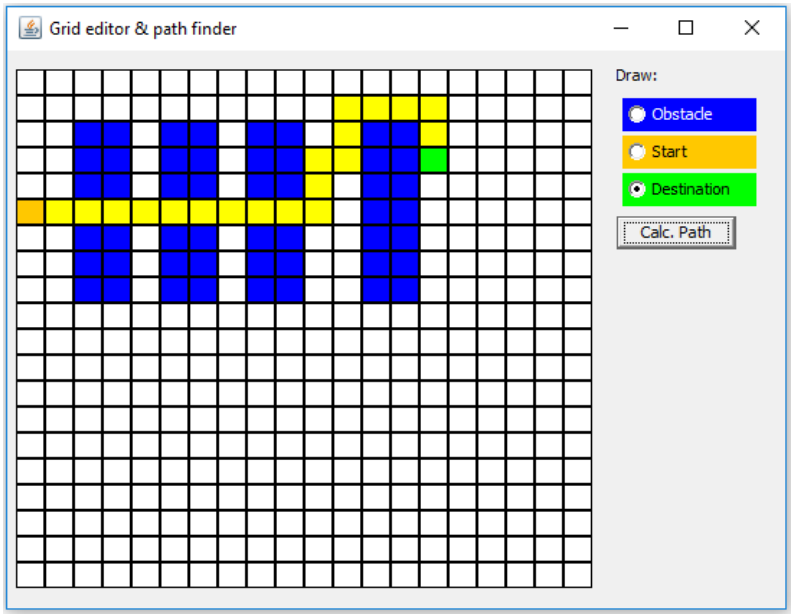


Assignment 6

Content Application Development Day 6
Lecture
In the lecture Inheritance, userinterfaces and algorithms were introduced. This assignment covers several skills learned in previous lectures and assignments, but this time with less detailed explanation.
Tutorial
In the tutorial you will practice building an interactive map editor with path-finding capabilities. It is for a large part based on examples found online.

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 6 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.

Create an interactive map editor with path-finding capabilities

In this tutorial-based assignment we are going to make an editor which can be used to create a map from a room, indicate a start and a destination (for a robot), and extend it with path-finding capabilities.

On the design of this application

Normally you would make a design for all elements of the application. To make this assignment not too complex, we resort to already existing solutions found on the internet and integrate these into a userinterface we build ourselves. This means a design for this application might be primarily a userinterface design: e.g. a sketch of the interface and the possible interactions.

In this assignment we program the dynamics of the application: the userinterface and the tiles, the application logic (responding to clicks, checking conditions) and finding the path and displaying it.

Online resources

To find and compare possible existing solutions to the 'problem', some searches were done on the internet. Some of the search terms used are (you can also omit the word 'java' for more general solutions):

```
path finding java
a star algorithm java
grid generate room map java
interactive edit map java
mouse pixel draw panel java
```

These led to these two (partial) solutions:

- For the path-finding problem we found a Java version of the A* algorithm. The java files for this solution are in the folder "pathfinding". Unfortunately, the original website is no longer available.
- For a tile-based map editor we found this solution (at the bottom of the page, post by "Gilbert Le Blanc"), the PixelPainter class and it's helper classes which you can find in the folder "userinterface".

