

JAVA

Klaus Knopper

(C) 2005 knopper@bw.fh-kl.de

Zusammenfassung

Java ist eine objektorientierte Programmiersprache, die komplexe Zusammenhänge und Strukturen anhand von Objekten und Klassen modelliert, und dem Programmierer damit die Abstraktion einer Aufgabenstellung erleichtert und die Übersichtlichkeit auch großer Programme fördert. Durch Kapselung von Funktionen und Vererbung wird die Entwicklung von Programmierschnittstellen besonders gut unterstützt, was für die Implementation von „großen“ Anwendungen (Middleware, GUIs auf dem Desktop) sehr hilfreich sein kann.

Grundsätzliches zu Java

- Der Java-Compiler (**javac**) erzeugt üblicherweise keinen selbstständig lauffähigen Maschinencode, sondern einen sogenannten *Bytecode*, zu dessen Ausführung die virtuelle Java-Maschine (JVM bzw JRE) benötigt wird.
- Java-Bytecode gibt es in einer „abgespeckten“ Form, die in einem Browser (mit Java-Support) lauffähig ist („Applet“-Klasse), sowie in der klassischen Form als Objektdaten zur Ausführung mit der JVM.
- Mit Hilfe des GNU Java Compilers **gcj** lassen sich bestimmte Java-Programme auch mit Laufzeitbibliotheken binden, so dass dann doch ein architekturabhängiges Binary entsteht, aber dies wird (noch) selten verwendet.

Folie 2

Vorteile von Java-Programmen

- Architektur (Rechner-) unabhängig, zur Ausführung wird lediglich eine installierte Java Virtuelle Maschine benötigt.
- Große Menge an vordefinierten Klassen und Methoden.
- „Baukasten“-Prinzip.
- Leistungsfähige Entwicklungsumgebungen (z.B. **eclipse**) verfügbar.

Folie 3

Nachteile von Java

- Langsam, da nur interpretiert und nicht direkt als Maschinencode ausgeführt.
- Es ist eine Virtuelle Maschine notwendig, die zur Programmversion „passen“ muss.
- Inkompatibilitäten durch mangelhafte Versionierung von Klassenbibliotheken und ggf. veraltete virtuelle Maschinen.
- Wird schnell unübersichtlich ohne Entwicklungsumgebung.
- Zwar sind viele Klassenmethoden „selbsterklärend“ benannt, jedoch ist der Quelltext gegenüber beispielsweise C oft unverhältnismäßig umfangreich bei gleicher Funktionalität.

Folie 4

Beispiel: „Hello, World!“

```
public class hello
{
    public static void main(String[] args)
    {
        System.out.println("Hello, World!");
    }
}
```

Achtung: Die JVM führt standardmäßig die Klasse aus, die so heißt wie die übersetzte Datei. Der Quelltext muss hier also **hello.java** heißen, und der Bytecode wird nach der Übersetzung (**javac hello.java** erzeugt **hello.class**) entsprechend mit **java hello** ausgeführt!

Folie 5

Ein Applet

„Java ist einfach.“ (?)

```
import java.applet.*;
import java.awt.*;
public class Text extends Applet {
    String hello = "Hello World";
    public void paint(Graphics g) {
        g.drawString(hello, 5, 25);
    }
}
```

Dieses Applet kann, in HTML-Seiten eingebettet per **SCRIPT**-Tag, vom Browser ausgeführt ein Fenster mit dem Textinhalt „Hello, World!“ öffnen, eine funktionierende JVM als Browser-Plugin vorausgesetzt.

