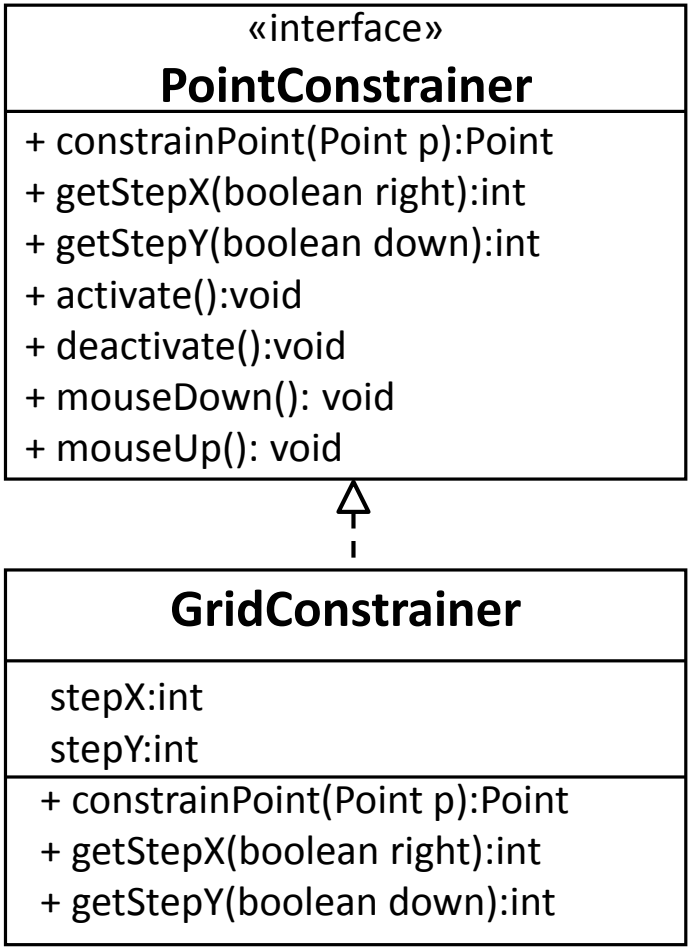
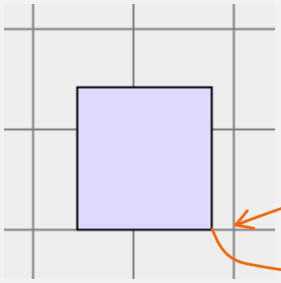
Assignment 5 : Snap to Grid

