Aufgabe 1

```
public class IWalkerSpace extends WalkerSpace {
  // ATTRIBUTES
  public enum Direction {NORTH, EAST, WEST, SOUTH, NORTHWEST, NORTHEAST, SOUTHWEST, SOUTHEAST};

  // CONSTRUCTOR
  public IWalkerSpace(int n) {
    super(n);
  }

  // METHODS
  public void calculateFreeDirections(Walker walker){
  // calculate the list of free directions a walker can take
    walker.currentFreeDirs.clear();
    if (walker.alive){
      if (isEastFree(walker)) walker.currentFreeDirs.add(Direction.EAST.ordinal());
      if (isWestFree(walker)) walker.currentFreeDirs.add(Direction.WEST.ordinal());
      if (isNorthFree(walker)) walker.currentFreeDirs.add(Direction.NORTH.ordinal());
      if (isSouthFree(walker)) walker.currentFreeDirs.add(Direction.SOUTH.ordinal());
      if (isNorthFree(walker)) walker.currentFreeDirs.add(Direction.NORTHWEST.ordinal());
      if (isSouthFree(walker)) walker.currentFreeDirs.add(Direction.SOUTHWEST.ordinal());
      if (isNorthFree(walker)) walker.currentFreeDirs.add(Direction.NORTHEAST.ordinal());
      if (isSouthFree(walker)) walker.currentFreeDirs.add(Direction.SOUTHEAST.ordinal());
    }
    if (walker.currentFreeDirs.isEmpty()){
      walker.setDead();
      space[walker.x][walker.y].markWith(walker);
    }
  }
  public void moveWalkers(){
  // try to move all walkers one step
    for ( Walker walker : walkers){
      if (walker.alive){
        int d = rand.nextInt(walker.currentFreeDirs.size());
        Direction dir = Direction.values()[walker.currentFreeDirs.get(d)];
        switch (dir){
          case NORTH: moveNorth(walker); break;
          case WEST:  moveWest(walker); break;
          case SOUTH: moveSouth(walker); break;
          case EAST:  moveEast(walker); break;
          case NORTHWEST: moveNorthWest(walker); break;
          case SOUTHWEST: moveSouthWest(walker); break;
          case NORTHEAST: moveNorthEast(walker); break;
          case SOUTHEAST: moveSouthEast(walker); break;
        }
      }
    }
  }
  public void moveNorthEast(Walker walker){
  // if the left position is free a walker go to the left
    if (isNorthEastFree(walker)){
      --walker.x;
      --walker.y;
      markPositionWith(walker);
    }
  }
  public void moveNorthWest(Walker walker){
  // a walker make a step to the right
    if (isNorthWestFree(walker)){
      ++walker.x;
      --walker.y;
```

```
   markPositionWith(walker);
  }
 }
 public void moveSouthEast(Walker walker){
 // if the left position is free a walker go to the left
  if (isSouthEastFree(walker)){
   --walker.x;
   ++walker.y;
   markPositionWith(walker);
  }
 }
 public void moveSouthWest(Walker walker){
 // a walker make a step to the right
  if (isSouthWestFree(walker)){
   ++walker.x;
   ++walker.y;
   markPositionWith(walker);
  }
 }
 public boolean isNorthEastFree(Walker walker){
  int x = walker.x;
  int y = walker.y;
  return (x>xMin && y>yMin && space[x-1][y-1].color == bgColor);
 }
 public boolean isNorthWestFree(Walker walker){
  int x = walker.x;
  int y = walker.y;
  return (x<xMax && y>yMin && space[x+1][y-1].color == bgColor);
 }
 public boolean isSouthEastFree(Walker walker){
  int x = walker.x;
  int y = walker.y;
  return (x>xMin && y<yMax && space[x-1][y+1].color == bgColor);
 }
 public boolean isSouthWestFree(Walker walker){
  int x = walker.x;
  int y = walker.y;
  return (x<xMax && y<yMax && space[x+1][y+1].color == bgColor);
 }
 public float getMeanWalkingDistance() {
  float meanWalkingDistance = 0;
  int numOfWalkers = 0;
  for (Walker walker : walkers){
   meanWalkingDistance += walker.steps;