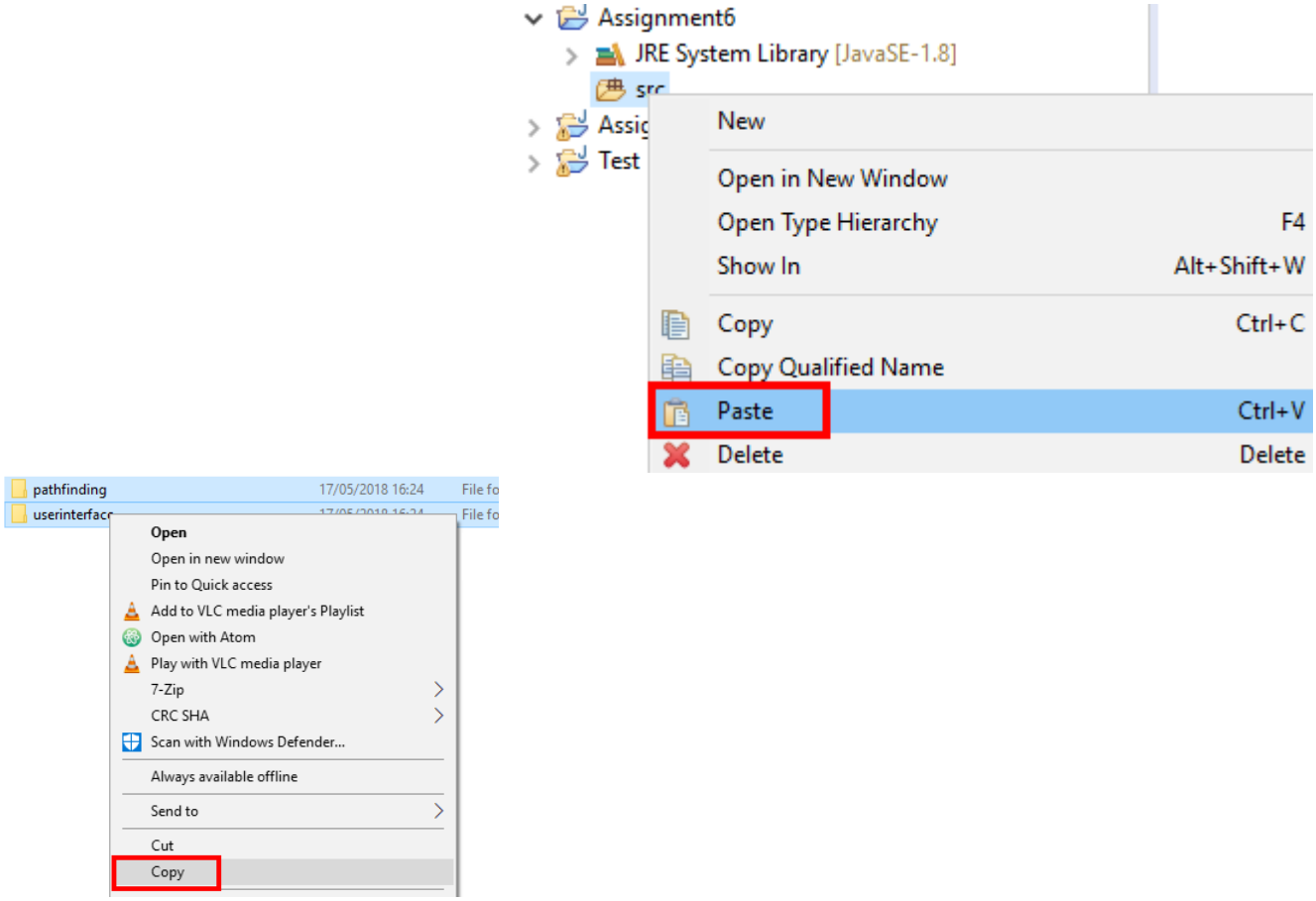
These two solutions are available as folders in the zip-file of this assignment and can be used as packages in the Eclipse project of this assignment. In this assignment we will build a userinterface based on the PixelPainter class and using the path-finding solution.

Table of Contents

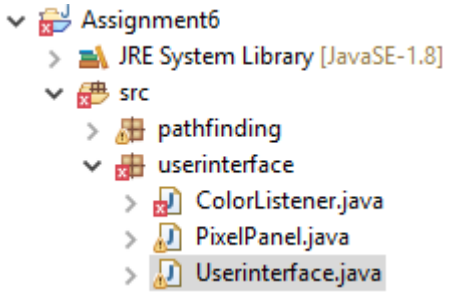
1. Create project & user interface

Create a new project.

