Übungsgruppe 3
273784 Philipp Fischer

Übung Programmierung
Blatt 9

Einzelabgabe Aufgabe 3
von Philipp Fischer

WorldCoordinate.java

```java
/**
 * SHEET 9 - EXERCISE 3
 * This class basically is a data structure
 * for managing coordinates of the world.
 * It is self-explanatory, so comments were left out
 *
 * @author 273784 Philipp Fischer & 274196 Lucas Brutschy
 * @version 22.12.2006
 * platform: J2RE 1.5.0_08-b03, Linux 2.6.17-gentoo
*/
public class WorldCoordinate {
    private int x, y;
    /* These fields will be used for looking in the neighborhood
     * of a certain individual
     */
    public static final WorldCoordinate[] directNeighborhood = {
        new WorldCoordinate(-1,0),
        new WorldCoordinate(1,0),
        new WorldCoordinate(0,-1),
        new WorldCoordinate(0,1)
    };

    public static final WorldCoordinate[] extendedNeighborhood = {
        new WorldCoordinate(-1,1),
        new WorldCoordinate(1,1),
        new WorldCoordinate(-1,-1),
        new WorldCoordinate(1,-1)
    };

    public static final WorldCoordinate[] fullNeighborhood = {
        new WorldCoordinate(-1,0),
        new WorldCoordinate(1,0),
        new WorldCoordinate(0,-1),
        new WorldCoordinate(0,1),
        new WorldCoordinate(-1,1),
        new WorldCoordinate(1,1),
        new WorldCoordinate(-1,-1),
        new WorldCoordinate(1,-1)
    };

    public WorldCoordinate(int x, int y) {
        setXY(x, y);
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public void setY(int y) {
        this.y = y;
    }

    public void setX(int x) {
        this.x = x;
    }

    public void setXY(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public WorldCoordinate getTransformed(int relativeX, int relativeY) {
        return new WorldCoordinate(this.x + relativeX, this.y + relativeY);
    }

    public WorldCoordinate getTransformed(WorldCoordinate relative) {
        return new WorldCoordinate(this.x + relative.getX(), this.y + relative.getY());
    }

    public String toString() {
        return "("+x+","+y+")";
    }
}
```

Individual.java

```java
import java.util.Random;

/**
 * SHEET 9 - EXERCISE 3
 * Individual (abstract)
 *
 * @author 273784 Philipp Fischer
 * @version 22.12.2006
 * platform: J2RE 1.5.0_08-b03, Linux 2.6.17-gentoo
*/
public abstract class Individual extends Thread {
    // Saves the position, the individual is currently at
    private WorldCoordinate position;

    // The world, this individual is living in
    private WoToBa myWorld;

    // The age indicates wether this already is a Tonky, or still a Wonky
    // Also a Basto needs the age not to pair twice in one round
    private int age;
    private int ageLastPaired;

    /**
     * Returns the age of the individual and
     * optionally increases the age afterwards.
     * @param increaseAfter If you want the individual to age
     */
    public int getAge() {
        return age;
    }

    public void age() {
        age++;
    }

    /**
     * Marks this round with "already paired"
     */
    public void pair() {
        ageLastPaired = age;
    }

    /**
     * Returns if the individual has paired this round
     */
    public boolean didPair() {
        return ageLastPaired == age;
    }

    /**
     * The standard constructor for newborns
     */
    public Individual() {
        this(0);
    }

    /**
     * Constructor for specifying the age
     */
    public Individual(int age) {
        // Living nowhere
        this.myWorld = null;
        setPosition(null);

        // Set the age
        this.age = age;
        ageLastPaired = -1;
    }

    /**
     * Returns the world, this individual is living on
     */
    public WoToBa getWorld() {
        return myWorld;
    }

    /**
     * Sets or changes the world, this individual is living on
     */
    public void setWorld(WoToBa world) {
```

