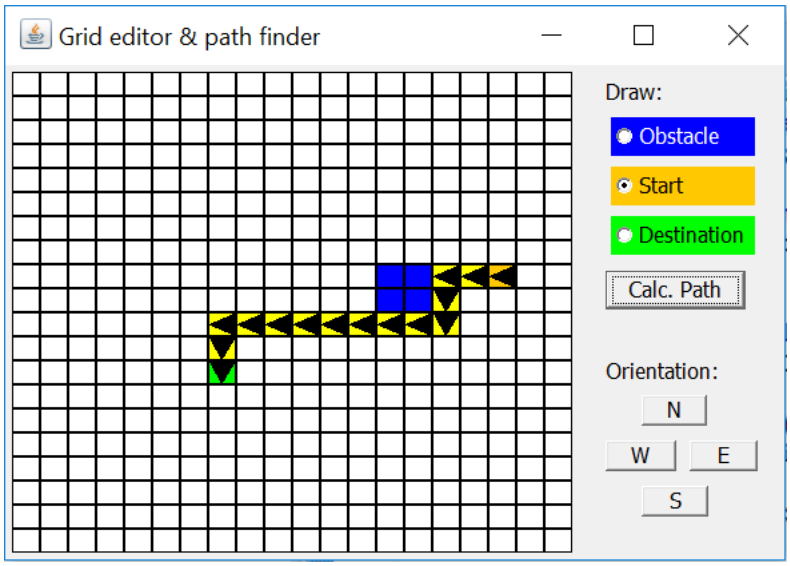


Assignment 7

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 Explorer robot build in previous assignments.

At the end of day 6 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 Explorer robot based on the path found. It will:

- Take into account the orientation of the robot: which direction is it facing?
- Calculate turns and distances to travel to generate driving instructions
- Show the robot walk 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 Explorer robot. The instructions will look like:

```
t90d100t-90d30t180d60
```

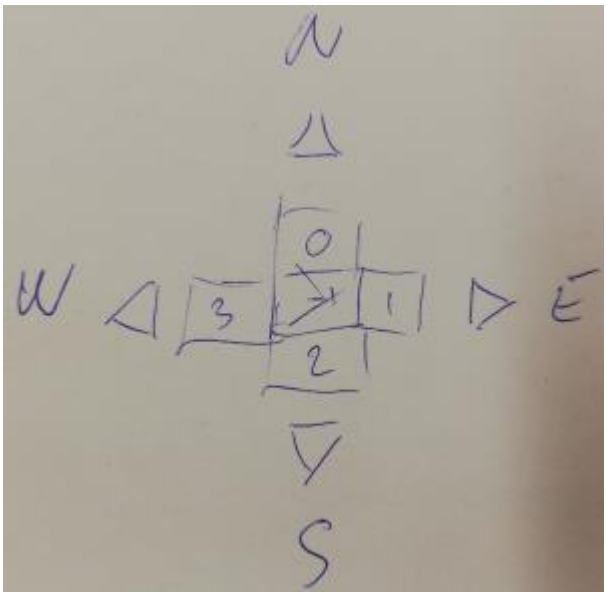
Where "t" means *turn* and "d" means *drive*. In human language these example instructions are:

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

Table of Contents

1. Show robot orientation in userinterface

Show the orientation of the robot, as a triangle. To accomplish this, change the drawing method of class PixelPanel. The class should already have a class-variable **orientation**. If not add it, it should be and integer with an initial value of -1 (= no orientation).



The values of class-variable **orientation** will be -1 or 0-3 for N, E, S, W. 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 *drawComponent()* method of class PixelPanel will draw a triangle pointing North.

To draw a triangle, use the example below:
(add it to the *drawComponent()* method)

```
g.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;

// Draw 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

Polygon p = new Polygon(x, y, n); // This polygon represents a triangle with the
                                  // above vertices
g.fillPolygon(p); // Draw the polygon filled
```

This entire piece of code only has to be executed if the orientation is >= 0. Add an if-statement for that.

Add another selection statement (*if* or *switch*) which uses the orientation to draw a triangle for each possible value of the orientation. If **orientation==0**, draw the north-facing triangle, if **orientation==1** the east-facing triangle, and so on. Only the assignment of the tree points is different for each triangle, so only that part should be in the selection statement for the orientation.

Next, in the *setStart()* method of the userinterface set the orientation of the panel to 1. 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).

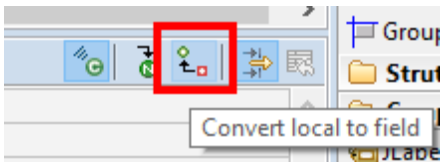
Test the application. When 'Start' is selected, and you click a tile, is the east-facing triangle drawn?

