GREENFOOT: METEOR

Diese Anleitung ist für den Lehrer gedacht und beschreibt in Kurzform schrittweise, wie das Programm Stück für Stück erweitert wird. Die Schüler können jeden Schritt selbst mit Screenshots und Ausdrucken des Programm-Codes dokumentieren.

Einen Meteoriten steuern

(→ Meteor_02)

```
public class Meteor  extends Actor
{
   public void act()
   {
      this.moveLeft();
      if(this.getX()<1)
      {
         // Wir 'besorgen' uns die Welt mit this.getWorld()
         // Dann können wir nämlich alle Methoden der Welt aufrufen!
          this.getWorld().removeObject(this);
      }
   }

    public void moveLeft()
   {
      this.setLocation(this.getX()-3, this.getY());
   }
}
```

Die Welt zu Beginn mit einer Rakete und Meteoriten füllen

(→ Meteor_03)

```
public class MeteorWorld  extends World
{

   // Wir erzeugen ein Attribut, um mitzuzählen, wie oft die act-Methode
   // aufgerufen wurde. Attribute fangen immer mit private an.
   // Ganze Zahlen bezeichnet man in der Programmierung als Integer (hier: int)
   private int zaehler = 0;

   public MeteorWorld()
   {
      // Create a new world with 800x600 cells with a cell size of 1x1 pixels.
      super(800, 600, 1);
      this.fillWorld();
   }

public void fillWorld()
   {
      // Wir sagen dem Computer, dass wir ein Objekt der Klasse Rocket verwenden wollen
      Rocket meineRakete;

      // Jetzt erzeugen wir das Objekt meineRakete:
      meineRakete = new Rocket();

      this.addObject(meineRakete, 100, 300);

      // Wir fügen ein Objekt der Klasse Meteor ein:
      Meteor meteor1;
      meteor1 = new Meteor();
      this.addObject(meteor1, 800, 350);
   }
}
```

Damit zeitgesteuert immer neue Meteoriten erscheinen, diese Methode in der Klasse MeteorWorld einfügen:

```
public void act()
{
// Jedes mal, wenn die act-Methode aufgerufen wird,
// wird der zaehler um 1 nach oben gezählt!
zaehler = zaehler + 1;

// Nur alle 15 Millisekunden soll ein neuer Meteor erzeugt werden:
if(zaehler > 15)
{
 Meteor nochEinMeteor;
 nochEinMeteor = new Meteor();
 this.addObject(nochEinMeteor, 800, Greenfoot.getRandomNumber(600));
 zaehler = 0;
}
```

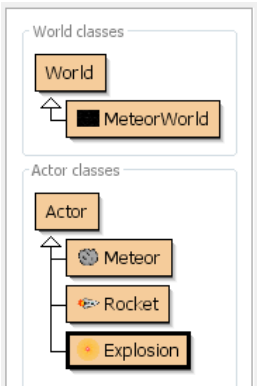
Kollision der Rakete mit einem Meteoriten

(→ Meteor_04)

```
public class Rocket extends Actor
{
    public void act()
    {
        if(Greenfoot.isKeyDown("down"))
        {
            moveDown();
        }

         if(Greenfoot.isKeyDown("up"))
        {
            moveUp();
        }

        checkCollision();
    }

    public void moveDown()
    {
      setLocation(getX(), getY() + 2);
    }

     public void moveUp()
    {
      setLocation(getX(), getY() - 2);
    }

    public void checkCollision()
    {
        Actor meinMeteor;
        meinMeteor = getOneIntersectingObject(Meteor.class);
        if(meinMeteor != null)
        {

          Greenfoot.stop();
          System.out.println("GAME OVER!");
        }
       else
       {
          // Tue nichts!
       }
    }
}
```

Explosion bei Kollision

(→ Meteor_05)