Individual.java

```
81          this.myWorld = world;
82      }
83
84      /**
85       * Sets the current position of the individual
86       */
87      public void setPosition(WorldCoordinate position) {
88          if(position == null) return;
89          this.position = position;
90      }
91
92      /**
93       * Returns the current position of the individual
94       */
95      public WorldCoordinate getPosition() {
96          return position;
97      }
98
99      /**
100      * Moves the individual by one
101      * individual-specific step-unit
102      */
103     public void move() {
104         Random numberGenerator = new Random();
105
106         /* Now we try to move the Individual, but only try 20 times to
107          * prevent any infinite loops. If we can't move after 20 trials
108          * it may be surrounded.
109          */
110         int trialCounter = 0;
111         boolean moved = false;
112         do {
113             /* We generate two relative coordinates, with one
114              * of the values -1,0,1 respectively
115              */
116             int xDirection = numberGenerator.nextInt(3) - 1;
117             int yDirection = numberGenerator.nextInt(3) - 1;
118
119             /* Now we try to move the Individual. Note that this method
120              * is synchronized, so that only one individual can move at a time
121              */
122             moved = getWorld().moveIndividualBy(this, xDirection, yDirection);
123
124             trialCounter++;
125         } while(trialCounter < 20 && !moved);
126     }
127
128     /**
129      * Makes the individual perform the
130      * appriopriate action for the current
131      * round (e.g. pair, eat, die)
132      * @return If this action made us die
133      */
134     public abstract boolean action();
135
136     /**
137      * Starts the Thread
138      */
139     public void run() {
140         if(myWorld == null) return;
141
142         // Caches the kind of this thread
143         int kind = (this instanceof TonkyWonky) ? 1 : 2;
144
145         myWorld.addThread(kind);
146
147         boolean goingToDie = false;
148
149         // Wait for tasks as long as the task-number is positive.
150         // Negative means die (for all individuals)
151         while(myWorld.getAction() >= 0 && !goingToDie) {
152             // Now we wait for the global notifyAll signal from the world
153             synchronized(myWorld) {
154                 // We are ready to do something
155                 myWorld.threadEnteringReadyState();
156
157                 try {
158                     myWorld.wait();
159
160                 /* If an interrupt was caught, somebody wants us to die.
```

Individual.java

```
                     * Do so by returning from the run() method.
                     */
                    } catch(InterruptedException e) {
                        myWorld.removeThread(kind, true);
                        return;
                    }
                    myWorld.threadEnteringBusyState();
                }

                // Now what does the world want us to do?
                switch(myWorld.getAction()) {
                    case 0: // Nothing
                        break;
                    case 1: // Move
                        move();
                        // If this is a Basto, we move two times
                        if(this instanceof Basto) {
                            move();
                        }
                        break;
                    case 2: // Do your action
                        //If action returns true, this means die
                        if(action()) goingToDie = true;
                        break;
                    default: break;
                }

            }

            myWorld.removeThread(kind, false);
        }

        /**
         * Returns the appriopriate Symbol for this individual as a string
         */
        public String toString() {
            char[] kindChars = new char[1];
            kindChars[0] = this.getKind();
            return new String(kindChars);
        }

        /**
         * Returns the appriopriate Symbol for this individual as a chracter
         */
        public abstract char getKind();
    }
```

TonkyWonky.java

```java
import java.util.Random;

/**
 * SHEET 9 - EXERCISE 3
 * This class implements the behaviour
 * of a Tonky or a Wonky
 *
 * @author 273784 Philipp Fischer
 * @version 22.12.2006
 * platform: J2RE 1.5.0_08-b03, Linux 2.6.17-gentoo
*/
public class TonkyWonky extends Individual {
    // Specifies the number of children a Tonky drops
    private static final int TONKYCHILDREN = 3;

    /**
     * Standard constructor forwarding
     */
    public TonkyWonky() {
        super();
    }

    /**
     * Constructor forwarding
     */
    public TonkyWonky(int age) {
        super(age);
    }

    /**
     * This method let's the Tonky or Wonky perform
     * its characteristic action such as pair
     * or become a Tonky
     */
    public boolean action() {
        // A Wonky doesn't do anything
        // Nevertheless the age always increases
        if(getAge() < 5) {
            age();
            return false;
        }

        age();

        /* Only one Tonky can look for partners at a time
         * so that they can't discover each other at once.
         */
        synchronized(getWorld()) {
            if(didPair()) {
                return false;
            }

            TonkyWonky partner = (TonkyWonky)getWorld().checkNeighborhood(this, "T", false);
            if(partner != null) {

                if(!partner.didPair()) {
                    this.pair();
                    partner.pair();

                    Random randomGenerator = new Random();
                    for(int i = 0; i < TONKYCHILDREN; i++) {
                        TonkyWonky newChild = new TonkyWonky();

                        /* If this random number (0 or 1) is 0
                         * the Tonky drops the child around him,
                         * otherwise around his partner
                         */
                        boolean success = false;
                        if(randomGenerator.nextInt(2) == 0) {
                            if(!getWorld().dropIndividual(this, newChild)) {
                                // If we don't have space around us, try the partner
                                success = getWorld().dropIndividual(partner, newChild);
                            } else {
                                success = true;
                            }
                        } else {
                            success = getWorld().dropIndividual(partner, newChild);
                        }

                        if(success) newChild.start();
```