   numOfWalkers ++;
  }
  return meanWalkingDistance/numOfWalkers;
 }
}
```

Aufgabe 2

```
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Polygon;
import java.util.Random;

abstract class AbstractShape implements Shape, Animation {
  // ATTRIBUTES
  double radius;
  Point center = new Point();
  Color color = Color.lightGray;
  ShapesWorld world;
  double velocity;
  int steps = 0;
  Random rand = new Random();

  public void setShapesWorld(ShapesWorld theWorld){
    this.world = theWorld;
  }

  // GET METHODS
  public Point getCenter() { return center; }
  public double getRadius() { return radius; }
  public Color getColor(){ return color; }

  // FILL METHODS
  public static void fillCircle(Graphics g, double x, double y, double r){
    g.fillOval((int) (x-r/2),(int) (y-r/2),(int) r,(int) r);
  }
  public static void outlineCircle(Graphics g, double x, double y, double r){
    g.drawOval((int) (x-r/2),(int) (y-r/2),(int) r,(int) r);
  }
  public static void fillNtagon(Graphics g, double x, double y, double r, int n){
    if( n == 0 ){
      fillCircle(g, x, y, r);
    } else {
      Polygon p = getNtagon(g, x, y, r, n);
      g.fillPolygon(p);
    }
  }
  public static void outlineNtagon(Graphics g, double x, double y, double r, int n){
    Polygon p = getNtagon(g, x, y, r, n);
    g.drawPolygon(p);
  }
  public static Polygon getNtagon(Graphics g, double x, double y, double r, int n){
    int[] x_coords = new int[n];
    int[] y_coords = new int[n];
    double deg = 360/n;
    for( int i=0; i<n; i++ ) {
      y_coords[i] = (int) (y+r*Math.sin(Math.toRadians(i*deg-90)));
      x_coords[i] = (int) (x+r*Math.cos(Math.toRadians(i*deg-90)));
    }
    return new Polygon(x_coords, y_coords, n);
  }

  // METHODS
  public boolean contains(double x, double y) {
    if( x<(center.x-radius) || x>center.x+radius ||
            y<(center.y-radius) || y>(center.y+radius) ){
      return false;
    } else {
      return true;
    }
  }
  public void destroy(){
    this.world.removeShape(this);
  }

  // MOVE METHODS
  public void moveTo(double x, double y){
    center.x = (int) x;
    center.y = (int) y;
  }
  public void move(int dir, double velocity){
    double velocityDiag = Math.sqrt(velocity/2);
    switch(dir) {
      case 0: center.x += velocity; break;
      case 1: center.x -= velocity; break;
      case 2: center.y -= velocity; break;
      case 3: center.y += velocity; break;
      case 4: center.y -= velocityDiag;
              center.x += velocityDiag; break;
      case 5: center.y += velocityDiag;
              center.x += velocityDiag; break;
      case 6: center.y -= velocityDiag;
              center.x -= velocityDiag; break;
      case 7: center.y += velocityDiag;
              center.x -= velocityDiag; break;
    }
  }
  public boolean directionInFrame(int dir, double factor){
    int xMax = world.getMax_X();
    int xMin = world.getMin_X();
    int yMax = world.getMax_Y();
    int yMin = world.getMin_Y();
    boolean available = false;
    switch(dir){
      case 0: available = (center.x+factor*radius <= xMax); break;
      case 1: available = (center.x-factor*radius >= xMin); break;
      case 2: available = (center.y-factor*radius >= yMin); break;
      case 3: available = (center.y+factor*radius <= yMax); break;
      case 4: available = (center.x+factor*radius <= xMax) &&
                          (center.y-factor*radius >= yMin); break;
      case 5: available = (center.x+factor*radius <= xMax) &&
                          (center.y+factor*radius <= yMax); break;