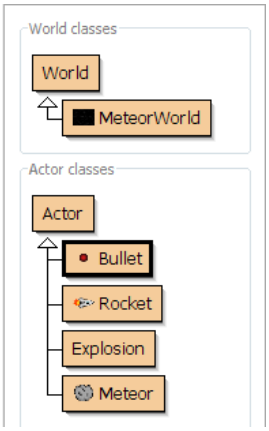


Eine neue Klasse *Explosion* einfügen (Bild: explosion.png). Dieses einfach im Greenfoot-Projekt in den Ordner *images* kopieren. Damit es bei der Explosion auch Geräusche gibt: *Explosion.wav* in den Ordner *sounds* kopieren!

```
public class Explosion extends Actor
{
   // Wir brauchen ein Attribut, um mitzuzählen, wie viele Milliskunden das
   // Explosions-Objekt schon sichtbar ist. Dieses Objekt soll nämlich nach
   // z.B. 100ms wieder verschwinden.
   int zaehler = 0;

   public Explosion()
   {
      Greenfoot.playSound("Explosion.wav");
   }

   public void act()
   {
      // Der zaehler wird jede Millisekunde um 1 nach oben gezählt:
      zaehler = zaehler +  1;

      // Nach 100ms soll die Explosion wieder verschwinden:
      if(zaehler > 100)
      {
         this.getWorld().removeObject(this);
      }
   }
}

public class Rocket  extends Actor
{
   ……
   public void checkCollision()
   {
       Actor meinMeteor;
      meinMeteor = this.getOneIntersectingObject(Meteor.class);

      if(meinMeteor != null)
      {
         Explosion meineExplosion;
         meineExplosion = new Explosion();

         this.getWorld().addObject(meineExplosion, this.getX(), this.getY());

         this.getWorld().removeObject(this);
      }
   }
}
```

Raumschiff kann feuern

(→ Meteor_06)



Eine neue Klasse Bullet einfügen (Bild: *other / beeper.png*).

```
public class Bullet  extends Actor
{
   public void act()
   {
      // Objekt soll sich nach rechts bewegen, wenn es sich noch nicht
      // am rechten Rand befindet. Ansonsten soll es verschwinden:
      if(this.getX() < 790)
      {
         this.setLocation(this.getX() + 5, this.getY());
      }
      else
      {
         this.getWorld().removeObject(this);
      }
   }
}
```

```
public class Rocket  extends Actor
{
   public void act()
   {
      …

       if(Greenfoot.isKeyDown("space"))
       {
         this.getWorld().addObject(new Bullet(), this.getX() + 60, this.getY());
       }

      this.checkCollision();
   }
```

```
public class Meteor  extends Actor
{
    public void act()
   {
      this.moveLeft();

      // Wenn die X-Koordinate des Meteors kleiner als 1 ist,
      // dann soll das Meteor-Objekt vernichtet werden
      if(this.getX() < 1)
      {
         this.getWorld().removeObject(this);
         // Die act-Methode muss abgebrochen werden, weil sonst noch die
         // checkCollision-Methode aufgerufen würde. Das Objekt gibt es ja aber
         //  nicht mehr! Abbruch mit return!
         return;
      }

      this.checkCollision();
   }
```

```
public void checkCollision()
{
   Actor meinBullet;
   meinBullet = this.getOneIntersectingObject(Bullet.class);

   if(meinBullet != null)
   {
     Explosion meineExplosion;
     meineExplosion = new Explosion();
     this.getWorld().addObject(new Explosion(), this.getX(), this.getY());

     // Das Meteor-Objekt wird aus der Welt entfernt:
     this.getWorld().removeObject(this);
   }

}
```

Die Munition begrenzen und Bonus-Munition einsammeln

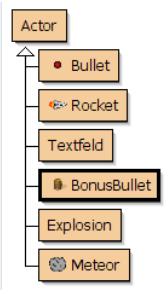
(→ Meteor_07)

Zuerst programmieren, dass nur 10 Schuss zur Verfügung stehen:

```
public class Rocket extends Actor
{

  // Attribut 'shots': Vorrat an Munition
   private int shots = 10;

   public void act()
   {
       ……

       if(Greenfoot.isKeyDown("space"))
       {
           if(shots > 0)
           {
             this.getWorld().addObject(new Bullet(), this.getX() + 60, this.getY());
             // Munition reduzieren:
             shots = shots - 1;
           }
      }
      checkCollision();
   }
```

