# **The Java Game Development Tutorial**

After a lot of hard work, we have made it to finish the English version of the tutorial. As I said, you should be already able to program in Java, although you don't have to be a expert! The tutorial will give you a very detailed introduction into the development of online games using Java. We will talk about Java Applets because everyone has a Internet browser, so everyone is able to play these games without installing the JDK! The first 5 chapters will lead you from the basic structure of a applet to the development of a whole (although pretty small) game. The next chapters (6 - ...) will give you solutions to problems like AI, level editors, random generation of landscapes... . At the end of every chapter you'll find a link to download the sourcecode of the applet we programmed in this chapter and you can take a look at the working applet too. Alright, here we go!

1. [Basic structure of an applet](http://javaboutique.internet.com/tutorials/Java_Game_Programming/GrundstrukturAppletEng.html)
2. Animation
   1. [Move a ball](http://javaboutique.internet.com/tutorials/Java_Game_Programming/BallBewegungEng.html)
   2. [Double buffering](http://javaboutique.internet.com/tutorials/Java_Game_Programming/BildschirmflackernEng.html)
   3. [Ball bouncing and change the direction](http://javaboutique.internet.com/tutorials/Java_Game_Programming/BallStopEng.html)
3. Sound and Gif's
   1. [Using sound in applets](http://javaboutique.internet.com/tutorials/Java_Game_Programming/SoundinAppletsEng.html)
   2. [Using pictures in applets](http://javaboutique.internet.com/tutorials/Java_Game_Programming/EinfuegenvonBildernEng.html)
4. Event handling in applets
   1. [Mouse events](http://javaboutique.internet.com/tutorials/Java_Game_Programming/MausereignisseEng.html)
   2. [Keyboard events](http://javaboutique.internet.com/tutorials/Java_Game_Programming/TastaturereignisseEng.html)
5. The first complete game
   1. [Outline of the program and structure of classes](http://javaboutique.internet.com/tutorials/Java_Game_Programming/DasErsteSpielEng.html#outline)
   2. [Random movement of the ball](http://javaboutique.internet.com/tutorials/Java_Game_Programming/DasErsteSpielEng2.html#random)
   3. [Hit a ball by clicking on it](http://javaboutique.internet.com/tutorials/Java_Game_Programming/DasErsteSpielEng2.html#hit)
   4. [Count score and loose lifes](http://javaboutique.internet.com/tutorials/Java_Game_Programming/DasErsteSpielEng2.html#count)
   5. [Change the mouse pointer](http://javaboutique.internet.com/tutorials/Java_Game_Programming/DasErsteSpielEng2.html#change)
   6. [Start the game after double clicing on the applet](http://javaboutique.internet.com/tutorials/Java_Game_Programming/DasErsteSpielEng3.html)

# **Basic structure of an applet**

Well, this is very simple and the basic structure of an applet will be the same for every applet you will program. The major difference between an applet and an application is that there is no public static void main (String[]args) - method in an applet! Instead every applet should implement the following methods:

// import necessary packages  
import java.applet.\*;  
import java.awt.\*;  
  
// Inherit the applet class from the class Applet  
public class FirstApplet extends Applet  
{

// Now you should implement the following methods   
  
// init - method is called the first time you enter the HTML site with the applet  
public void init() {... }  
  
// start - method is called every time you enter the HTML - site with the applet  
public void start() {... }  
  
// stop - method is called if you leave the site with the applet  
public void stop() {... }  
  
// destroy method is called if you leave the page finally (e. g. closing browser)  
public void destroy {... }  
  
/\*\* paint - method allows you to paint into your applet. This method is called e.g. if you move your browser window or if you call repaint() \*/  
public void paint (Graphics g) { }

}

To insert an applet to a HTML - site you have to add the following lines to your HTML document.

<html>  
<body>  
<p><applet code ="FirstApplet.class" width=700 height=400>  
</applet></p>  
</body>  
</html>  
  
This is the most important line: <p><applet code ="FirstApplet.class" width=700 height=400> .

1. applet code ="FirstApplet.class" tells the browser where to find the "extends Applet" - class.
2. width and height tell the browser how big the applet should be

To learn other html - tags please search for a tutorial in the internet.

# **How to move a ball**

We want to start with a very essential step. We will program an applet in which a ball is moving from the left to the right hand side. I know this is nothing BIG but if you want to learn how to program games it is very important to understand how to animate objects!  
  