Folie 6

Java-Schlüsselwörter

Die folgenden Schlüsselwörter sind Bestandteil der Sprache Java, und dürfen nicht als Bezeichner für Variablen verwendet werden:

abstract	boolean	break	byte
case	char	class	const
continue	default	do	double
else	extends	finally	float
for	goto	if	implements
import	instanceof	int	long
native	new	package	private
protected	return	short	super
switch	synchronized	this	throws
transient	try	void	while

Folie 7

Variablendeklarationen in Java

Im Gegensatz zu C können in Java Variablen auch zwischen Anweisungen bzw. Methodenaufrufen deklariert werden, z.B. erst unmittelbar vor ihrer ersten Benutzung.

```
public class VarDecl {  
    public static void main(String[] args) {  
        int a;  
        a = 1;  
        char b = 'x';  
        System.out.println(a);  
        double c = 3.1415;  
        System.out.println(b); System.out.println(c);  
        boolean d = false; System.out.println(d);  
        for(int i = 0; i < 10; i++)  
            System.out.println(i);  
    }  
}
```

Folie 8

Klassen vs. Dateien

Es ist zwar möglich, Klassen innerhalb anderer Klassen zu definieren, üblicherweise werden Klassen aber in einzelnen Dateien gespeichert, wobei der Dateiname dem Klassennamen (plus Endung **.java**) entspricht, unter Beachtung von Groß- und Kleinschreibung.

Wird nun in einer Klasse auf Methoden oder Variablen einer anderen Klasse zugegriffen, so versucht die Java VM die jeweilige Klasse aus einer Datei **KlassenName.class** nachzuladen, d.h. jede in einem Java-Programm verwendete Klasse darf (und wird üblicherweise auch) in einer separaten Datei gespeichert und übersetzt werden.

Der „Zusammenbau“ des Programms und die Evaluation der Klassenmethoden und Variablen erfolgt zur Laufzeit, sofern nicht schon der Compiler die Informationen über eine Klasse benötigt, von der die korrekte Übersetzung des anderen Klasse abhängt. In diesem Fall muss der Quelltext für die jeweiligen Klassen in der richtigen Reihenfolge übersetzt werden.

Folie 9

Klassenmethoden (1)

In jeder Klasse lassen sich neben Variablen auch *Methoden* (vergl. C: globale Funktionen) definieren, die nur im Zusammenhang mit einem Objekt dieser Klasse, oder einem expliziten Methodenaufruf aus der Klasse (solange dort nicht auf Objektelemente zugegriffen wird), aufrufbar sind.

```
public class Auto {
    public String name;
    public int    erstzulassung;
    public int    leistung;
    public int alter() { return erstzulassung - 2000; }
}
...
Auto a = new Auto();
a.erstzulassung = 2004;
System.out.println( a.alter() );
```

Folie 10

Klassenmethoden (2)

Als **static** deklarierte Methoden dürfen auch direkt innerhalb einer Klasse aufgerufen werden, ohne dass erst ein Objekt dieser Klasse erzeugt werden muss.

```
public class Klasse {
    public static void hallo() {
        System.out.println("Ich bin Klasse!");
    }
}
...
Klasse.hallo();
```

Folie 11

Sichtbarkeit von Objektattributen in Java

Mit dem Schlüsselwort **private** vor einer Variablen innerhalb einer Klasse wird dafür gesorgt, dass diese Variable nur für die Klasse selbst und deren Klassenmethoden sichtbar ist. **public** sorgt hingegen dafür, dass auch Objekte von anderen Klassen Zugriff auf die Variablen bekommen.

Es gilt als guter Stil, klasseninterne Variablen stets als **private** zu deklarieren (Voreinstellung), und **public**-Methoden zu definieren, die einen kontrollierten Zugriff auf die privaten Variablen von außen erlauben.

```
public class Pizza {  
    private int scharf;  
    public boolean getPfefferMenge() { return scharf; }  
    public void setPfefferMenge(int wieviel) {  
        scharf = wieviel;  
    }  
}
```

Folie 12

static-Variablen

Als **static** deklarierte Variablen existieren, ähnlich wie bei C, vom ersten Laden der Klasse bis zum Programmende, und können entsprechend von mehreren Instanzen/Objekten einer Klasse gemeinsam benutzt werden, werden also nicht für jede Instanz neu erzeugt. Manche von der Java VM und der Java Klassenbibliothek vorgegebene Variablen oder Methoden müssen als **static** definiert werden, z.B. das schon bekannte `public static void main()` als automatisch aufgerufene Methode.