Copy the two folders from the zip-file and paste them into the **src** folder of the project:

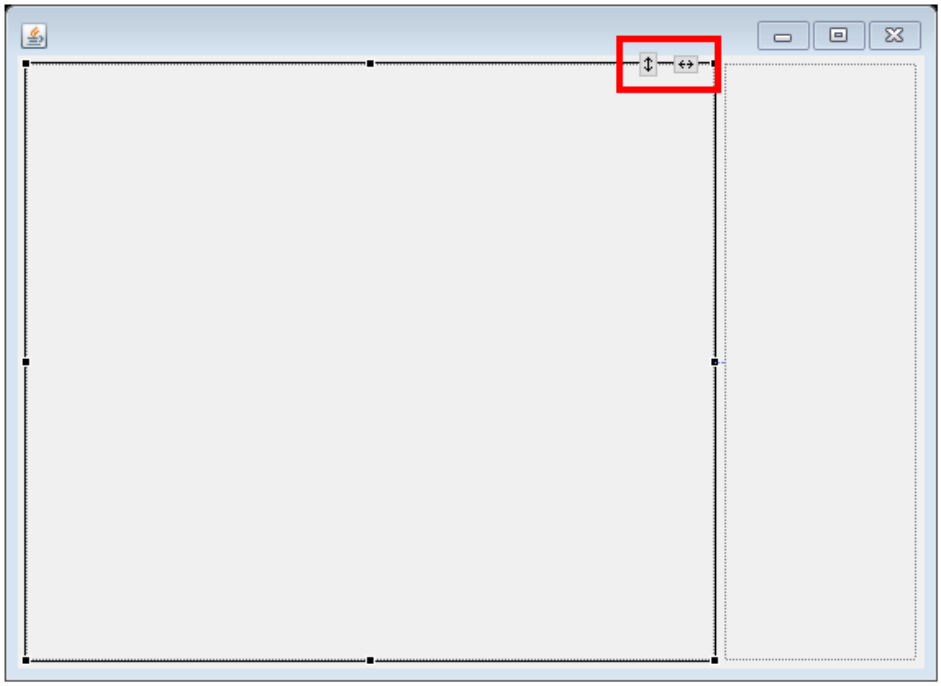


There will be some errors in the ColorListener class, which you can ignore for now. In the package `userinterface`, add a `JFrame` for the userinterface, name it **`Userinterface`**. The project should now look like:

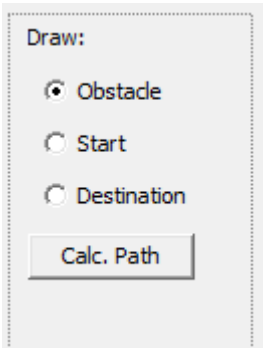


Create userinterface

Set the layout of the contentPane to GroupLayout. Add two JPanels and make them both auto resizable:

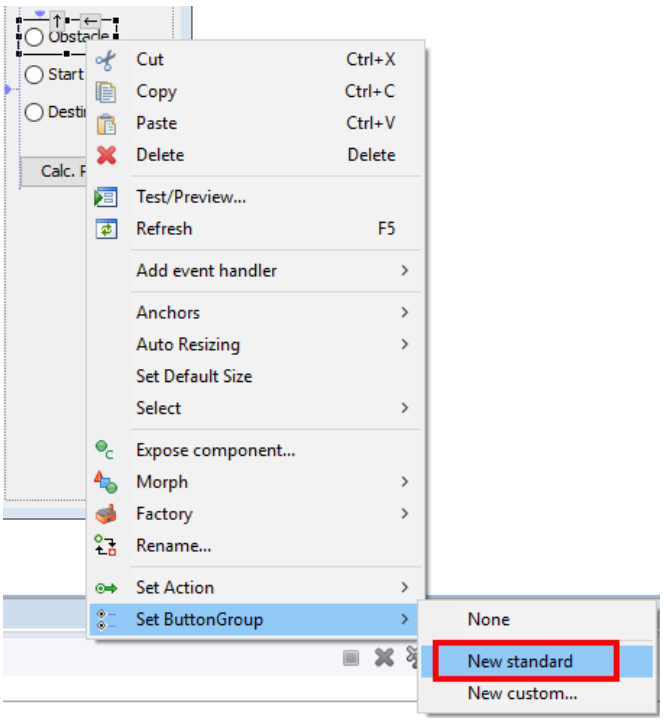


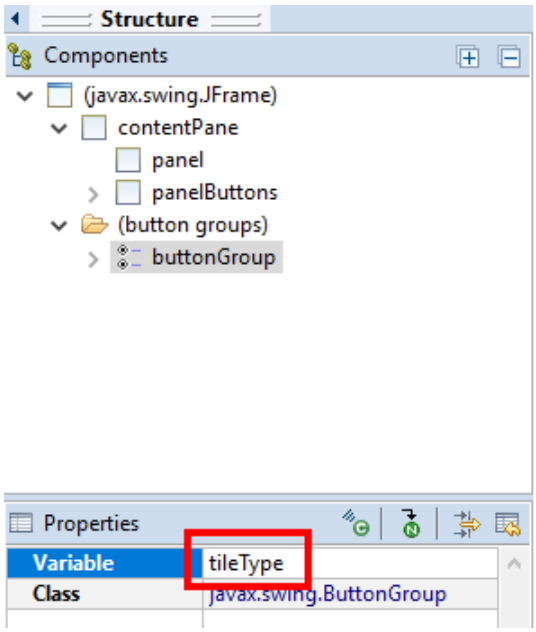
If the left panel does not have the name 'panel', rename it to 'panel'. Set the layout of the left panel to GridLayout, and the right panel to GroupLayout.



Add a label, 3 radio buttons and a button to the right panel. Make sure the titles of the radio buttons are 'Obstacle', 'Start' and 'Destination'. Set the first radio button as selected.

Make the 3 radio buttons member of the same button group, name this group 'tileType': first right-click one of the radio buttons and choose *Set ButtonGroup > New standard*, then change its variable name to tileType:





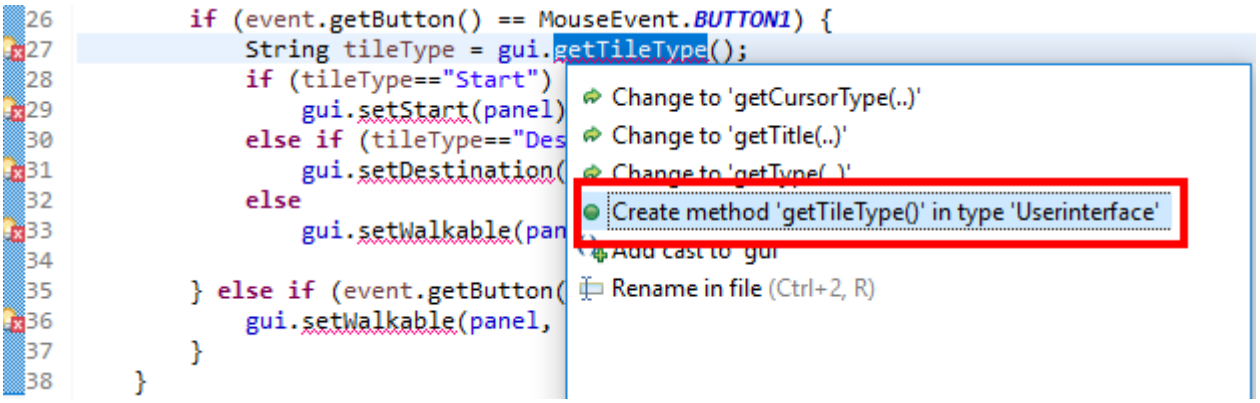
Make the other 2 radio buttons member of the same group by right-clicking them and choose *Set ButtonGroup > tileType*.

To make the relation with the colored tiles of the map stronger, you could also change the background color of the 3 radio buttons to blue, orange and green (see color-table in section 3 also).

Save the project (*File* > *Save All*).

2. Add methods to the Userinterface class

We will first add the missing methods which cause the errors in the ColorListener class. Open ColorListener.java. For each error, select the option to add a method to the Userinterface class:



We also need a method which will calculate the path from start to destination. Think of a name for this method and add it to the Userinterface class. Call the method from the event handler of the "Calc. Path" button. Add a line `System.out.println("Calculate path...");` to that method and test if pressing the button works (you should see the message in the Console).