At the beginning we have to write our basic structure of an applet again but we will add two little things. Our applet has to implement the interface Runnable and the corrosponding method run() to animate an object. The structure of the applet should look like this:

import java.applet.\*;  
import java.awt.\*;  
  
public class BallApplet extends Applet *implements Runnable*  
{

public void init() { }  
  
public void start() { }  
  
public void stop() { }  
  
public void destroy() { }  
  
*public void run () { }*  
  
public void paint (Graphics g) { }

}

Threads  
A thread is a piece of program that is able to run parallel to other parts of the program (multithreading). Threads are implemented by the class Thread, the interface Runnable and the method run(), we have already implemented these two things in the step before. Important methods of the class Thread are:

* Thread.start(): starts a thread
* Thread.stop(): stops a thread
* Thread.sleep(time in milliseconds): stops thread for a certain amount of time

You can find more functions of the thread class in the Java API!   
  
And here comes the code!

To move a object we need another object that has to be an instance of the class Thread; we declare this object in the start - method of our applet:

public void start ()  
{

// define a new thread   
Thread th = new Thread (this);  
// start this thread  
th.start ();

}

Now this thread is running in the run() - method of our applet. Every time all methods... in the run - method have been called, we stop the thread for a short time. Your run method should look like this:

public void run ()  
{

// lower ThreadPriority   
Thread.currentThread().setPriority(Thread.MIN\_PRIORITY);  
  
// run a long while (true) this means in our case "always"  
while (true)  
{

// repaint the applet  
repaint();  
  
try  
{

// Stop thread for 20 milliseconds  
Thread.sleep (20);

}  
catch (InterruptedException ex)  
{

// do nothing

}  
  
// set ThreadPriority to maximum value  
Thread.currentThread().setPriority(Thread.MAX\_PRIORITY);

}

}

What we have now is a neverending loop that executes all things within the loop, waits 20 milliseconds and executes everything once again and so on. But how can we move a circle that is painted by the applet?

Well this is a very simple idea: Our circle has a x - and a y - position. If we were to add 1 to the x - position everytime the thread is executed, the ball whould move across the applet, because it is painted at a different x - position everytime we execute the thread!  
  
Ok, let's start with drawing a circle: Add these lines to the paint - method of the applet:

public void paint (Graphics g)  
{

// set color  
g.setColor (Color.red);  
  
// paint a filled colored circle  
g.fillOval (x\_pos - radius, y\_pos - radius, 2 \* radius, 2 \* radius);

}

And we need the following instance variables at the head of the program:  
  
int x\_pos = 10;  
int y\_pos = 100;  
int radius = 20;  
  
To move a ball we change the value of the x\_pos variable everytime the thread is executed. Our run - method should look like this:

public void run ()  
{  
...

while (true)  
{

*// changing the x - position of the ball/circle  
x\_pos ++;*  
...

}

}

If you add this applet to an HTML document as seen in the previous chapter, a red ball should be moving across the applet one time!

# **Double buffering**

I'm sure you recognized in the first applet that the circle is flickering. There is a very simple reason for this. Everytime the paint - method is called by repaint() the applet screen is cleared completly. Because of this we can see for a millisecond a absolutly blank screen. To suppress this phenomena, we have three possibilities:

1. We don't clear the screen at all
2. We clear the screen only where something is changing
3. We use double buffering

#### **Don't clear the screen at all**

This idea seems to be the solution to all our problems (but it is not!). In our case it would mean that the ball would paint a thick red line across the applet because the screen stays red at every position the ball has been once. This might be ok for some situations but we want to see a moving ball, not a thick red line. So this technique is just useful for objects that are not moving.

At first it is important for you to know one thing. The call of repaint() doesn't call the paint() - method at the same time. Instead, a method called update() is called. If one doesn't overwrite this method, update() clears the complete screen and afterwards calls paint() which paints the background and our circle again. To avoid clearing the screen you have to overwrite the update() - method. Our new update() - method doesn't clear the screen anymore, but just calls paint(). This can be done with three line of code:

public void update(Graphics g)  
{

paint(g);

}

As I said before this is no solution for our applet. But it is essential to understand that repaint() doesn't call paint() but update() which then calls paint()!