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

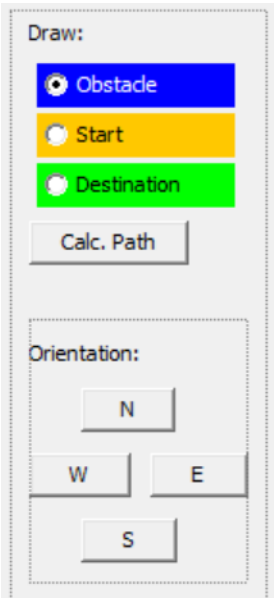
These buttons should only be visible when the "Start" radio button is selected. To accomplish this, put them in a panel, make it a field** and show/hide the panel on click of the radio buttons (this is explained later).
To hide the panel at startup, under Advanced properties, remove the check at field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

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

**



The panel with the buttons might look like:

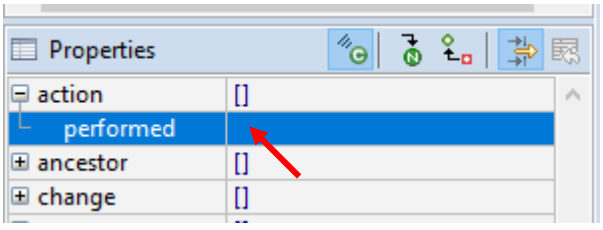


Next, add eventhandlers to the N-E-S-W buttons. Example of code for eventhandler of button N:

```
if (startX <0 || startY<0 ) return; // if no start set yet, do noting
pixelPanels[startX][startY].orientation = 0; // north = 0
pixelPanels[startX][startY].repaint();
```

Repeat this for the other buttons and change the value of the orientation accordingly.

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-Wbuttons visible:

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

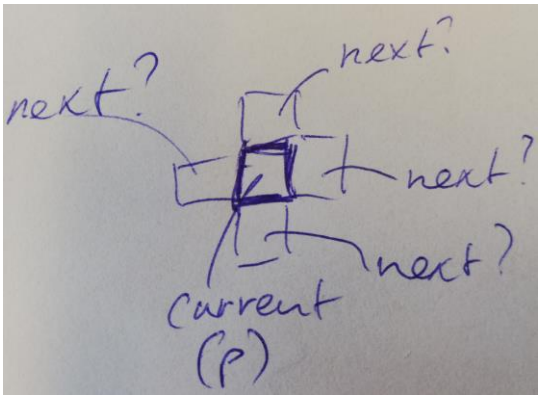
The other two buttons make the panel not-visible. (Your name of the panel that contains the N-E-S-Wbuttons might be different).

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 robot 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 inside the loop that shows the path, before the call to the *repaint()* 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 method that compares the orientation with two parameters, the current tile and the next tile.

Add a method to the class which will do the 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 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):

```
int turn = 0;
// where is the next tile?
if (p.x==next.x && p.y==next.y+1) { // N?
   if (p.orientation==0) turn = 0; // no turn necessary, already facing right direction
   else if (p.orientation==1) turn = -90;
   else if (p.orientation==2) turn = 180;
   else if (p.orientation==3) turn = 90;
   next.orientation = 0;
}
```

In this example we also determine the degrees the robot 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 variables turn and **`next.orientation`** will get different values!

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:

```
pixelPanels[startX][startY]
```

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 for options are handled correctly.

To show the orientation of the robot 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.getBackgroundColor()!=Color.GREEN)
```

3. Generate driving 'instructions'

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

```
drive(100); // drive one meter
turn(-90); // turn 90 degrees counterclockwise
turn(180); // turn 180 degrees
...
```

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 Explorer app (the Arduino app previously made).
'Commands' sent from the Path finding app to the Explorer app could look like:

```
d100
t-90
t180
```

or packed together:

```
d100t-90t180
```

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 static 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. Variable drive will be increased for each tile the robot drives: if the robot 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 have to be done are:

```
if no turn necessary (turn==0):
   increase drive by tileSizeCM
else: (a turn will be made)
   // before making a turn, save driving distance (if there):
   was a previous distance to drive 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 turn to the driving instructions // (add "t"+turn to the driving instructions)
```

Some parts are already written as comments, as these do not lead to code, but are meant to clarify things. Indented lines are part of if/else structure!

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 `"t"+turn`.
The pseudo-code "give drive a (start-)value of tileSizeCM" simply means: **`drive = tileSizeCM`**.

The generated driveInstructions 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 robot

To show the movements of the robot 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:

```
private Timer timer;
```

Use the Timer from 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 at the end of the method *showPath()*.

The timer has to be stopped if the counter is equal to the size of the path. Add an if-statement to *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 Explorer robot

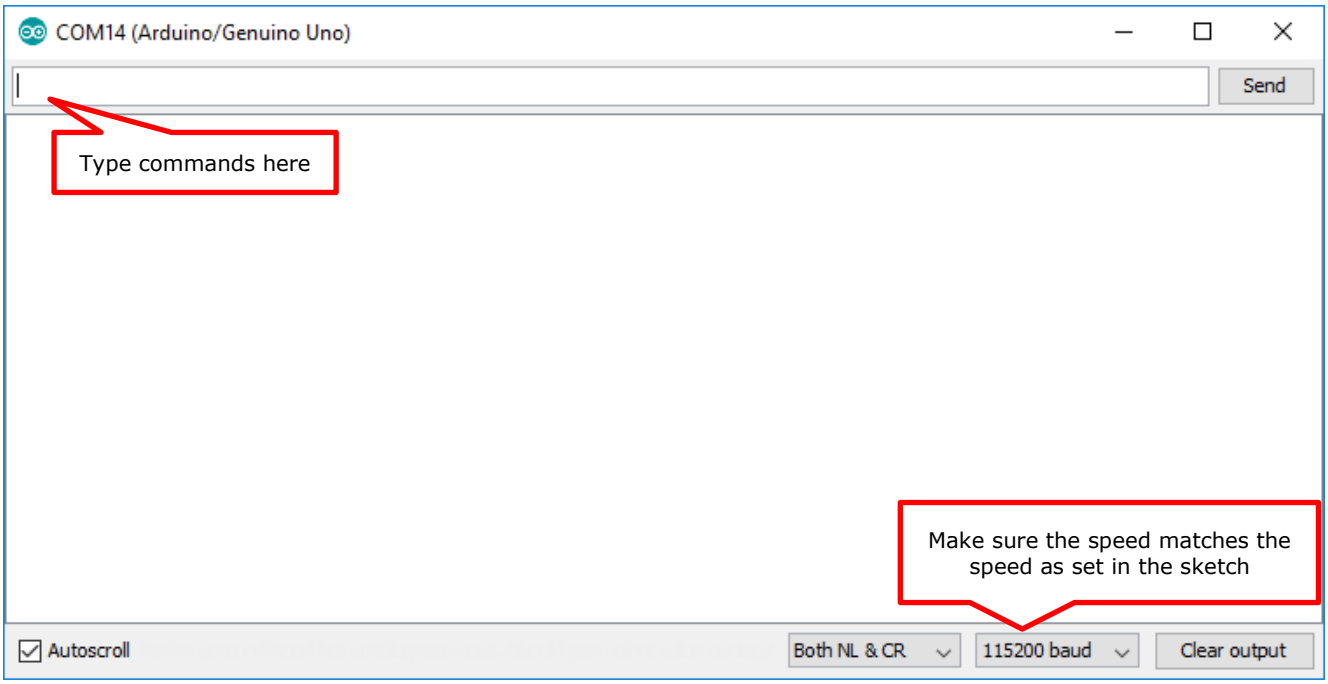
To send the driving instructions to the Explorer robot, both apps must communicate with each other. In past lectures other examples were given which use communication:

- 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 Explorer robot with your phone (using the Blynk App) was given in practical assignment 2.
- Further examples of communication were given in practical assignment 3 and these Blog posts.

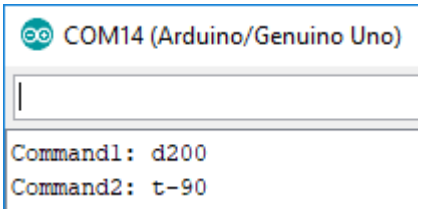
This appendix will show how to send the driving instructions from the Java app to the Arduino running the Explorer sketch via USB, with an addition to make it able to receive the instructions.

Create an Arduino app that can read the driving instructions

Test this example sketch. 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 "d200t-90", it should give:



Now all we must do is add this to the Arduino sketch of the Explorer robot, 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
```

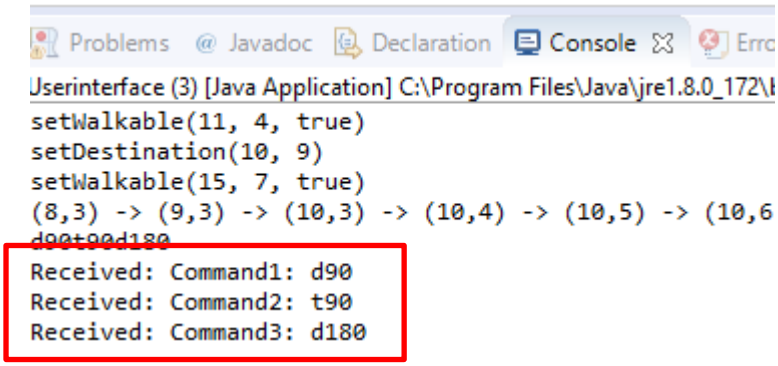
On a Mac, comports are looking different. It might be something like `/dev/tty.usbmodem*` or `/dev/tty.usbserial*`

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:



Make Explorer robot respond to commands

Based on the previous test sketch, the original Explorer sketch was extended with tree methods **`readCommand()`**, **`doCommand()`** (based on showCommand()) and **`setInstructions()`**. The last method will just ‘hold’ the instructions when they are received. Then, when the GO button is pressed, it will process the instructions (starts carrying them out). See the if-statement in **`checkSensors()`**:


```
else if ( evshield->getButtonState(BTN_GO) ) { // start/stop
  isDriving = !isDriving;
  if (isDriving) {
    if (instructions.length()>0) // if there are instructions, execute them
      readCommand(instructions);
    else // otherwhise, just drive:
      drive();
  }
  else {
    stop();
  }
```

You can download the modified Explorer sketch **explorer_with_comm.ino** here and test it**.**

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