Folie 13

Objekterzeugung in Java

In Java sind Objektvariablen zunächst *Referenzen* (ähnlich wie Zeiger in C), denen entweder ein bereits existierendes Objekt (bzw. dessen Referenz) zugewiesen werden kann, oder ein *neues* Objekt, was mit dem Java-Schlüsselwort **new** und dem Aufruf des *Konstruktors* erzeugt wird.

```
public class Test {  
  
    public class NeuesObjekt {  
        public int zahl;  
    }  
  
    public static void main(String[] args) {  
        NeuesObjekt n = new NeuesObjekt();  
        n.zahl = 1;  
    }  
}
```

Folie 14

Konstruktor

Jede Klasse enthält, unsichtbar, (mindestens) eine Methode, die so heißt wie die Klasse selbst, und die beim Erzeugen eines Objektes dieser Klasse automatisch aufgerufen wird. Man kann diese Methode selbst (über-)schreiben, und mit Parametern versehen:

```
public class Auto  
{  
    public String name;  
    public int    erstzulassung;  
    public int    leistung;  
    public Auto(String name) { this.name = name; }  
}
```

Später kann dann ein Objekt der Klasse **Auto** mit `Auto a = new Auto("Flitzer");` erzeugt werden, wodurch die Variable **name** mit dem String "Flitzer" vorbelegt wird. **this** ist stets eine Referenz auf das aktuelle Objekt, und kann normalerweise entfallen (allerdings nicht in diesem Beispiel, warum?).

Folie 15

Destruktor

Jede Klasse enthält, unsichtbar, eine Methode, die nach Beendigung der Lebenszeit eines Objektes dieser Klasse automatisch von der sog. „Garbage Collection“ der Java VM aufgerufen wird. Es kann allerdings passieren, dass dies erst zum Ende des gesamten Programms passiert, und daher die Anweisungen im Destruktor gar nicht zur Ausführung gelangen.

```
public class Test {
    public class KonstruktorTest {
        KonstruktorTest() {
            System.out.println("Konstruktor wurde aufgerufen!");
        }
        protected void finalize() {
            System.out.println("Destruktor wurde aufgerufen!");
        }
    }
    public static void main(String[] args) {
        KonstruktorTest t1 = new KonstruktorTest();
    }
}
```

Folie 16

Assoziationen zwischen Klassen

Ein Objekt **k1** der Klasse **Kunde** greift auf die Methode **abbuchen()** der Klasse **Konto** zu.

```
class Kunde
{
    Konto k1;
    void geld_abheben() { k1.abbuchen(); }
}

class Konto
{
    public void abbuchen() { ... };
}
```

Folie 17

Aggregation

Ein Auto besteht aus 4 Rädern und einem Motor.

```
class Auto
{
    Rad vr, vl, hr, hl;
    Motor m;

    void starte()
    {
        m.ein();
    }
}
```

Die Klassen **Rad** und **Motor** müssen natürlich noch entsprechend definiert werden.

Folie 18

Vererbung von Klasseneigenschaften

Ein Kaffeemaschine hat eine Funktion zur Befüllung von Tassen. Eine spezielle Kaffeemaschine Cafe2000 hat zusätzlich einen Timer.

```
class Kaffeemaschine
{
    int tasse;
    public void befüllung(int wieviel) {
        tasse = wieviel;
    }
}
class Cafe2000 extends Kaffeemaschine
{
    Time t;
    public void timer(Time time) {
        t = time;
    }
}
```

Folie 19

Polymorphie

```
class PTier {
    public String farbe;
    String kennung() {
        return "Keine Ahnung was ich mal werde.";
    }
}
class PPinguin extends PTier {
    String kennung() { return "Ich bin ein Pinguin."; }
}
...
PPinguin pingu = new PPinguin();
System.out.println( pingu.kennung() );
```

Frage: Was würde ausgegeben werden für die Fälle
PTier pingu = new PTier(); oder
PTier pingu = new PPinguin();?