      case 6: available = (center.x-factor*radius >= xMin) &&
                          (center.y-factor*radius >= yMin); break;
      case 7: available = (center.x-factor*radius >= xMin) &&
                          (center.y+factor*radius <= yMax); break;
    }
    return available;
  }
  public boolean directionOccupied(int dir){
    Shape neighbour = this.world.getClosestShape(this);
    boolean occupied = false;
    if( neighbour != null ){
      switch(dir){
        case 0: occupied = neighbour.contains(center.x+radius, center.y); break;
        case 1: occupied = neighbour.contains(center.x-radius, center.y); break;
        case 2: occupied = neighbour.contains(center.x, center.y-radius); break;
        case 3: occupied = neighbour.contains(center.x, center.y+radius); break;
        case 4: occupied = neighbour.contains(center.x+radius, center.y) ||
                           neighbour.contains(center.x, center.y-radius); break;
        case 5: occupied = neighbour.contains(center.x+radius, center.y) ||
                           neighbour.contains(center.x, center.y+radius); break;
        case 6: occupied = neighbour.contains(center.x-radius, center.y) ||
                           neighbour.contains(center.x, center.y-radius); break;
        case 7: occupied = neighbour.contains(center.x-radius, center.y) ||
                           neighbour.contains(center.x, center.y+radius); break;
      }
    }
    return occupied;
  }

  // INTERACTIONS
  public void userClicked(double atX, double atY){
    System.out.println("click");
  }
  public void userTyped(char key){
    System.out.println("key");
  }
  public boolean enslaved(){
    boolean slave = false;
    Shape neighbour = this.world.getClosestShape(this);
    if( neighbour instanceof Wrapper ){
      slave = this.contains(neighbour.getCenter().x, neighbour.getCenter().y);
    }
    return slave;
  }
}

public class Alien extends AbstractShape implements Shape, Animation {
  // ATTRIBUTES
  double velocity = 2;
  boolean clicked = false;
  int dir = rand.nextInt(8);
  int gender = rand.nextInt(2);
  int steps = 400;
  final int RADIUS = 20;
  // gener 0: female, 1: male

  // CONSTRUCTOR
  public Alien(double x, double y) {
    this.radius = RADIUS;
    this.color = Color.GREEN;
    this.center = new Point(x,y);
  }
  public Alien() {
    this.radius = RADIUS;
    this.color = Color.GREEN;
    int x = rand.nextInt(15)*20-150;
    int y = rand.nextInt(15)*20-150;
    this.center = new Point(x,y);
  }

  // DRAW METHODS
  public void draw(Graphics g){
    g.setColor(color);
    if( gender == 1 ){
      fillNtagon(g, center.x, center.y-1.1*radius, radius/3, 3);    //antenna
    } else {
      fillNtagon(g, center.x, center.y-1.1*radius, radius/2.5, 0);  //antenna
    }
    outlineCircle(g, center.x, center.y-radius/2, radius);          //helmet
    fillCircle(g, center.x, center.y-radius/4, radius);             /head
    fillAlienBody(g, center.x, center.y, radius);                   //body
    outlineNtagon(g, center.x, center.y+radius, radius/3, 5);       //hooverpad
  }
  public static void fillAlienBody(Graphics g, double x, double y, double r){
    int[] x_coords = { (int) (x-r*0.7), (int) (x+r*0.7), (int) (x+r*0.2), (int) (x+r*0.4),
                       (int) (x-r*0.4), (int) (x-r*0.2) };
    int[] y_coords = { (int) (y),       (int) (y),       (int) (y+r/2),   (int) (y+r),
                       (int) (y+r),     (int) (y+r/2) };
    Polygon p = new Polygon(x_coords, y_coords, 6);
    g.fillPolygon(p);
  }
  public void destroy(){