#### **Clear the screen only where something is changing**

This solution is based on the idea to repaint just those parts of our applet where something has changed. This concept is very good for a game like snakin'. If the last part of your snake is colored in the same color as the background, this part overpaints the parts of the snake, where the snake has been. I don't want to talk about this solution in detail, because one sould use this method just in very special situations. So let's talk about the double buffering which is a really good and effective solution to avoid a flickering screen and the best of it all: You can use this method in every applet the same way as I do now, so you'll never have to worry about that problem again!

#### **Double buffering**

As I said you can use double buffering in every applet in a very easy way. Double buffering means to paint all the things in the paint() method to an offscreen image. If all things that have to be painted, are painted, this image is copied to the applet screen. The method does the following in detail:

1. Generates a new offscreen-image by using createImage and stores this image in a instance variable( = generate an empty image)
2. Call getGraphics to get graphic context for this image
3. Draw everything (including to clear the screen completely) on the offscreen image ( = draw image in the background)
4. When finished, copy this image over the existing image on the screen. ( = draw image in the foreground)

This technique means that the picture is already painted before it is copied to the screen. When copiing the image to the foreground the old pixels are overlayed by the new ones. There won't be any flickering picture anymore because there is not a millisecond where you see an empty screen!

The only disadvantage of the double buffering is, that it produces a large amount of data and every image is drawn two times (offscreen and when copying to the screen). But in most cases and on a fast computer this is much better than wasting time on finding an other solution!

Well after all this theory I will show you how to integrate the double buffering into our "ball moving" applet (use the [applet](http://javaboutique.internet.com/tutorials/Java_Game_Programming/Applets/Ballbewegung1/BallBewegung1Applet.html) of the chapter before as basic code!):

##### ***Double buffering: the code***

// declare two instance variables at the head of the program  
private Image dbImage;  
private Graphics dbg;  
  
... other code ...  
  
/\*\* Update - Method, implements double buffering \*/  
public void update (Graphics g)  
{

// initialize buffer  
if (dbImage == null)  
{

dbImage = createImage (this.getSize().width, this.getSize().height);  
dbg = dbImage.getGraphics ();

}  
  
// clear screen in background  
dbg.setColor (getBackground ());  
dbg.fillRect (0, 0, this.getSize().width, this.getSize().height);  
  
// draw elements in background  
dbg.setColor (getForeground());  
paint (dbg);  
  
// draw image on the screen  
g.drawImage (dbImage, 0, 0, this);

}

As I said before you can copy and paste this code into every applet that uses animations!

# **Bounce a ball**

Well Ok we are now able to move a ball across the applet without any flickering of the screen. This means that we know the two most important steps to program a game with animated objects. In this chapter I want to tell you how to change the movement direction of the ball. This is the first time that we can't solve a problem by a "standard" technique, as we could do in the chapters before (Double buffering...). But I'm sure if you understand the solution, you will have your own ideas once you are in a similar situation. In our last two applets, the ball is moving across the applet area just one time. But now we want to avoid, that the ball is leaving the applet area, instead the ball should bounce in the opposite direction if it hits a wall. What shall we do? It would be good, if you would stop reading now and try to find a solution by yourself. In the following chapter you'll get to know my solution (which is not the only one!). Please use the "Ball" - applet with double buffering as the basic code for this chapter!

##### ***How to move a ball in different x - directions***

First of all we have to think of the technique we used to move the ball in the second chapter. What did we do? Well, we increased the x - position of the ball every time the thread was executed and and repainted the ball at the new position. Maybe you didn't realize it, but we used a "speed vector" for the x - direction of the ball, everytime we changed the x - position of the ball by calling *x\_pos ++* in the run - method. That means that we can also define a variable called *x\_speed* that stores the value which will be added to x\_pos by calling now *x\_pos += x\_speed*. Imagine, x\_speed has the value 1, this means, that the ball is moving from the left to the right hand side (same direction as in the chapter before), because x\_pos will be increased by one everytime the run - method is executed. Now it is really easy to change the direction of the ball. We only have to set the value of x\_speed to -1! This means, that everytime we call *x\_pos += x\_speed* -1 is "added" to the x - position of the ball. That way the value of x\_pos is decreased by one and the ball is moving from the right to the left hand side!

Now we are able to change the direction of the ball! And there is just one "problem" left! How can we (the computer) decide, when to change the direction? Well, we have to test everytime we execute the run method, at what x - position our ball is. The code to do this looks like this:

// Ball is bounced if its x - position reaches the right border of the applet   
if (x\_pos > appletsize\_x - radius)  
{

// Change direction of ball movement  
x\_speed = -1;

}  
// Ball is bounced if its x - position reaches the left border of the applet  
else if (x\_pos < radius)  
{

// Change direction of ball movement  
x\_speed = +1;

}

Everytime we execute the run - method we test if the ball has reached one of the borders (left or right). If this is the case, we change the value of x\_speed and that way we are able to change the direction of the ball movement.