Folie 20

Deklaration von Arrays

Arrays in Java sind Objekte, das bedeutet auch, dass Arrays Methoden und Instanz-Variablen besitzen können. Für die Deklaration muss zuerst eine Array-Variable deklariert werden und anschließend das erzeugte Array der Variablen zugewiesen werden.

```
int[] a; // Deklaration Array-Variable a
// Zuweisung eines Arrays mit 5 int-Elementen
a = new int[5];
```

Die Deklaration und Initialisierung kann auch in einem Schritt durchgeführt werden.

```
int[] a = new int[5];
int[] a = { 1, 2, 3, 4, 5}; // Initialisierung
```

Folie 21

Zugriff auf Arrays

Der Zugriff auf ein Array-Element erfolgt, wie in C, über seinen Index, beginnend mit 0.

```
public static void main(String[] args)
{
    int[] zahl = new int[2];
    zahl[0] = 2;
    zahl[1] = 3;
    System.out.println("zahl hat " + zahl.length +
                       " Elemente.");
    System.out.println(zahl[0]);
    System.out.println(zahl[1]);
}
```

Folie 22

Die **String**-Klasse (1)

In der Java-Klasse **String** sind im Gegensatz zum aus C bekannten **char *** auch Methoden definiert, die es erlauben, mit aus der Arithmetik bekannten Operatorzeichen Zeichenketten zusammenzuhängen, oder Umwandlungen zwischen Text und Zahlen vorzunehmen.

```
public class StringVerkett
{
    public static void main(String[] args)
    {
        int a = 5;
        double x = 3.14;

        System.out.println("a = " + a);
        System.out.println("x = " + x);
    }
}
```

Folie 23

Die **String**-Klasse (2)

```
public class StringVergleich
{
    public static void main(String[] args)
    {
        String a = new String("hallo");
        String b = new String("hallo");
        System.out.println("a == b liefert " + (a == b));
        System.out.println("a != b liefert " + (a != b));
    }
}
```

Folie 24

Die **String**-Klasse (3)

```
public class StringVergleich2
{
    public static void main(String[] args)
    {
        String a = new String("hallo");
        String b = new String("hallo");
        System.out.println("a.equals(b) liefert " +
                           a.equals(b));
    }
}
```

Folie 25

Integeroperationen - Beispiel

Die Standard-Klasse **Integer** in Java stellt einige praktische Methoden zur Verfügung, die mit dem Basistyp **int** nicht so einfach zu realisieren sind.

```
public class IntOut0 {
    public static void main (String args[]) {
        int k=987654321;
        System.out.println(k + " zur Basis 10 ist " +
                           Integer.toString(k));
        System.out.println(k + " zur Basis  2 ist " +
                           Integer.toBinaryString(k));
        System.out.println(k + " zur Basis  8 ist " +
                           Integer.toOctalString(k));
        System.out.println(k + " zur Basis 16 ist " +
                           Integer.toHexString(k));
        System.out.println(k + " zur Basis  3 ist " +
                           Integer.toString(k,3));
    }
}
```

Folie 26

Fehler und Ausnahmebehandlungen

In Java lassen sich „ungewöhnliche“ Ereignisse wie die vorzeitige Beendigung von Tastatureingaben, Fehler beim Lesen und Schreiben von Dateien etc. mit Hilfe sogenannter „Exceptions“ abbilden, und unter Verwendung von **try ... catch** Fehlerbehandlungsmethoden definieren. Man kann jedoch auch durch gezieltes „Werfen“ von Fehlersignalen die Standardfehlerbehandlung der Java VM auslösen, ähnlich wie **exit(Fehlercode)**; bei C dem aufrufenden Programm einen Fehler nebst Programmabbruch mitteilt.

```
public class Ausnahmen
{
    public static void main(String[] args)
    {
        if (args.length < 2) {
            throw new IllegalArgumentException();
        }
        ...
    }
}
```

Folie 27

Texteingaben mit Java

Das Einlesen von Variablen ist in Java, verglichen mit C, verhältnismäßig kompliziert, da für die verschiedenen Eingabemethoden zunächst Eingabeobjekte erzeugt werden müssen, und Fehler bei der Eingabe mit **try...catch**-Konstruktionen (sog. Ausnahmebehandlungen) abgefangen werden müssen.