Add similar print statements the other methods of the Userinterface class (you can just print the name of the method).

3. Integrate the PixelPainter class

We will use a two-dimensional array of PixelPainter objects to display the editor. We need to keep track of each panel to be able to for instance change its color (if for example we need to draw the path of the robot).

A two-dimensional array of PixelPainter objects can be declared as follows:

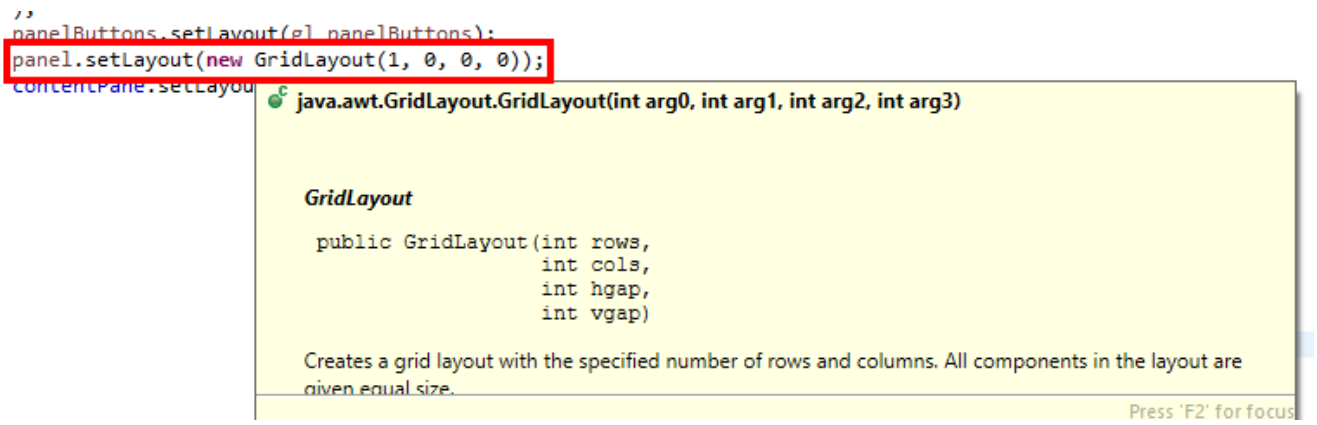
```
int width = 20, height = 20;
PixelPanel[][] pixelPanels = new PixelPanel[width][height];
```

Add these as class-variables to the Userinterface class.

To initialize the array of pixelPanels, add this code to the constructor (place it at the end!):

```
// initialize the two-dimensional array of PixelPainter objects:
for (int y = 0; y < height; y++) {
	for (int x = 0; x < width; x++) {
		PixelPanel p = new PixelPanel(x,y);
		pixelPanels[x][y] = p;
		p.addMouseListener(new ColorListener(p, this));
		panel.add(p);
	}
}
```

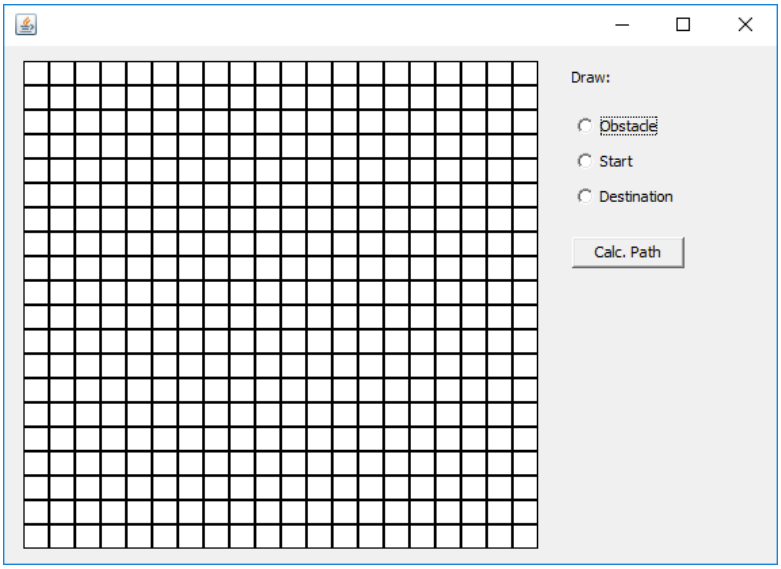
To make the panel look proper, we must set the gridLayout pattern to display as a grid using the width and height. Find the line of code that sets the layout of panel to GridLayout:



Change that to:

```
panel.setLayout(new GridLayout(height, width, 0, 0));
```

Run the application. The map should be drawn like this:



Clicking the tiles in the grid should print the names of the methods that are called in the Console (if you added the print statements in step 2).

To check if the radio buttons work, we will add code to the method *getTileType()*. This method should get the text of the currently selected radio button (which is used to select the tile type). If you Google this: "get text current selected radio button java". You will end up here. The answer from 'Rendicahya' contains a workable solution inside the method *getSelectedButtonText()*. The code used in that method is:

```
for (Enumeration<AbstractButton> buttons = tileType.getElements(); buttons.hasMoreElements();)
{
    AbstractButton button = buttons.nextElement();

    if (button.isSelected()) {
        return button.getText();
    }
}
```


Copy this into the method *getTileType()*. This will result in some errors, you can fix the imports yourself. Check if the name of your ButtonGroup is **`tileType`**. If not, change it to the name you used.

Run the application, click a tile, select another radio button, click a tile. If for instance "Start" is selected, it should show the name of the *setStart()* method when you click a tile.

Change color of the tiles
We will use the following colors for tiles to indicate what the meaning of a tile is:

Tile color	Means
White	Walkable space
Blue	Obstacle (not walkable space)
Orange	Start position of the robot (can be only one tile)
Green	Destination of the robot (for now, can be only one tile)
Yellow	Path of the robot (from start to destination

If a tile is clicked while 'obstacle' is selected, it must turn blue (it has become an obstacle). This will result in a method call of `setWalkable(panel, false)`: the second parameter will be `false`.

So in method *setWalkable()* we have to check this parameter. The method now looks like:

```
public void setWalkable(PixelPanel panel, boolean walkable) {
    System.out.println("setWalkable()");
}
```

It might be a good idea to rename your parameters of this method, if they have different names.

In method *setWalkable()*, insert an if-statement, which checks the second parameter and changes the background color accordingly:

```
if (walkable)
   panel.setBackgroundColor(Color.WHITE);
else
   panel.setBackgroundColor(Color.BLUE);

panel.repaint();
```

Check if it works. Clicking a tile with the left mouse button should make it blue, and with the right button should turn it white again.

Now in the methods *setStart()* and *setDestination()* add code that turns the tiles orange and green. (a call to `panel.setBackgroundColor()` and `panel.repaint();` is sufficient)

The basic tile editor works now, but it is possible to add multiple start and destination tiles.

Have one start and destination tile
The robot can have only one start position and (for now) there also can be only one destination. We must remember these positions, and if for instance a start has already been set, and another is clicked, we should erase the previous (that's the most user-friendly solution).

To remember something, we can add it as a class-variable. To remember the start and destination tiles, we can add variables for these. A tile is a PixelPanel. A possible way to use these as class-variables could be:

```
PixelPanel start, destination;
```

Now we can check if a start or destination was already set in the methods setStart() and setDestination(). We take the setStart() as an example, add this code to that:

```
// if a previous start was set, clear it:
if (start!=null)
    setWalkable(start, true);
// store start position:
start = panel;
```

If an object reference had not been set or initialized yet, it will have the value **null**. More info.

Do you understand what this does? Add a similar check to the setDestination() method and store the destination position also.

Check if now only one start or destination can be set.

There is one more case we must consider: if 'Obstacle' is selected, and you click on a tile which is already set as start or destination, the start or destination must be cleared. So in setWalkable(), check if the clicked tile is start or destination. We show it for the start:

```
// check if clicked panel is start:
if (panel==start) start = null; // clear start
```

Add a similar check for the destination.