TonkyWonky.java

```
                    }
                }
            }
        }

        return false;
    }

    /**
     * Returns the appriopriate Symbol for
     * this individual (T for Tonky or W for Wonky)
     */
    public char getKind() {
        return (getAge() < 5)?'w':'T';
    }

}

```

Basto.java

```java
import java.util.Random;

/**
 * SHEET 9 - EXERCISE 3
 * This class implements the behaviour
 * of a Basto
 *
 * @author 273784 Philipp Fischer
 * @version 22.12.2006
 * platform: J2RE 1.5.0_08-b03, Linux 2.6.17-gentoo
*/
public class Basto extends Individual {
    // Specifies the number of children a Basto drops
    private static final int BASTOCHILDREN = 1;

    /* Indicates wether the Basto is currently in an
     * eating sequence or in a starving sequence.
     * true means eating.
     */
    private boolean eatStarveState = false;

    // How long is that sequence?
    private int eatStarveDuration = 0;

    /**
     * Performs any tasks the Basto has to
     * (eat, pair)
     */
    public boolean action() {
        // First: Get older
        age();

        // Then: EAT

        /* Look for Wonkies or Tonkies around us.
         * Nothing should change while looking for a TonkyWonky, so
         * we synchronize with the world here.
         */
        synchronized(getWorld()) {
            // Look around for Wonkys and Tonkys
            TonkyWonky food = (TonkyWonky)getWorld().checkNeighborhood(this, "[wT]", false);
            if(food != null) {
                // Kill and eat
                food.interrupt();
                getWorld().setIndividualTo(null, food.getPosition());

                /* If we were eating before, change the state to starving and start with 1.
                 * If not, continue starving.
                 */
                if(!eatStarveState) {
                    eatStarveState = true;
                    eatStarveDuration = 1;
                } else {
                    eatStarveDuration++;
                }

            } else {
                /* If we were starving before, change the state to eating and start with 1.
                 * If not, continue eating.
                 */
                if(eatStarveState) {
                    eatStarveState = false;
                    eatStarveDuration = 1;
                } else {
                    eatStarveDuration++;
                }
            }
        }

        if(eatStarveState == false && eatStarveDuration >= 15) {
            // Die of hunger
            getWorld().setIndividualTo(null, getPosition());
            interrupt();
            return true;
        } else if(eatStarveState == true && eatStarveDuration >= 20) {
            // Die of adiposis
            getWorld().setIndividualTo(null, getPosition());
            interrupt();
            return true;
        }
```

Basto.java

```java

        // Then: PAIR
        synchronized(getWorld()) {
            if(didPair()) return false;

            Basto partner = (Basto)getWorld().checkNeighborhood(this, "B", false);
            if(partner != null) {
                if(!partner.didPair()) {
                    this.pair();
                    partner.pair();

                    Random randomGenerator = new Random();
                    for(int i = 0; i < BASTOCHILDREN; i++) {
                        Basto newChild = new Basto();

                        /* If this random number (0 or 1) is 0
                         * the Basto drops the child around him,
                         * otherwise around his partner
                         */
                        boolean success = false;
                        if(randomGenerator.nextInt(2) == 0) {
                            if(!getWorld().dropIndividual(this, newChild)) {
                                // If we don't have space around us, try the partner
                                success = getWorld().dropIndividual(partner, newChild);
                            } else {
                                success = true;
                            }
                        } else {
                            success = getWorld().dropIndividual(partner, newChild);
                        }

                        if(success) newChild.start();
                    }
                }
            }
        }
        return false;
    }

    /**
     * Returns the appriopriate Symbol for
     * this individual (B for Basto)
     */
    public char getKind() {
        return 'B';
    }
}