- **Snap To Grid**
- **Snap To Nearest**



Snap to Grid

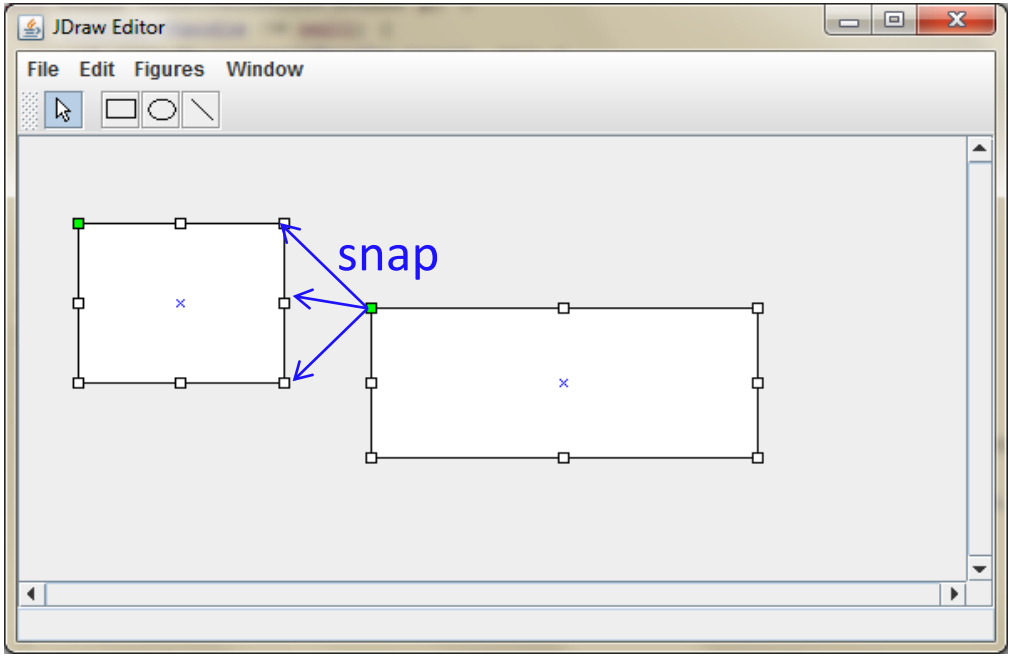
```
public class GridConstrainer implements PointConstrainer {
   private final int stepX, stepY;
   public GridConstrainer(int sx, int sy) {
      this.stepX = Math.max(1, sx);
      this.stepY = Math.max(1, sy);
   }
   @Override public Point constrainPoint(Point p) {
      int x = ((p.x+stepX/2) / stepX) * stepX;
      int y = ((p.y+stepY/2) / stepY) * stepY;
      return new Point(x, y);
   }
   @Override public int getStepX(boolean right) { return stepX; }
   @Override public int getStepY(boolean down) { return stepY; }
   @Override public void activate() {}
   @Override public void deactivate() {}
   @Override public void mouseDown() {}
   @Override public void mouseUp() {}
}
```



p.x= 128
128+10/2 = 133
133/10 = 13
13*10 = 130

Snap to Nearest Object

- **Idea**
  - Snap to the positions of near-by handles instead of to fixed grid points



- **3 Cases**
  - Existing figure is changed over the handles: Handle.dragInteraction()
  - New figure is defined: Tool.dragInteraction()
  - Figure (or whole selection) is moved: move(…)

Case 1: Existing figure is changed over handles

```
public class SnapGrid implements PointConstrainer {
   private static final int SNAP = 15;
   private final DrawView view;
   public SnapGrid (DrawView view) { this.view = view; }

   public Point constrainPoint(Point p) {
      for (Figure f : view.getModel().getFigures()) {
         if (!view.getSelection().contains(f))
            for (FigureHandle h : f.getHandles()) {
               Point pos = h.getLocation();
               if (nearBy(p, pos)) { return pos; }
      }     }
      return p;
   }

   private boolean nearBy(Point p, Point q) {
      return p.distance(q) < SNAP;
   }
```

All figures in the model, except the figures in the selection

Check handles of each figure. If a handle is near-by, then its position is returned

Case 2: New figure is created over tool

- **Problem**
  - New figure is not yet in the selection, so it "snaps" to itself
    => We have to exclude the created figure (after it has been included)
    => Register an observer in the model (in ctor, activate or mouseDown)

```
private DrawModelListener listener;
private Figure figure;

public SnapGrid(DrawView view) {
   this.view = view;
   listener = e -> {
      if (e.getType() == DrawModelEvent.Type.FIGURE_ADDED) {
         figure = e.getFigure();
      }
   };
   view.getModel().addModelChangeListener(listener);
}
```

Case 2: New figure is created over tool

- **Problem**
  - New figure is not yet in the selection, so it "snaps" to itself
    => We have to exclude the created figure
    => Register an observer in the model (in ctor, activate or mouseDown)