You might be asking why I use the radius of the ball in the code above. The x - position of the ball is its middle point, which means that the left or the right border of the ball is its x - position plus or less the radius. To make us believe that the ball would be bouncing when it hits the wall, we have to add or substract the radius to / from the the value of x\_pos.

##### ***How to "beam" the ball***

Once you understand the technique of how to test where the ball is, you can implement many other things. Imagine we want to let the ball appear on the other side of the applet once the ball has left the applet area. That's very easy to do, we have to test again, where the ball is and then we have to set a new x - position for the ball instead of changing the value of x\_speed.

// Test if ball has left applet area if (x\_pos > appletsize\_x + radius)  
{

// Set a new x\_pos value for the ball  
x\_pos = -20;

}

Later, in chapter 5, I will talk about how to move a ball not only in a one dimensional way but in a two dimensional area (which means in x and y direction). That is really easy to do. You will have not only one vector (x\_speed) but two (x\_speed, y\_speed), and a y\_pos and a x\_pos. If one changes both values everytime run() is called the ball will be moving over the whole area. But of course we have to test 4 different possibilities, where the ball could leave the applet. If you want to, you can program an applet where the ball can move in different directions (2D) and is bouncing from every border of the applet (would be a good practice ;-). Ok, that's it, take a look at the applets and download the code!

# **Using sounds in applets**

Sound files make a game much more exiting and they are really easy to "add" to a applet! It will take you just a few lines of code. In this chapter I want to show you, with the help of the "ball bounce" applet (chapter 1 c), how to add sounds in \*.au Format; Java is not able to handle \*.wav - files! You should hear this sound every time the ball is bouncing from one of the walls.

At first we need an object reference to an AudioClip object, thats why we declare an instance variable AudioClip called bounce.

// Instance variable AudioClip bounce   
*AudioClip bounce;*

For some reason I don't know, I get a strange error message in InternetExplorer 5 if this variable is public or private. Thats why I declare it protected here.

Now we have to load the sound file to our applet (you have to import java.applet.\* and java.net.\* to do this!). Write the folliwing lines into the init() - method of your applet:

// Load an audio file which is in the same directory as the class files of the applet   
*bounce = getAudioClip (getCodeBase(), "bounce.au");*

You can play this file by writing *bounce.play()* wherever you need to play it in this applet.

If the sound file is placed in another directory, (for example audio), than the class files, you can load it by writing *bounce = getAudioClip (getCodeBase(), "audio/bounce.au");* laden. Unfortunatly you can't use \*.wav - files which are much more common. But you can turn \*.wav files into \*.au files using a wave editor (for example the shareware program GoldWave). Additionally these files have to be 8 Bit, 8000 khz, Mono files. All this is a little bit complicated, but you can find many Java games in the internet and use their sound files at first.

# **Using pictures in applets**

Just like "adding" sound files to an applet you can add pictures. Now I want to explain to you how to do this by adding a background picture to our "ball bounce" applet. Again we have to declare an instance variable, an *Image* object called backImage:

*Image backImage;*

Then we load a picture to our applet and store it in the object reference backImage (import java.applet.\* and java.net.\*!!). This works very similar to loading sound files, please add these lines to your init() - method:

*// load file Land.jpg   
backImgage = getImage (getCodeBase (), "Land.jpg");*

As a last step we have to paint this picture in the paint() - method of the applet:

*// drawing the image: g.drawImage (name, x- value, y - value, frame)  
g.drawImage (backImage, 0, 0, this);*

That's it! I used a \*.jpg - file in this example but that is not that good. Normally you should use \*.gif - files, because their size is not that big, (makes the applet run faster), and they are more common in the internet. Another reason to use \*.gif - files is that you can define the background of a \*.gif - file transparent. So you can use them to overlay for example a ball object (which is just a colored circle at the moment) with a real good looking picture. You can see a example for this in the game [Streethockey](http://www.javacooperation.gmxhome.de/Applets/Games/StreetHockey/StreetHockeyEng.html) and maybe there will be a "advanced topic" for this technique someday!