```

WoToBa.java

```java
import java.util.Random;

/**
 * SHEET 9 - EXERCISE 3
 * WoToBa the virtual WonkyTonkyBasto world
 *
 * @author 273784 Philipp Fischer
 * @version 22.12.2006
 * platform: J2RE 1.5.0_08-b03, Linux 2.6.17-gentoo
*/
public class WoToBa {
    // This array saves the references to the individuals living in the world
    private Individual[][] world;

    // Here width and height of the world are being saved
    private int width, height;

    /* This specifies, which action to perform the next time
     * the individuals receive a notify-signal.
     * -1 = die
     *  0 = do nothing
     *  1 = move
     *  2 = do your action
     */
    private int nextAction;

    /* Serves as a mutex on which the threads notify the world
     * if a thread becomes ready or busy
     */
    public Object readyMutex = new Object();

    // Indicates how many threads are ready for the next action
    private int readyCount = 0;

    // How many threads are currently running for this world?
    private int tonkyWonkyThreadCount = 0;
    private int bastoThreadCount = 0;

    /**
     * The standard constructor creates a world with the given size
     * and initializes it without individuals
     * @param width The new width of the world
     * @param height The new height of the world
     */
    public WoToBa(int width, int height) {
        // If we have a valid size, intialize a world without individuals
        if(width > 0 && height > 0) {
            this.width = width;
            this.height = height;

            world = new Individual[width][height];
            for(int x = 0; x < width; x++) {
                for(int y = 0; y < height; y++) {
                    world[x][y] = null;
                }
            }
        } else {
            this.width = 0;
            this.height = 0;

            world = null;
        }
    }

    /**
     * Returns the number of threads that are currently marked as ready
     */
    public int getReadyThreadCount() {
        return readyCount;
    }

    /**
     * Increases the number of thread that are marked as ready
     */
    public void threadEnteringReadyState() {
        synchronized(readyMutex) {
            readyCount++;
            readyMutex.notifyAll();
        }
    }

    /**
     * Decreases the number of thread that are marked as ready
     */
    public void threadEnteringBusyState() {
        synchronized(readyMutex) {
            readyCount--;
            readyMutex.notifyAll();
        }
```

WoToBa.java

```
90      }
91
92      /**
93       * Returns the number of threads of a certain kind currently running
94       * @param kind 1 means TonkyWonky, 2 means Basto, 0 makes this method return the total
95       */
96      public int getThreadCount(int kind) {
97          switch(kind) {
98              case 0:
99                  // We synchronize this, so threadcounts don't change inbetween the sum
100                 synchronized(readyMutex) {
101                     return bastoThreadCount + tonkyWonkyThreadCount;
102                 }
103             case 1:
104                 return tonkyWonkyThreadCount;
105             case 2:
106                 return bastoThreadCount;
107             default:
108                 return -1;
109         }
110     }
111
112     /**
113      * Increases the number of threads currently running
114      * @param kind The corresponding kind of Individual (1=TonkyWonky, 2=Basto)
115      */
116     public void addThread(int kind) {
117         synchronized(readyMutex) {
118             switch(kind) {
119                 case 1:
120                     tonkyWonkyThreadCount++;
121                     break;
122                 case 2:
123                     bastoThreadCount++;
124                     break;
125             }
126             readyMutex.notifyAll();
127         }
128     }
129
130     /**
131      * Decreases the number of threads currently running
132      * @param kind The corresponding kind of Individual (1=TonkyWonky, 2=Basto)
133      * @param wasBusy Indicates wether the thread to remove was busy before
134      */
135     public void removeThread(int kind, boolean wasBusy) {
136         synchronized(readyMutex) {
137             switch(kind) {
138                 case 1:
139                     tonkyWonkyThreadCount--;
140                     break;
141                 case 2:
142                     bastoThreadCount--;
143                     break;
144             }
145             if(wasBusy) {
146                 threadEnteringBusyState();
147             }
148             readyMutex.notifyAll();
149         }
150     }
151
152
153     /**
154      * Returns the width of the world
155      */
156     public int getWorldWidth() {
157         return width;
158     }
159
160     /**
161      * Returns the height of the world
162      */
163     public int getWorldHeight() {
164         return height;
165     }
166
167     /**
168      * Sets the next action (see nextAction) for all Individuals of this world
169      * @param action The next action to perform
170      */
171     synchronized public void setAction(int action) {
172         this.nextAction = action;
173     }
174
175     /**
176      * Returns the next action (see nextAction) for all Individuals of this world
177      * @param action The next action to perform
178      */
```