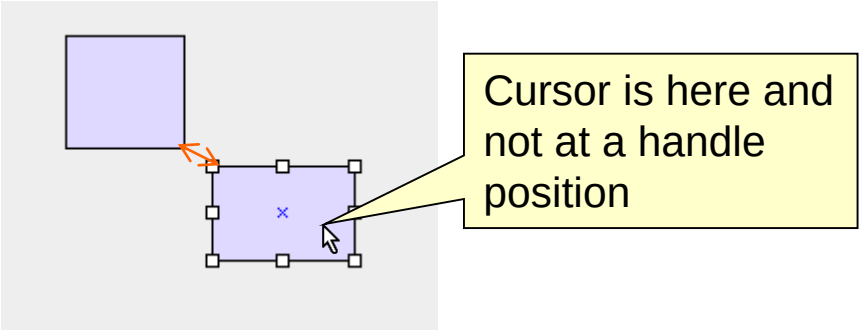
```
public Point constrainPoint(Point p) {
  for (Figure f : view.getModel().getFigures()) {
    if (!view.getSelection().contains(f) && f != figure) {
      ...
```

Only snap to figures which are not selected

Only snap to figures which are not being created

Case 3: Figures are moved with the mouse

- **Problem**
  - The coordinates of the mouse are not at a handle position but anywhere inside the figure



  - Idea: compare all handles of the selected figures with the handles of the other figures
    - if such a delta is small enough, then add this delta to the current mouse position
    - Problem: on the next call of constrainPoint, the delta is zero and the mouse coordinates are not changed, i.e. the figure jumps back

Case 3: findNearHandleOf

```
private FigureHandle findNearHandleOf(FigureHandle h) {
   for (Figure f : view.getModel().getFigures()) {
      if (!view.getSelection().contains(f)) {
         for (FigureHandle nh : f.getHandles()) {
            if (nearBy(h.getLocation(), nh.getLocation())) {
               return nh;
            }
         }
      }
   }
   return null;
}
```

Finds a handle near-by to handle h and returns it

Iterates over all figures which are not selected

- Looks for a handle which is near-by the handle h which is passed as an argument to method `findNearHandleOf`
- If no such handle is found, null is returned

Case 3: constrainPoint

```
private Point p0; // snapped/modified mouse coordinates
private boolean snapped = false;
public Point constrainPoint(Point p) {
   if (snapped) {
      if (nearBy(p0, p)) { return p0; }
      else { snapped = false; return p; }
   }
   for (Figure s : view.getSelection()) {
      for (FigureHandle h : s.getHandles()) {
         FigureHandle nearHandle = findNearHandleOf(h);
         if (nearHandle != null) { snapped = true;
            int dx = nearHandle.getLocation().x-h.getLocation().x;
            int dy = nearHandle.getLocation().y-h.getLocation().y;
            return p0 = new Point(p.x + dx, p.y + dy);
         }
   }}
   return p;
}
```

Prevents jumping back after a snap to a handle

Iterates over all figures which are moved (does not work for new figures)

If a near-by handle is found, then p0 is moved by the corresponding delta and the new point is stored in p0

Remarks

- **Is PointConstrainer interface powerful enough?**
  - No, access to the model and to the selection is missing
    - Missing references can be passed to a concrete constrainer over its constructor
  - No, kind of operation is not known in the constrainer
    - If a new figure is created, it is not known which is this figure, a possible workaround is to register a listener
- **Is PointConstrainer an application of State or Strategy?**

PointConstrainer: Strategy vs State

- **Strategy**
  - constrainPoint is a "compute" method, an algorithm how the coordinates are mapped
  - Strategy is set externally, not by the constrainer itself
  - Constrainer does not know about other implementations
  - Contains algorithm-specific state (like snapped, p0), i.e. it is a stateful strategy (and state is stateless)
- **State**
  - Constrainer can be changed at run-time
  - Constrainer has several event-methods (`getStepX`, `getStepY` and `constrainPoint`)
  - Point constrainer can be exchanged at run-time (and therefore has `activate` and `passivate` methods)
  - PointConstrainer represents grid-state of the draw view
  - PointConstrainer defines state-specific behavior of the view