# **Handle mouse events**

I think there is no game that doesn't "interact" with the player. In our case, as a programmer, you have two possibilities to allow the player to interact with the computer: the player has to use the mouse or the keyboard. In the next two chapters I want to tell you how to handle mouse and keyboard events in an applet. Again this is much easier to do in an applet than in a real application programmed for example using Java Swing. Also you won't be able to use the things you will learn here about event handling in an application, because event handling is very different in applications. But every good book about Java tells you how to handle events in applications so I think it will be no problem for you to learn it by yourself if you need to.

Now I want to change our "ball movement" applet so that the ball changes the direction if the user clicks on the applet.

To do this you have to implement just one method in our Main class called *mouseDown*. This method should look like this:

*// Method to handle mouse down events   
public boolean mouseDown (Event e, int x, int y)   
{*

*// Change direction   
x\_speed = - (x\_speed);   
  
// DON'T FORGET (although not necessary here)!!   
return true;*

*}*

Of course you can handle other events than the mouseDown event. Java makes it possible to overwrite the following methods the same way as I did above and handle other mouse events that way. Here comes a list of the methods:

1. Mouse click events:
   1. public boolean mouseDown (Event e, int x, int y): handles events that occur if mouse button is pressed down
   2. public boolean mouseUp (Event e, int x, int y): handles events that occur if mouse button is released again.
   3. Use the variable e.clickCount to get the number of clicks performed. That's how you can handle for example double clicks! You'll see a example for this in chapter 5 ("Our first game").
2. Mouse movement:
   1. public boolean mouseMove (Event e, int x, int y): handles events that occur if mouse is moved over the applet
   2. public boolean mouseDrag (Event e, int x, int y): handles events that occur if mouse is moved over the applet with pressed mouse button
3. Mouse position:
   1. public boolean mouseEnter(Event e, int x, int y): handles events that occur if mouse enters the applet
   2. public boolean mouseExit (Event e, int x, int y): handles events that occur if mouse leaves the applet

With the help of the variables int x and int y you can get the coordinates where the event happened, which is very important for many games. Also you have to take care of the return statement "return true;" that often has no meaning for the functionality of your game but is expected by the method.

# **Event handling of key events**

Handling key events in applets works very similar to the mouse event handling I talked about in the chapter before. There is one special method for every possible event (key down, key up). You have to overwrite the corrosponding method, if you want to handle, for example, a key down - event. In this chapter we'll have a short look at these two methods. Afterwards we will change our "moving ball" applet that way, so that one can change the direction of the ball movement by pressing the left or right cursor key and stop the ball movement by pressing the space bar. Be carefull! An applet is only able to handle key events once you have clicked in the applet window!

Here comes the explanation of the methods:

1. public boolean keyDown (Event e, int key): This method listens to events that occur if a key is pressed down.

Every key has a value (ASCII). This value is given to the method with the help of the "key" variable. Space bar has a value of 32. If you want your applet to listen to Space bar pressed down, you only have to test, if the value of "key" is 32 (details later). Sometimes you don't know which value a certain key has. Then you can print out this value to the standard output by writing this line in your keyDown(...) - Method:   
  
*System.out.println ("Charakter: " + (char)key + " Integer Value: " + key);*   
  
Some frequently used keys have special variables. For example the cursor keys have Event.LEFT, Event.RIGHT, Event.UP, Event.DOWN. These values are field variables in the class Event. For more variables please read the API!

1. public boolean keyDown (Event e, int key): This method reacts to key up - events. You can use this method exactly the same way as the keyDown - Method!

Now we want to modify our "moving ball" applet a little bit. The user should be able to change the direction of the ball movement by pressing the left or right cursor key. If he presses the space bar, the ball should stop its movement. To make this possible, we have to add the following lines to our applet:

*// method to handle key - down events   
public boolean keyDown (Event e, int key)   
{*

*// user presses left cursor key   
if (key == Event.LEFT)  
{*

*// changing x - speed so that ball moves to the left side (x\_speed negative)  
x\_speed = -1;*

*}  
// user presses right cursor key   
else if (key == Event.RIGHT)  
{*

*// changing x - speed so that ball moves to the right side (x\_speed positive)  
x\_speed = 1;*

*}  
// user presses space bar (value = 32!)  
else if (key == 32)  
{*

*// Stop ball (x\_speed = 0)  
x\_speed = 0;*

*}  
else  
{*

*/\* Additionally the method prints out the ASCII - value if an other key is pressed. This is not necessary but a possibility for you to test which value a key has.\*/   
System.out.println ("Charakter: " + (char)key + " Integer Value: " + key);*