WoToBa.java

```
179     public int getAction() {
180         return this.nextAction;
181     }
182 
183     /**
184      * Makes the individuals perform the previously specified action
185      * and returns when they are done.
186      */
187     public void doNextAction() {
188         if(getThreadCount(0) < 1) return;
189 
190         // First wait for everybody to be ready
191         synchronized(readyMutex) {
192 
193             try {
194                 while(readyCount < getThreadCount(0)) {
195                     readyMutex.wait();
196                 }
197             } catch(InterruptedException e) {
198                 return;
199             }
200         }
201 
202         // Then notify all threads to start their actions
203         synchronized(this) {
204             this.notifyAll();
205         }
206 
207         // And wait again until everybody is done
208         synchronized(readyMutex) {
209             try {
210                 readyMutex.wait();
211 
212                 while(readyCount < getThreadCount(0)) {
213                     readyMutex.wait();
214                 }
215             } catch(InterruptedException e) {
216                 return;
217             }
218         }
219     }
220 
221     /**
222      * Returns if the given position is in the world
223      * @param position The position to check for validity
224      * @return If the given position is in the world
225      */
226     public boolean isPositionValid(WorldCoordinate position) {
227         return !(position.getX() < 0 || position.getY() < 0
228                 || position.getX() >= width || position.getY() >= height);
229     }
230 
231     /**
232      * Returns the individual at the given position
233      * @param position The position you want to look at
234      * @return Null if there is nothing, the Individual otherwise
235      */
236     public Individual getContentAt(WorldCoordinate position) {
237         if(world == null) return null;
238         if(!isPositionValid(position)) return null;
239         return world[position.getX()][position.getY()];
240     }
241 
242     /**
243      * Returns an individual at a position relatively seen to another
244      * Individual
245      * @param center The individual that is looking around
246      * @param relativeX The horizontal offset
247      * @param relativeY The vertical offset
248      * @return An individual, or null if nothing is there
249      */
250     public Individual getNeighbor(Individual center, int relativeX, int relativeY) {
251         if(world == null) return null;
252         return getContentAt(center.getPosition().getTransformed(relativeX, relativeY));
253     }
254 
255     /**
256      * This method checks either the simple or the full neighborhood
257      * for a certain race and returns the coordinate of the first occurence.
258      * @param seeker The individual that looks around
259      * @param identifier The race to look for
260      * @param FullNeighborhood If the full neighborhood should be scanned
261      * @return Null, if nothing was found, the coordinates otherwise
262      */
263     public Individual checkNeighborhood(Individual seeker, String identifiers, boolean FullNeighborhood) {
264         if(world == null || seeker == null) return null;
265 
266         // The current neighbor looking at
267         Individual neighbor;
```