Klasse *BonusBullet* erstellen (Bild: objects / barrel.png)!

```
public class BonusBullet  extends Actor
{
   public void act()
   {
      this.moveLeft();

      if(this.getX() < 1)
      {
        this.getWorld().removeObject(this);
        // act-Methode sicherheitshalber abbrechen:
        return;
      }
   }
```



```
    public void moveLeft()
    {
      this.setLocation(this.getX()-2, this.getY());
    }
}
```

```
public class MeteorWorld  extends World
{
    private int zaehler = 0;
    // Wir zählen mit, damit ein Objekt der Klasse BonusBullet erst erscheint,
    // nachdem 10 Meteoriten erzeugt wurden:
    private in bonusZaehler = 0;

    public MeteorWorld()
    {
        super(800, 600, 1);
        this.fillWorld();
    }

    public void act()
    {
        zaehler = zaehler +  1;

        if(zaehler > 15)
        {
            Meteor nochEinMeteor;
            nochEinMeteor = new Meteor();
            this.addObject(nochEinMeteor, 800, Greenfoot.getRandomNumber(600));
            zaehler = 0;

            // Pro erzeugtem Meteor wird der bonusZaehler um 1 nach oben gezählt:
            bonusZaehler = bonusZaehler + 1;
        }

        // Pro 10 Meteoriten soll es neue Munition zum Aufsammeln geben:
        if(bonusZaehler > 9)
        {
            BonusBullet meinBonus;
            meinBonus = new BonusBullet();
            this.addObject(meinBonus, 800, Greenfoot.getRandomNumber(600));

            bonusZaehler = 0;
        }
    }
```

```
public class Rocket extends Actor
{

  private int shots;

  public void act()
   {
      ……

      // Bonus-Munition aufgesammelt?
      this.checkBulletCollected();

      // Kollisionsprüfung muss auf jeden Fall ganz unten in der act-Methode stehen.
      // Sonst könnte es passieren, dass die Rakete vernichtet wird und noch etwas
      // anderes tun möchte (z.B. checkBulletCollected) => Absturz!
      this.checkCollision();
   }

   public void checkBulletCollected()
   {
     Actor meinBonus;
     meinBonus = this.getOneIntersectingObject(BonusBullet.class);

     if(meinBonus != null)
     {
       shots = shots + 5;
       this.getWorld().removeObject(meinBonus);
     }
   }
}
```

Damit beim Spiel noch Musik zu hören ist, müssen die entsprechenden Sound-Dateien in den Sounds-Ordner des Greenfoot-Projekts kopieren. In der Klasse MeteorWorld sind dann noch diese Änderungen erforderlich:

```
public class MeteorWorld extends World
{
   // Attribut isThereSound: Spielt die Musik schon?
   // Als Datentyp verwenden wir BOOLEAN: Damit kann man nur 'true' und 'false' speichern.
   private boolean isThereSound = false;

   public void act()
   {
      …….

      // Musik spielen!
      this.playMyMusic();
   }
```

```
    // Falls noch keine Musik spielt: Jetzt Musik spielen!
    // Die Musik wählen wir zufällig aus ('galactica.wav oder 'galactica2.wav')
    public void playMyMusic()
    {
       if(isThereSound == false)
         {
            int zufallszahl;
            zufallszahl = Greenfoot.getRandomNumber(2);

            if(zufallszahl == 0)
            {
               Greenfoot.playSound("galactica.wav");
            }
            else
            {
               Greenfoot.playSound("galactica2.wav");
            }
            isThereSound = true;
         }
       }
  }
```

Ein einfacher Counter: Den Spielstand anzeigen

(→ Meteor_08)

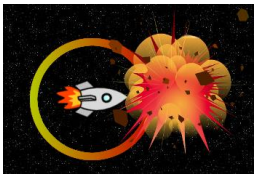
Im nächsten Schritt lassen wir uns den Spielstand anzeigen, indem wir einen einfachen Counter erstellen. Das geht in Greenfoot ziemlich leicht, weil Greenfoot eine eigene Methode zum Anzeigen von Texten auf dem Spielfeld hat, nämlich *showText()*. Diese Methode ist nicht besonders flexibel, wir können leider Schriftgröße und Farbe nicht festlegen. Dafür ist das Arbeiten mit dieser Methode aber sehr einfach.