*}  
  
// DON'T FORGET (although it has no meaning here)   
return true;*

*}*

Now you are able to program every essential thing used in a game. You can move objects, handle events and load sounds and images. In the next chapter we will program our first complete game (which was the first game I programmed in Java). To understand the next chapter you have to know something about classes, objects, calling methods of other classes... . And of course you have to know everything I talked about in the last chapters. By programming this game I will show you some new techniques; (how to "hit" a ball, changing mouse pointer, random ball movement...). After programming this game you should be able to program your own games. Well Ok have fun in the next chapter and now you can watch the applet you programmed in this chapter and download the source code!

# **Our first game**

### **The game idea**

In our game, the player has to shoot at two balls, (a red and a blue one), which are flying around in our applet. The direction and speed will be chosen at random; the user can hit a ball by clicking on it with the mouse pointer. If a ball reaches a border of the applet without being hit, the player looses one of his 10 lives. If the player loses all of his lives, the game is over.

### **Outline of classes and methods**

##### ***The Main class***

This class implements all methods concerning the animation of objects (balls) and "management" of the game. These are init(), start(), stop(), paint(), update(), run(), and mouseDown(), to handle mouse click events. The class also manages all objects, (two ball objects, the player object), and the thread in which the game is running!

1. init(): Initializes all objects, gets audio data, sets mouse pointer and gets the game speed from a parameter in the HTML - file.
2. start(): starts thread
3. stop(): stops thread
4. run(): moves the balls
5. paint(...): calls paint methods of the both balls, paints score and life information to the screen. If the game is over it writes out an information text about how well the player has been playing (for example: "You are the champion")
6. update(...): Implements double buffering
7. mouseDown (...): Listens to mouse events. If the game is running it is watching out for "player hits ball"; if the game is stopped, (start or over), then it listens for a double click and starts the game (again).

##### ***The class Player***

This class is very simple. It has two instance variables to store, the score and the lives of the player. There are also two methods to add score, (addScore(int plus)), loose lives, (looseLife) and two methods to transmit these values to the Main class (getScore(), getLifes()). As you see, a pretty simple class.

##### ***The class Ball***

This class is the most complicated one in the game. It implements all of the important methods for a ball object including the following functions:

1. Ball(...): Constructor, gets all important attributes of a ball object (color, speed, y - direction, ...) and initializes variables in the ball object.
2. move(): Method moves the ball and tests by calling isOut() if ball is out which means, player loses one life.
3. ballWasHit(): This method is called if the ball was hit by the player. Afterwards the new direction of the ball is set at random, and the ball is set back to the start position.
4. userHit (int x, int y): This method tests with the help of vectors, if ball was hit or not (details later).
5. isOut(): this method tests if the ball is out. The player loses one life if this is true.
6. DrawBall(Graphics g): This method draws the ball and is called from the paint method of the Main - class.

##### ***How the methods work together:***

Everytime we call the run - method in our Main class, we call the move - methods of the ball. This method moves the ball and tests if the ball left the applet area by calling isOut(). If the ball has left the applet area, isOut() calls the looseLife() method of the player object and the number of lives is decreased by one.

Everytime the user clicks in the applet area, the mouseDown method in the Main class is called. This method calls userHit(), which is testing if the ball was hit by the user and returns true if this is the case and returns false if the ball was not hit. If true is returned by userHit() ballWasHit() is called. This method adds the score to the player score, and resets the ball... .

##### ***One important thing before we start***

It is essential, if you want to understand what I'm doing, that you download the [sourcecode](http://javaboutique.internet.com/tutorials/Java_Game_Programming/SourceCodes/ErstesSpiel/DasersteSpiel.zip) and read it while I'm explaining things. I won't explain every step in detail, instead you should read the sourcecode, try to understand those things I'm not talking about with help of the other chapters, and learn things by yourself that way.

Even more important might be that you begin to write your own little programs very soon and try to solve problems on your own. Because every game needs its own, very special solutions and mostly there will be no one to help you to find these. Use this little game and all the other applets and sourcecodes on these pages as blueprints and "dictionarys" which help you to implement your own games. You'll learn most if you try it by yourself!