WoToBa.java

```
268
269         // Simple neighborhood:
270         for(WorldCoordinate currentDirection : WorldCoordinate.directNeighborhood) {
271             if((neighbor = getContentAt(seeker.getPosition().getTransformed(currentDirection))) != null) {
272                 // If the identifier of this individual matches our search-expression,
273                 // then return its position to the seeker
274                 if(neighbor.toString().matches(identifiers)) {
275                     return neighbor;
276                 }
277             }
278         }
279
280         /* If nothing was found until now and we only want
281          * the simple neighborhood, return null
282          */
283         if(!FullNeighborhood) {
284             return null;
285         }
286
287         for(WorldCoordinate currentDirection : WorldCoordinate.extendedNeighborhood) {
288             if((neighbor = getContentAt(seeker.getPosition().getTransformed(currentDirection))) != null) {
289                 // If the identifier of this individual matches our search-expression,
290                 // then return its position to the seeker
291                 if(neighbor.toString().matches(identifiers)) {
292                     return neighbor;
293                 }
294             }
295         }
296
297         return null;
298     }
299
300     /**
301      * Moves an individual in the current world to another position
302      * @param toMove
303      * @param relativeX
304      * @param relativeY
305      * @return
306      */
307     synchronized public boolean moveIndividualBy(Individual toMove, int relativeX, int relativeY) {
308         if(world == null) return false;
309
310         // Is there anything to move?
311         if(toMove == null) return false;
312
313         // Is it in this world?
314         if(toMove.getWorld() != this) return false;
315
316         // Do we really want to move? If not we are done
317         if(relativeX == 0 && relativeY == 0) return true;
318
319         WorldCoordinate oldPosition = toMove.getPosition();
320         WorldCoordinate newPosition = oldPosition.getTransformed(relativeX, relativeY);
321
322         // Chek if both the old and the new position are valid
323         if(!isPositionValid(oldPosition) || !isPositionValid(newPosition)) return false;
324
325         // Is there someone else at the target? If yes, we can't move there
326         if(getContentAt(newPosition) != null) return false;
327
328         // Now we are fine. Change the references and clear the old field
329         world[newPosition.getX()][newPosition.getY()] = toMove;
330         world[oldPosition.getX()][oldPosition.getY()] = null;
331
332         // Also inform the individual about its new position
333         toMove.setPosition(newPosition);
334
335         return true;
336     }
337
338     /**
339      * Places a given individual at the given position in the world.
340      * @param toSet The individual to place
341      * @param position The position to place the individual at
342      * @return If the placement was done succesfully
343      */
344     synchronized public boolean setIndividualTo(Individual toSet, WorldCoordinate position) {
345         if(world == null) return false;
346
347         if(!isPositionValid(position)) return false;
348
349         /* We don't want to overwrite individuals
350          * unless this is explicitly wanted (with toSet==null)
351          */
352         if(toSet != null && world[position.getX()][position.getY()] != null) return false;
353
354         world[position.getX()][position.getY()] = toSet;
355
356         // If this really is an individual, inform it about its new position
```

WoToBa.java

```
357             if(toSet != null) {
358                 toSet.setWorld(this);
359                 toSet.setPosition(position);
360             }
361 
362             return true;
363         }
364 
365         /**
366          * Randomly drops a certain Individual in one of the
367          * fields around another one
368          * @param center The Individual that drops
369          * @param toDrop The Individual that is dropped
370          * @return If the individual has been dropped successfully
371          */
372         public boolean dropIndividual(Individual center, Individual toDrop) {
373             if(world == null) return false;
374             if(center == null) return false;
375             return dropIndividual(center.getPosition(), toDrop);
376         }
377 
378         /**
379          * Randomly drops a certain Individual in one of the
380          * fields around the given position
381          * @param center The Individual that drops
382          * @param toDrop The Individual that is dropped
383          * @return If the individual has been dropped successfully
384          */
385         public boolean dropIndividual(WorldCoordinate center, Individual toDrop) {
386             if(world == null) return false;
387             if(center == null) return false;
388 
389             Random randomGenerator = new Random();
390 
391             // Try dropping 10 times
392             int trialCounter = 0;
393             while(trialCounter++ < 10) {
394                 WorldCoordinate currentPosition;
395                 currentPosition = center.getTransformed(WorldCoordinate.fullNeighborhood[
396                                         randomGenerator.nextInt(WorldCoordinate.fullNeighborhood.length)
397                                     ]);
398                 if(isPositionValid(currentPosition) && getContentAt(currentPosition) == null) {
399                     return setIndividualTo(toDrop, currentPosition);
400                 }
401             }
402 
403             // Well this did not work, but maybe it was fate. Now try without randomization
404             for(int i = 0; i < WorldCoordinate.fullNeighborhood.length; i++) {
405                 WorldCoordinate currentPosition;
406                 currentPosition = center.getTransformed(WorldCoordinate.fullNeighborhood[i]);
407                 if(isPositionValid(currentPosition) && getContentAt(currentPosition) == null) {
408                     return setIndividualTo(toDrop, currentPosition);
409                 }
410             }
411 
412             // Everything is blocked. We cannot drop
413             return false;
414         }
415 
416         /**
417          * This method generates a character array that can be passed
418          * to the Visualize class to display the world on the screen
419          * @return The generated character array
420          */
421         synchronized public char[][] toCharArray() {
422             if(world == null) return null;
423             char[][] result = new char[this.width][this.height];
424 
425             for(int y = 0; y < height; y++) {
426                 for(int x = 0; x < width; x++) {
427                     if(world[x][y] == null) {
428                         result[x][y] = ' ';
429                     } else {
430                         result[x][y] = world[x][y].getKind();
431                     }
432                 }
433             }
434             return result;
435         }
436 
437         /**
438          * This method generates an overview of the world in text-form
439          * and returns it as a string.
440          * It needs to be synchronized to prevent a change in the world
441          * while generating the picture of it.
442          */
443         synchronized public String toString() {
444             if(world == null) return "";
445             String result = "";
```