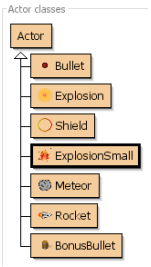
```
import greenfoot.*;  // (World, Actor, GreenfootImage, Greenfoot and MouseInfo)

public class Rocket extends Actor
{
   // Wir zählen den Punktestand mit:
   private int counter = 0;

   public void act()
   {
      …….

      // Den Counter um 1 nach oben zählen:
      counter = counter + 1;

      // Den Counter auf die Welt kleben. Problem: Im counter haben wir eine ganze Zahl gespeichert
      // (int). Mit der showText()-Methode kann man aber nur Texte ausgeben. Man muss also die
      // Integer-Zahl in einen Text umwandeln. Das geht mit: Integer.toString(IrgendeineZahl)
      this.getWorld().showText("Score: " + Integer.toString(counter), 700, 40);

      this.checkCollision();
   }
}
```

Die Rakete erhält einen Schutzschild

(→ Meteor_09)

Ab jetzt soll die Rakete über einen Schutzschild verfügen. Wenn ein Meteor-Objekt mit dem Schild zusammenstößt, wird der Meteor zerstört. Weil unsere bisherigen Explosionen ziemlich groß sind, erstellen wir eine neue Klasse mit dem Namen *ExplosionSmall*, die ein anderes und kleineres Bild einer Explosion verwendet (*explosion_klein.png*). Das hat rein optische Gründe, weil man sonst vor lauter Riesen-Explosionen nichts mehr erkennen würde.





Wir erstellen also eine neue Klasse *ExplosionSmall*. Der Programmcode in dieser Klasse ist grundsätzlich der gleiche wie in der Klasse *Explosion*. Einzige Änderung: Die kleine Explosion soll schneller wieder verschwinden.

```
public class ExplosionSmall extends Actor
{
   // Wir brauchen ein Attribut, um mitzuzählen, wie viele Milliskunden das
   // Explosions-Objekt schon sichtbar ist. Dieses Objekt soll nämlich nach
   // z.B. 100ms wieder verschwinden.
   private int zaehler = 0;

   public ExplosionSmall()
   {
      Greenfoot.playSound("Explosion.wav");
   }

   public void act()
   {
      // Der zaehler wird jede Millisekunde um 1 nach oben gezählt:
      zaehler = zaehler +  1;

      // Nach 30ms soll die Explosion wieder verschwinden:
      if(zaehler > 30)
      {
         this.getWorld().removeObject(this);
      }
   }
}
```

Jetzt erstellen wir eine Klasse *Shield* (Bild: shield.png). In dieser Klasse schreiben wir erstmal keinen Programmcode. Der Schild soll also noch nichts tun, nur gut aussehen, wenn wir ihn erscheinen lassen. Das Erscheinen eines Shield-Objekts programmieren wir in der Klasse Rocket (wenn der Spieler die Enter-Taste drückt).

```
public class Rocket extends Actor
{
   …

   public void act()
   {
      …
      if(Greenfoot.isKeyDown("enter"))
      {
          Shield meinShield;
          meinShield = new Shield();
          this.getWorld().addObject(meinShield, this.getX(), this.getY());
      }
   }
```

Im nächsten Schritt müssen wir den Schutzschild dazu bringen, die Meteor-Objekte zu zerstören, wenn Sie mit ihm zusammenstoßen. Wir müssen also in der Shield-Klasse eine Kollisionsabfrage programmieren. Zum Glück funktioniert die völlig analog zu den anderen Kollisions-Abfragen, die wir bisher programmiert haben. Wer jetzt über den Programmcode überrascht ist, hat bisher nur abgeschrieben und dabei das Gehirn ausgeschaltet ;-)

```
public class Shield extends Actor
{
   public void act()
   {
       this.checkCollision();
   }


   public void checkCollision()
   {
      Actor meinMeteor;
      meinMeteor = this.getOneIntersectingObject(Meteor.class);

      if(meinMeteor != null)
      {
```

```
            ExplosionSmall meineExplosion;
            meineExplosion = new ExplosionSmall();

            // Wir 'besorgen' uns die Welt mit this.getWorld()
            // Dann können wir nämlich alle Methoden der Welt aufrufen!
            this.getWorld().addObject(meineExplosion, meinMeteor.getX(), meinMeteor.getY());

            // Entfernt den Meteor aus der Welt
             this.getWorld().removeObject(meinMeteor);
        }
    }
}
```

Jetzt gibt es noch ein echtes Problem: Das Shield-Objekt müsste sich mit der Rakete bewegen. Das heißt: Das Shield-Objekt muss auf die *getX()*- und die *getY()*-Methode der Rakete zugreifen können, um zu wissen, wo sich die Rakete gerade befindet.

