(15, 7, true)
(8,3) -> (9,3) -> (10,3) -> (10,4) -> (10,5) -> (10,6
d90t90d180
Received: Command1: d90
Received: Command2: t90
Received: Command3: d180
```

Make Rover car respond to commands

Based on the previous test sketch, the original sketch for the car was extended with two methods **`readCommand()`**, **`doCommand()`** (based on showCommand()). When the GO button is pressed, it will process the instructions (starts carrying them out). See the if-statement in the **`loop()`**:

```
else if ( evshield.getButtonState(BTN_GO) ) { // start/stop
  Serial.println(F("GO"));
  dr_forward = !dr_forward;
  if (dr_forward) {
    if (instructions.length() > 0) { // if there are instructions, execute them
      readCommand();
      // make sure the wheels are straight after execution of commands
      straight();
    }
    else // otherwhise, just drive:
      drive();
  }
```

You can download the modified Rover sketch **`rover_with_comm_v2.ino`** here and test it**.**

Holding the instructions temporarily (in the variable *`instructions`*) until the Go button is pressed allows us to transfer the commands while connected via USB cable, disconnect the car, place it on the start tile (or any start-position) with the proper orientation, and hit ‘Go’.