### **Randomize the movement direction of the two balls**

As I said in the second chapter, we will move our ball in this game not only in the x - direction but in the y - direction, too. To do this we have to add a y\_speed vector to the applet as well as the x\_speed vector you already know. Our ball will have an x\_pos variable which will be changed with every call of move() by adding the value of x\_speed to it; (remenber: x\_speed can be negative), and our ball will also have a y\_pos variable which will be changed by adding y\_speed. In our game the value of y\_speed will not be changing. The first ball will be moving up (y\_speed = -1) and the other ball will be movind down (y\_speed = 1). Be careful, in Jave the y - coordinates get bigger when moving down!! The value of x\_speed will be chosen at random everytime a ball left the applet.

##### ***Random number generator***

First of all we need a random generator in Java. We have to import the class *java.util.\*;* into the class Ball and declare the following instance variable afterwards:

*Random rnd = new Random ();*

Now we can generate a random integer number by calling *rnd.nextInt()*. Because we want to have values between -3 and +3 in our game, we have to calculate this random number mod 4 afterwards.

Now we can randomize the movement direction of the ball. We have to set the value of x\_speed to a new random value, everytime the ball is hit or the ball leaves the applet area.

##### ***The move() - method of the ball object***

First of all we have to declare four instance variables (x\_speed, y\_speed, x\_pos, y\_pos) and initialize them in the constructor of our ball object. Everytime move() is called we add the value of x\_speed to the x\_pos variable and the value of y\_speed to the y\_pos variable. Everytime the ball is hit or has left the applet, we initialize x\_speed with a random generated value, y\_speed will be constant. Here comes the code:

*public void move()  
{*

*// Add values of y and x\_speed to x and y position  
pos\_x += x\_speed;  
pos\_y += y\_speed;  
  
/\* Call isOut() and test if ball is still within the applet area \*/  
isOut();*

*}*

##### ***The method isOut()***

This method tests if a ball has left the applet. If one of the four if - statements is true, the position of the ball will be reset, an audio file will be played, the player loses one life and x\_speed will get a new value (per random).

The tests of this method are quite the same as in chapter 2c, but there are four borders we have to test rather than two. But the technique stays the same and you should try to understand this method by yourself; with help of chapter 2c this should be no problem for you.

### **How to hit a ball**

This game was the first one I'd ever programmed and the problem of how to hit a ball by clicking on it caused me a lot of headaches. Now I will try to explain my solution to you: Our ball has an x and a y position. The mouseDown - method in the main class, which calls userHit (int x\_mouse, int y\_mouse) gives the coordinates, where the user clicked on the applet, to the method userHit(). Now, our method has to decide if a ball has been hit or not. But how can we do this?

It is not enough to test if the x and y position of the ball is exactly the same as the position where the player clicked on the applet! This would mean that the player has to hit *exactly* in the middle of our ball, something that is almost impossible when the ball is moving!

Ok, second idea: We accept all mouseDown coordinates as a hit that are in a certain range around the exact position of our ball. Of course the x and the y position have to be in this range at the same time (&& - test). That way we would test if the player hit into a rectangle around the ball (for example if the ball has the radius 10, this rectangle should be 20 pixles each side). This was the first idea I tried to implement but it didn't work out as well as I thought it would. The whole thing worked sometimes and sometimes not, even if I hit the ball exactly! I don't even have the slightest idea why it didn't work .

Well, alright, third idea: I think, everyone has heard of vectors. In our game we have two vectors: one is the x and y position of the ball and the other one is the x and y position of the mouse click. If we calculate the length of the connection vector of these two vectors and if the length of this connection vector is smaller than the radius of the ball, the player has hit the ball.

First of all we have to calculate the connection vector:

*// Calculate connection vector  
double x = maus\_x - pos\_x;  
double y = maus\_y - pos\_y;*

Now we use Pythagoras (c = Math.sqrt a² + b²) to get the length of this vector, which is the distance between the mouse click and the position of the ball:

*// Calculate distance  
double distance = Math.sqrt ((x\*x) + (y\*y));*

In the last step of the method we are testing if the distance we calculated in the step before is smaller than a certain value, (for example the radius of the ball). I have choosen 15 as a good value although our ball has only the radius 10. But that has no special reason, it just seemed to fit best!

*// If distance is smaller than 15, player has hit the ball  
if (distance < 15)  
{*

*player.addScore (10\* Math.abs(x\_speed) + 10);  
return true;*

*}  
else return false;*

Now we are able to hit the ball by clicking on it with the mouse pointer.