    this.world.removeShape(this);
    for( int i=0; i<radius; i++ ){
      this.world.addShape(new PanikStuck(center.x, center.y,
                                         getColor(), 0, rand.nextInt(8)+radius/4 ));
    }
  }

  // PLAY METHOD
  public void play(){
    steps++;
    if( this.enslaved() ){
      Shape master = this.world.getClosestShape(this);
      if( !(master instanceof TimeWrapper) ){
        move(dir, velocity);
      }
    } else {
      if ( steps%(2*RADIUS) == 0 ){ //make him move smoother
        dir = rand.nextInt(8);
        while( !directionInFrame(dir, 6.0) ){
          dir = rand.nextInt(8);
        }
      }
      move(dir, velocity);
      if( this.gender == 0 && steps > 500 && directionOccupied(dir) ){
        Shape neighbour = this.world.getClosestShape(this);
        if( neighbour instanceof Alien ){
          Alien mate = (Alien) neighbour;
          if( mate.gender == 1){
            steps = 0;
            this.world.addShape(new BabyAlien(center.x, center.y));
          }
        }
      }
    }
  }

  // INTERACTIONS
  public void userClicked(double atX, double atY){
    if( this.contains(atX, atY) ){
      this.destroy();
    }
  }
  public boolean contains(double x, double y) {
    if( x<(center.x-radius) || x>center.x+radius || y<(center.y-radius) || y>(center.y+radius) ){
      return false;
    } else {
      return true;
    }
  }
}

public class BabyAlien extends Alien implements Shape, Animation {
  // CONSTRUCTOR
  public BabyAlien(double x, double y) {
    this.velocity = 0.5;
    this.radius = 5;
    this.color = Color.CYAN;
    this.center = new Point(x,y);
  }
  public BabyAlien() {
    this.velocity = 0.5;
    this.radius = 5;
    this.color = Color.CYAN;
    this.center = new Point();
  }

  // DRAW METHODS
  public void draw(Graphics g){
    g.setColor(color);
    fillCircle(g, center.x, center.y-radius/4, radius);         //head
    fillAlienBody(g, center.x, center.y, radius);               //body
    outlineNtagon(g, center.x, center.y+radius, radius/3, 5);   //hooverpad
  }
  public void mature(){
    this.world.removeShape(this);
    this.world.addShape(new Alien(center.x, center.y));
    for( int i=0; i<4; i++ ){
      this.world.addShape(new PanikStuck(center.x, center.y, Color.GREEN, 0, 8 ));
      this.world.addShape(new PanikStuck(center.x, center.y, Color.CYAN, 0, 8 ));
    }
  }

  // PLAY METHOD
  public void play(){
    steps++;
    if( radius < RADIUS*.75 ){
      radius *= 1.005;
      velocity *= 1.005;
      if( this.enslaved() ){
        Shape master = this.world.getClosestShape(this);
        if( !(master instanceof TimeWrapper) ){
          move(dir, velocity);
        }
      } else {
        if ( steps%(2*RADIUS) == 0 ){ //make him move smoother
          dir = rand.nextInt(8);
          while( !directionInFrame(dir, 6.0) ){
            dir = rand.nextInt(8);
          }
        }
        move(dir, velocity);
      }
    } else {
      this.mature();
    }
  }
}

public class Wrapper extends AbstractShape implements Shape, Animation {
  // ATTRIBUTES
  double velocity = 4;
  boolean catched = false;
  Shape slave;
  int dir = rand.nextInt(8);
  final int RADIUS = 35;

  // CONSTRUCTOR
  public Wrapper() {
    int x = rand.nextInt(15)*20-150;
    int y = rand.nextInt(15)*20-150;
    this.center = new Point(x,y);
    this.radius = 35;
    this.color = Color.RED;
  }
  public Wrapper(double x, double y) {
    this.center = new Point(x, y);
    this.color = Color.RED;
    this.radius = 35;
  }

  // DRAW METHODS