Similar checks should also be added to setStart() and setDestination(), to prevent eg. setting the destination on the start or vise-versa.

4. Integrate path-finding code

To learn how the path-finding code works, we can look at the example code in `ExampleUsage.java` (it is in the package 'pathfinding').

The first line in the *main()* method creates a new Map for the path-finding engine:

```
public static void main(String[] args) {
    Map<ExampleNode> myMap = new Map<ExampleNode>(50, 50, new ExampleFactory());
```

We will use this example to add a map to the Userinterface class. Instead of the fixed size `50, 50` we use the width and height variables of the Userinterface class.

Add the map as a class-variable:

```
Map<ExampleNode> map;
```

(import both `Map` and `ExampleNode` from the *pathfinding* package).

And initialize it at the end of the constructor:

```
map = new Map<ExampleNode>(width, height, new ExampleFactory());
```

As we look further at the example code, we see a setting:

```
// setting:
myMap.CANMOVEDIAGONALY = false;
```


Which prevents diagonal movement. We will use this also, so add it to the code at the end of the constructor (remember we use variable `map` instead of `myMap`). Later on, if you for instance want to adapt this to a particular type of car/robot, you can change this setting.

The next line shows how to add obstacles:

```
// add obstacle:
myMap.setWalkable(3, 3, false);
```

This is something we can use in the *setWalkable()* method of our Userinterface class. Add that line to that method. To get the position (`3, 3` in example), use `panel.x, panel.y`. The third parameter (`false` in example), is in this case the variable `walkable` (second parameter of the method).

The same method call should be made in the methods *setStart()* and *setDestination()* also, as start and destination are walkable parts of the path. The third parameter must be **`true`** in these calls.

If we look further in the example, we see the code which will find the path:

```
// find path from 0,0 to 40,40
List<ExampleNode> path = myMap.findPath(0, 0, 40, 40);
```

Just like the map variable, add the variable **`path`** as a class variable. Then put the initialization part in the method which calculates the path (which was added in step 2). The parameters are the start and destination position (**`start.x, start.y, destination.x, destination.y`**).

An example of this was given in the presentation (of lecture 6).

The last lines in the example print the path to the Console:

```
for (int i = 0; i < path.size(); i++) {
    System.out.print("(" + path.get(i).getxPosition() + ", " + path.get(i).getyPosition() + ") -> ");
}
System.out.println("");
```

You can copy it to the method which calculates the path.

Inside this for-loop we can add code which shows the path in the panel. To change the background color of one panel inside the grid of panels, we need to access it. That can be done via the `pixelPanels` array:

```
PixelPanel p = pixelPanels[path.get(i).getxPosition()][path.get(i).getyPosition()];
```

Add this line of code inside the for-loop and add a call to the method *setBackgroundColor()* of object `p`, to change the background color of the panel to yellow.

Error prevention

The method which calculates the path should do some error prevention before it calculates the path. In some cases, the method needs to be aborted. This can be done with a return statement like this:

```
if (...) { System.err.println("Error message"); return; }
```

Warnings and errors should be sent to the standard error output (in the Console) like this:

```
System.err.println("This is an error.");
```

The following cases should be considered:

- If the start or destination position is not set (values are **`null`**), or if start and destination are the same, the pathfinding should not start. Add 3 if-statements to check this.
- If after finding the path, the size of the path is 0, no path could be found. Add a warning which deals with this situation.
- This is not really an error, but a convenience: for the destination to remain visible, prevent it from being overwritten by the path by changing the end-condition of the for-loop to `path.size()-1` (instead of `path.size()`)

Clearing a previous path

If the user changes the grid, eg. turns a part of the path into an obstacle, and hits the button to calculate the path again, the previous path should be cleared. You can check if a previous path was there with:

```
if (path != null) { ... }
```

An example of this was given in the presentation (of lecture 6).

Inside this if-statement, add a for-loop that clears each panel of the path (sets background color to white). Hint: this for-loop is almost the same as the other for-loop. Another hint: only panels with a yellow background should be cleared (these are still visible as parts of the path).

Finish

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.