WoToBa.java

```
446             for(int y = 0; y < height; y++) {
447                 for(int x = 0; x < width; x++) {
448                     if(world[x][y] == null) {
449                         result += "- ";
450                     } else {
451                         result += world[x][y] + " ";
452                     }
453                 }
454                 result += "\n";
455             }
456             return result;
457         }
458 
459     }
460 
```

WoToBaTestDrive.java

```java
import java.util.Random;

/**
 * SHEET 9 - EXERCISE 3
 * This class is used to test the WoToBa world
 *
 * @author 273784 Philipp Fischer
 * @version 22.12.2006
 * platform: J2RE 1.5.0_08-b03, Linux 2.6.17-gentoo
*/
public class WoToBaTestDrive {

    /**
     * This method tries to place an individual somewhere in the world.
     * Success is not guaranteed.
     * @param world The world to place it in
     * @param toPlace The individual to place
     */
    public static boolean randomlyPlaceIndividual(WoToBa world, Individual toPlace) {
        Random generator = new Random();
        int xPosition, yPosition;
        int trialCount = 0;

        do {
            xPosition = generator.nextInt(world.getWorldWidth());
            yPosition = generator.nextInt(world.getWorldHeight());
            if(world.setIndividualTo(toPlace, new WorldCoordinate(xPosition, yPosition))) {
                return true;
            }
            trialCount++;
        } while(trialCount < 50);
        return false;
    }

    /**
     * This method tries to create a TonkyWonky and place it somewhere in the world
     * @param world The world to place it in
     * @param startAge The age it is born with
     */
    public static void randomlyCreateTonkyWonky(WoToBa world, int startAge) {
        TonkyWonky newTonky = new TonkyWonky(startAge);
        if(randomlyPlaceIndividual(world, newTonky)) {
            newTonky.start();
        }
    }

    /**
     * This method tries to create a Basto and place it somewhere in the world
     * @param world The world to place it in
     */
    public static void randomlyCreateBasto(WoToBa world) {
        Basto newBasto = new Basto();
        if(randomlyPlaceIndividual(world, newBasto)) {
            newBasto.start();
        }
    }

    /**
     * The entry point of the program
     * @param args (unused)
     */
    public static void main(String[] args) {
        final int worldSize = 20;
        final int tonkyCount = 20;
        final int bastoCount = 5;

        // Create a visualizer to show the world on the screen
        Visualize visualizer = new Visualize(worldSize);

        // Create the WoToBa world "mars"
        WoToBa mars = new WoToBa(worldSize, worldSize);

        // Randomly place Wonkys and Tonky in the world. The amounts can be specified above
        for(int i = 0; i < tonkyCount; i++) {
            randomlyCreateTonkyWonky(mars, 5);
        }

        for(int i = 0; i < bastoCount; i++) {
            randomlyCreateBasto(mars);
        }
```

```
81
82          //Iterate 100 times
83          int i = 0;
84          while(i++ < 100) {
85              visualizer.anzeigen(mars.toCharArray(), i);
86
87              // And then release the command to move
88              mars.setAction(1);
89              mars.doNextAction();
90
91              visualizer.anzeigen(mars.toCharArray(), i);
92
93              // And then release a command to do the individual-specific action
94              mars.setAction(2);
95              mars.doNextAction();
96
97              // If one of the two races extincted, we can quit now
98              if(mars.getThreadCount(1) == 0 || mars.getThreadCount(2) == 0) break;
99          }
100
101         visualizer.anzeigen(mars.toCharArray(), i);
102
103         mars.setAction(-1);
104         mars.doNextAction();
105
106         System.out.println("Done!");
107
108     }
109
110 }
111
```