### **The player- object: Count score and lose lives**

To count the score and lose lives, we added the methods looseLife() and addScore (int plus) to our player class. Everytime a ball leaves the applet, isOut() calles player.looseLife(), that way the player loses one life. If the player hits a ball, the method userHit() calls player.addScore (10\* Math.abs(x\_speed) + 10) and adds the score to the score in the player object (the faster the ball has been, the more points the player gets). The methods addScore and looseLife are really simple.

But there is one important point: The player object is initialized in the Main class! This means that the reference to the player object has to be initialized in the ball object too, to make it possible that the ball object calls methods of the player object!!   
This is a very important technique because it happens quite often that more than one class needs a reference to a special object. But it means one has to initialize the object in one class, (I'm using the Main class to manage all game objects), and has to give the reference to this object, which is needed in more classes, to the other classes!

### **Change the mouse pointer to a crosshair cursor**

In this game it would look much better to have a crosshair mouse pointer than a normal mouse pointer. To get such a cursor, we have to add three lines of code to the Main class of our applet:

First of all a instance variable of the class cursor:

*// Crosshair cursor  
Cursor c;*

Now we add the following lines to the init() method:

*// generate a Crosshair cursor  
c = new Cursor (Cursor.CROSSHAIR\_CURSOR);  
// set this cursor as the standard cursor of the applet   
this.setCursor (c);*

You can find other mouse pointers in the Java API

### **Start the game with a double click on it and finish it if the player loses all his lives:**

Now we have almost done it, there is just one thing left to do: We want to start the game when the player clicks two times on the applet (double click) and not earlier. The advantage of this is not only that the player can decide when to start the game but in games where we need the keyboard, we can get the keyboard focus for this applet very easily this way. And of course the game shall be finished if the player loses all of his lives!

As a first step we add a boolean instance variable called isStoped to our Main class. If the value of isStoped is true, our game is not running, if the value is false, the game is running! Now we add a test to the run - method of the main class, which tests if the game is running (isStoped = false) and if the player has more than 0 lives. If these two things are alright, the two balls are moved.

*// run - method  
while (true)  
{*

*if (player.getLifes() >= 0 && !isStoped)  
{*

*redball.move();  
blueball.move();*

*}  
  
...*

*}*

In the next step we'll add a second test to the paint() - method of the applet. As long as the player has lives left, this method paints the two balls, the score and the lives of the player. If the game is stoped it paints also the information on the screen that the game can be started with a double click. If the player loses all of his lives, the paint method writes out the final score and the information that the game can be restarted by double clicking on the applet. The paint method looks like this:

*// paint() - method  
public void paint (Graphics g)  
{*

*// player has lives left  
if (player.getLifes() >= 0)  
{*

*// Paint the two balls, score ...  
  
...  
  
// If game is stopped  
if (isStoped)  
{*

*// paint information: "Start game with a double click"*

*}*

*}  
// Player has no lives left  
else if (player.getLifes() < 0)  
{*

*// paint final score, set isStopped true...  
// for details see sourcecode!*

*}*

*}*

We have almost made it, but...! At the moment we can't switch between the two game states (game stopped, game running) because we have no way to control the value of isStoped. To get this control, we have to add the following code to the mouseDown - method:

*// listen to mouse clicks  
public void mouseDown (Event e, int x, int y)  
{*

*// handle mouse events when game is running  
if (!isStoped)  
{*

*// Test if red ball has been hit  
if (redball.userHit (x, y))  
{*

*// play audio file  
hitnoise.play();  
  
// reset ball  
redball.ballWasHit ();*

*}  
// Test if blue ball has been hit  
if (blueball.userHit (x, y))  
{*

*// play audio file  
hitnoise.play();  
  
// reset ball  
blueball.ballWasHit ();*

*}  
else  
{*

*// play normal shot audio file  
shotnoise.play();*

*}*

*}  
// handle mouse events if game is stopped  
else if (isStoped && e.clickCount == 2)  
{*

*// reset all important values!  
isStoped = false;  
init ();*

*}  
  
return true;*

*}*

### **That's it!!**

Well, you've made it. I hope you understood everything and also these parts, I didn't explain in detail.

The next chapters will tell you something about very special solutions to problems I once had.   
Now feel free to program any game you want to program, use the power of Java, which makes almost everything possible, good luck in developing your own games and I hope, that I could help you a little bit with this tutorial! If you have problems, tutorials, games ... please send me a mail!

Now you can take a look at the game, download the sourcecode and have fun!