Die gute Nachricht: Das ist hier sehr einfach. Weil das Shield-Objekt ja genau da erzeugt worden ist, wo sich die Rakete befindet, überschneiden sich beide Objekte. In diesem Fall können wir mit der *getOneIntersectingObject*-Methode auf die Rakete zugreifen und deren Methoden benutzen.

```
public class Shield extends Actor
{
    public void act()
    {
        this.moveWithRocket();
        this.checkCollision();
    }

    public void moveWithRocket()
    {
        // Wir wollen auf die Methoden der Rakete zugreifen. Deshalb speichern
        // wir die Rakete im Objekt 'dieRakete' ab.
        Actor dieRakete;
        dieRakete = this.getOneIntersectingObject(Rocket.class);
        this.setLocation(dieRakete.getX(), dieRakete.getY());
    }
```

Der Schutzschild verschwindet und muss sich wieder aufladen

(→ Meteor_10)

Unser Spiel ist gerade ziemlich langweilig geworden, weil wir durch den Schutzschild unverwundbar sind. Das ändern wir jetzt: Nach 500ms soll der Schutzschild wieder verschwinden und muss sich erst neu aufladen. In einem ersten Schritt sorgen wir für das Verschwinden des Schilds.

```
public class Shield extends Actor
{
    // Das Objekt soll nur eine bestimmte Zeit vorhanden sein. Wir zählen also mit und brauchen ein Attribut:
    private int zaehler = 0;

    public void act()
    {
        // Der zaehler wird jede Millisekunde um 1 nach oben gezählt:
        zaehler = zaehler +  1;

        // Nach einiger Zeit soll der Schild verschwinden:
        if(zaehler > 500)
        {
            this.getWorld().removeObject(this);
            // act-Methode abbrechen, weil das Shield-Objekt nicht mehr existiert:
            return;
        }

        this.moveWithRocket();
        this.checkCollision();
    }
```

Jetzt soll sich der Schild wieder aufladen. Erst wenn 1000 act-Methoden vergangen sind, soll man den Schild wieder aktivieren können.

Lösungsidee: Wir erstellen ein Attribut *shieldCounter*. Der shieldCounter wird in jeder act-Methode um 1 nach oben gezählt. Sobald der shieldCounter größer als 1000 ist, soll zusätzlich auf dem Bildschirm rechts oben die Meldung „Shield Ready“ erscheinen. Wenn wir die Enter-Taste drücken, wird zusätzlich noch geprüft, ob shieldCounter > 1000. Weil also einiges passieren muss, lagern wir der Übersichtlichkeit halber nach Möglichkeit alles in eine eigene Methode aus (Name: *shieldCounterUpAndMessageOnScreen*).

```
public class Rocket extends Actor
{
   // Nachdem der Schutzschild seine Energie verbraucht hat, zählen wir mit.
   // Nach einer bestimmten Zeit ist der Schild wieder aufgeladen. Als Startwert
   // legen wir 1000 fest, damit wir gleich am Anfang einen Schild haben.
   private int shieldCounter = 1000;

   public void act()
   {
      …
      if(Greenfoot.isKeyDown("enter") && shieldCounter > 1000)
      {
         Shield meinShield;
         meinShield = new Shield();
         this.getWorld().addObject(meinShield, this.getX(), this.getY());

         // Wir setzen den shieldCounter auf 0, damit man erst nach einiger Zeit wieder
         // einen Schild aktivieren kann:
         shieldCounter = 0;
      }

      // shieldCounter um 1 nach oben zählen und Bildschirmausgabe "Shield Ready":
      this.shieldCounterUpAndMessageOnScreen();

      this.checkCollision();
   }

   public void shieldCounterUpAndMessageOnScreen()
   {
      shieldCounter = shieldCounter + 1;

      if(shieldCounter > 1000)
      {
         this.getWorld().showText("Shield Ready", 100, 40);
      }
      else
      {
         this.getWorld().showText("", 100, 40);
      }
   }
}
```

UND-Verknüpfung
Wenn man in JAVA zwei Bedingungen prüfen möchte, muss man diese mit **&&** verknüpfen!