Defensive Programming

Generally, we aimed for a robust program, so our way of defensive programming reflected this by always setting default values and catching exceptions, that even when they meant an extreme failure, set a default state and continued working.

/g2017s_se2_0303/src/swingfrontend/action/LoadFileAction.java

```
switch (n){
    case JOptionPane.CANCEL_OPTION:
        System.out.println("option chosen: Canel" + n);
        break;
    case JOptionPane.CLOSED_OPTION:
        System.out.println("option chosen: close" + n);
        break;
    case JOptionPane.YES_OPTION:
        System.out.println("option chosen: yes" + n);

        swingFrame.saveFile();
        swingFrame.loadFile();
        break;
    case JOptionPane.NO_OPTION:
        System.out.println("option chosen: no" + n);
        swingFrame.loadFile();
        break;
    default:
        System.out.print("option chosen: default- error abort, launch all nukes and
abandon ship! " + n);
        break;
}
```

By using Enumerations, we reduced the risk of errors in flexible situations such as with factories, strategy patterns or cell types.

/g2017s_se2_0303/src/cells/ECellType.java

```
public enum ECellType { TEXT, NUMBER, FORMULA }
```

The cited ECellType is used when checking for the type of a cell, making it clear-cut as to what is what.

By establishing the practice of returning null, we opened the risk of operating on null values, but by adhering to the convention of checking for null after every fetch operation we created a very robust convention that works for us.

/g2017s_se2_0303/src/charts/BarChart.java

```
int j = 0;
for (int i = 1; i < rows; i++) {
    double value = Double.parseDouble(mModel.getValueAt(i, 2).toString());
    if (mCities.get(j) != null) {
        dataset.addValue(value, mCities.get(j), area);
    }
    j++;
}
```