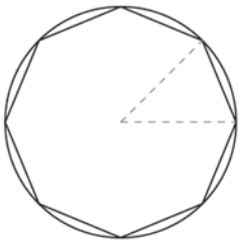
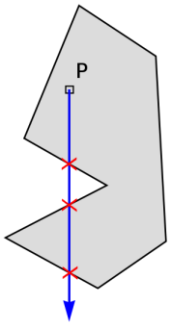
Tutorial #3. Lists

1) We need to create a polygon representation in Java. We will use **Cartesian coordinates** to represent the point, and **list** to aggregate them. Each point stands for **vertex**. Each two neighbor points represent the **edge** (+ one edge is between first and last points). Your task is to **generate polygon** approximating some complex curve. **Write a loop** that will calculate coordinates of polygon vertices. You may generate this data using any well know equation: circle, Lemniscate of Bernoulli or cardioid. Good way is to use parameter *t = 0..N-1* and polar coordinates equation for each Θ = 2 π t / N. Then find (x, y) = (ρ*cos Θ, ρ*sin Θ).



2) **Implement single-linked list as a class**. Your class should be able at least to *insert*, *delete*, *get* and *update* value *by index*.
   a) (*) Implement *Iterable* interface for your structure and *Iterator* class.
3) **Implement crossing number algorithms for "point in polygon" problem**.



Short explanation: for point of interest we build a random ray. If this ray intersects polygon even number of times (0, 2, 4, ...) – point is outside the polygon, else point is inside. Your implementation should be something like

```
static boolean inside(YourListType polygon, Point2D point) { ... }.
```

a) *(*)Use **foreach** loop to iterate over the list if [2a] is done.*
b) *(*) Propose your own **method for intersection check**. Else use the following implementation (checks if **ab** intersects with **cd** using javafx.geometry.Point2D).*

```
public static boolean intersects(Point2D a, Point2D b, Point2D c, Point2D d) {

        // We describe the section AB as A+(B-A)*u and CD as C+(D-C)*v
        // then we solve A + (B-A)*u = C + (D-C)*v
        // let's use Kramer's rule to solve the task (Ax = B) were x = (u, v)^T

        // build a matrix for the equation
        double[][] A = new double[2][2];
        A[0][0] = b.getX() - a.getX();
        A[1][0] = b.getY() - a.getY();
        A[0][1] = c.getX() - d.getX();
        A[1][1] = c.getY() - d.getY();

        // calculate determinant
        double det0 = A[0][0] * A[1][1] - A[1][0] * A[0][1];

        // substitute columns and calculate determinants
        double detU = (c.getX() - a.getX()) * A[1][1] - (c.getY() - a.getY()) * A[0][1];
        double detV = A[0][0] * (c.getY() - a.getY()) - A[1][0] * (c.getX() - a.getX());

        // calculate the solution
        // even if det0 == 0 (they are parallel) this will return NaN and comparison will fail -> false
        double u = detU / det0;
        double v = detV / det0;
        return u > 0 && u < 1 && v > 0 && v < 1;
}
```

   c) **Test** that your method works right for sections:
      i. (0,0)-(10, 10); (10, 0)-(0,10) - true
      ii. (0, 0)-(10, 10); (1, 0)-(11, 10) - false
      iii. (0, 0)-(10, 10); (1, 0)-(50, 10) - false
   d) **Run tests of your algorithm using generated polygon data.**
   e) (*) Use Monte Carlo method of integration to calculate the area of the polygon.