  public void draw(Graphics g){
    outlineCircle(g, center.x, center.y, radius);
  }

  // PLAY METHOD
  public void play(){
    steps++;
    if( catched ){
      moveTo(this.slave.getCenter().x, this.slave.getCenter().y);
      if( this.world.getClosestShape(this) != slave ){
        catched = false;
      }
    } else {
      if( steps > 30 && directionOccupied(dir) && !(this.world.getClosestShape(this) instanceof Wrapper) ){
        catched = true;
        slave = this.world.getClosestShape(this);
      } else {
        while( !directionInFrame(dir, 1.2) ){
          dir = rand.nextInt(8);
        }
        move(dir, velocity);
      }
    }
  }
}

public class TimeWrapper extends Wrapper implements Shape, Animation {
  // CONSTRUCTOR
  public TimeWrapper() {
    int x = rand.nextInt(15)*20-150;
    int y = rand.nextInt(15)*20-150;
    this.center = new Point(x,y);
    this.radius = 35;
    this.color = Color.PINK;
  }

  // PLAY METHOD
  public void play(){
    steps++;
    if( catched ){
      if( steps > 100 ){
        for( int i=0; i<3; i++){
          this.world.addShape(new MiniWrapper(center.x, center.y));
        }
        this.world.removeShape(this);
      } else {
        if( this.world.getClosestShape(this) != slave ){
          this.world.removeShape(this);
        }
      }
    } else {
      if( directionOccupied(dir) && !(this.world.getClosestShape(this) instanceof Wrapper) ){
        catched = true;
        slave = this.world.getClosestShape(this);
        moveTo(this.slave.getCenter().x, this.slave.getCenter().y);
        steps = 0;
      } else {
        while( !directionInFrame(dir, 1.2) ){
          dir = rand.nextInt(8);
        }
        move(dir, velocity);
      }
    }
  }
}

public class MiniWrapper extends Wrapper implements Shape, Animation {
  // CONSTRUCTOR
  public MiniWrapper(double x, double y) {
    this.center = new Point(x,y);
    this.radius = 8.75;
    this.color = Color.RED;
    this.velocity = 1;
  }

  // PLAY METHOD
  public void play(){
    steps++;
    if( radius < RADIUS ){
      while( !directionInFrame(dir, 1.2) ){
        dir = rand.nextInt(8);
      }
      move(dir, velocity);
      radius *= 1.01;
      velocity *= 1.01;
    } else {
      this.world.addShape(new Wrapper(center.x, center.y));
      this.world.removeShape(this);
    }
  }
}

public class Panik extends AbstractShape implements Shape, Animation {
  // ATTRIBUTES
  double velocity = 1;
  int collisions = 0;
  int shaking = 0;
  int dir = rand.nextInt(8);

  // CONSTRUCTOR
  public Panik() {
    int x = rand.nextInt(15)*20-150;
    int y = rand.nextInt(15)*20-150;
    this.center = new Point(x,y);
    this.radius = 12;
  }

  // DRAW METHODS
  public void draw(Graphics g){
    g.setColor(color);
    fillNtagon(g, center.x, center.y, radius, 7);
  }
  public void destroy(){
    this.world.removeShape(this);
    for( int i=0; i<20; i++ ){
      this.world.addShape(new PanikStuck(center.x, center.y, getColor(),
                                         rand.nextInt(4)+5, rand.nextInt(5)+3 ));
    }
  }

  // PLAY METHOD
  public void play(){
    steps++;
    while( !directionInFrame(dir, 1.2) ){
      dir = rand.nextInt(8);
    }
    if( directionOccupied(dir) ){
      collisions++;
      steps = 0;
      dir = rand.nextInt(8);
    }
    if( shaking > 0 || collisions > 20 && steps < 50 ){
      this.panic();
    } else {
      move(dir, velocity);
      if( steps > 50 ){
        collisions = 0;
      }
    }
  }
  public void panic(){
    if( shaking<50 ){
      if( shaking%2 ==0 ){
        move(0, 2*velocity);
      } else {
        move(1, 2*velocity);
      }
      shaking++;
    } else {
      this.destroy();
    }
  }
}

public class PanikStuck extends AbstractShape implements Shape, Animation {
  // ATTRIBUTES
  double velocity = 6;
  int shape = rand.nextInt(4)+6;
  int dir = rand.nextInt(8);
  // dir: 0: right, 1: left, 2: up, 3: down, 4: upright, 5:upleft, 6: downright, 7:downleft

  // CONSTRUCTOR
  public PanikStuck(double x, double y, Color color, int shape, double radius){
    this.radius = radius;
    this.velocity = radius;
    this.center = new Point(x,y);
    this.color = color;
    this.shape = shape;
  }
  public PanikStuck() {
    this.radius = rand.nextInt(4)+4;
    this.velocity = radius;
    this.center = new Point();
  }

  // DRAW METHODS
  public void draw(Graphics g){
    g.setColor(color);
    fillNtagon(g, center.x, center.y, radius, shape);
  }

  // PLAY METHOD
  public void play(){
    steps++;
    if( steps <= 10 ){
      this.move(dir, 10/velocity);
      velocity *= 0.9;
    } else if( center.y >= world.getMin_Y() ){
      move(3, velocity);
      velocity *= 1.1;
    }
  }
}

public class Mjolnir extends AbstractShape implements Shape, Animation {
  // ATTRIBUTES
  double radius = 20;
  final double VELOCITY = 20;
  Color color = Color.WHITE;

  // CONSTRUCTOR
  public Mjolnir() {
    int x = 200;
    int y = rand.nextInt(15)*20-150;
    this.center = new Point(x,y);
  }

  // DRAW METHODS
  public void draw(Graphics g){
    g.setColor(color);
    fillMjolnirHead(g, center.x, center.y, radius);
    fillMjolnirGrip(g, center.x, center.y, radius);
  }
  public static void fillMjolnirHead(Graphics g, double x, double y, double r){
    int[] x_coords = { (int) (x),   (int) (x),   (int) (x+0.3*r), (int) (x+1.2*r),
                       (int) (x+1.5*r), (int) (x+1.5*r), (int) (x+1.2*r), (int) (x+0.3*r)  };
    int[] y_coords = { (int) (y-r), (int) (y+r), (int) (y+1.3*r), (int) (y+1.3*r),
                       (int) (y+r),     (int) (y-r),     (int) (y-1.3*r), (int) (y-1.3*r)  };
    Polygon p = new Polygon(x_coords, y_coords, 8);
    g.fillPolygon(p);
  }
  public static void fillMjolnirGrip(Graphics g, double x, double y, double r){
    int[] x_coords = { (int) (x+1.5*r), (int) (x+1.5*r), (int) (x+4.5*r), (int) (x+4.5*r) };
    int[] y_coords = { (int) (y-0.2*r), (int) (y+0.2*r), (int) (y+0.2*r), (int) (y-0.2*r) };
    Polygon p = new Polygon(x_coords, y_coords, 4);
    g.fillPolygon(p);
  }
  public void destroy(){
    this.world.removeShape(this);
    for( int i=0; i<100; i++ ){
      this.world.addShape(new PanikStuck(center.x, center.y, Color.WHITE, 4, rand.nextInt(10)+3 ));
    }
  }

  // PLAY METHOD
  public void play(){
    if( velocity > VELOCITY/2 ){
      move(1, velocity);
      if( steps < 15 ){
        Shape neighbour = this.world.getClosestShape(this);
        if( neighbour != null && (neighbour.contains(center.x, center.y) || neighbour.contains(center.x, center.y-radius) || neighbour.contains(center.x, center.y+radius)) ){
          neighbour.destroy();
          velocity -= VELOCITY/8;
          steps++;
        }
      }
    } else {
      move(7, velocity*4);
      velocity += 3;
    }
  }

  // INTERACTIONS
  public void userClicked(double atX, double atY){
    if( this.contains(atX, atY) ){
      this.destroy();
    }
  }
  public boolean contains(double x, double y) {
    if( x<(center.x-radius) || x>center.x+radius || y<(center.y-radius) || y>(center.y+radius) ){
      return false;
    } else {
      return true;
    }
  }
}
```