Es bietet sich an, für häufig verwendete Einleseoperationen eigene Klassen und entsprechende Klassenmethoden anzulegen, die später von anderen Klassen einfach aufgerufen werden können.

Folie 28

Textein- und ausgabe - Beispiel #1

Textein- und ausgabe über die Kommandozeile:

```
import java.io.*; // io-Klassen laden
public class TextEinAusgabe {
    public static void main(String[] args) {
        System.out.println("Text eingeben:");
        try {
            BufferedReader in = new BufferedReader(
                new InputStreamReader(System.in) );
            String s = in.readLine();
            System.out.println("Der Text lautet: " + s);
        } catch( IOException ex ) {
            System.out.println( ex.getMessage() );
        }
    }
}
```

Folie 29

Textein- und ausgabe - Beispiel #2a

Textein- und ausgabe als grafische Applikation:

```
import java.awt.*;
import java.awt.event.*;
public class TextEinAusgabe extends Frame
{
    TextField eingabe;
    Label      ausgabe;
    public static void main( String[] args ) {
        TextEinAusgabe meinFenster = new TextEinAusgabe("Eingabe");
        meinFenster.setSize(400, 200);
        meinFenster.show();
    }
    public TextEinAusgabe( String fensterTitel ) {
        super(fensterTitel);
        Label hinweis = new Label( "Text eingeben");
        eingabe = new TextField();
        ausgabe = new Label();
```

Folie 30

Textein- und ausgabe - Beispiel #2b

```
add( BorderLayout.NORTH,  eingabe );
add( BorderLayout.CENTER, hinweis );
add( BorderLayout.SOUTH,  ausgabe );
eingabe.addActionListener(
    new ActionListener() {
        public void actionPerformed((ActionEvent ev) {
            meineMethode(); } } );
addWindowListener(
    new WindowAdapter() {
        public void windowClosing( WindowEvent ev ) {
            dispose();
            System.exit( 0 ); } } );
}
void meineMethode() {
    ausgabe.setText( "Der eingelesene Text lautet: " +
                    eingabe.getText() );
}
}
```

Folie 31

Textein- und ausgabe - Beispiel #3a

Text- und Ausgabe als Applet, eingebettet in eine HTML-Seite:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class TextEinAusgabe extends Applet
{
    TextField eingabe;
    Label      ausgabe;
    public void init() {
        Label hinweis = new Label( "Oben Text eingeben" );
        eingabe = new TextField( "      " );
        ausgabe = new Label();
        setLayout( new BorderLayout() );
        add( BorderLayout.NORTH,  eingabe );
        add( BorderLayout.CENTER, hinweis );
        add( BorderLayout.SOUTH,  ausgabe );
    }
}
```

Folie 32

Textein- und ausgabe - Beispiel #3b

```
    eingabe.addActionListener(
        new ActionListener() {
            public void actionPerformed((ActionEvent ev) {
                meineMethode(); } } );
    }
    void meineMethode() {
        ausgabe.setText( "Der Text lautet: " +
                        eingabe.getText() );
    }
}
```

Folie 33

Beispiel: Systemeigenschaften anzeigen

```
public class SystemProperties
{
    public static void main( String[] args )
    {
        System.out.println(
            System.getProperties().toString()
                .replace( ',', '\n' ).replace( '{', ' ' )
                .replace( '}', ' ' ) );
    }
}
```

NB: Jedes **replace()** liefert hier einen **String** zurück, in dem wieder die Methode **replace()** aufgerufen werden kann.

Folie 34

Grafik unter Java - die **Graphics()**-Klasse

Viele grundlegende Grafikfunktionen von Java sind in der **Graphics()**-Klasse untergebracht, die zum Paket **java.awt** gehört.

Die **Graphics()**-Klasse umfasst u.a. Methoden zum Zeichnen von Linien, Kreisen, Ellipsen, Rechtecken und Texten in verschiedenen Farben und Schriften.

Folie 35

Nachtrag: Grundgerüst eines Applets

```
// Importieren der notwendigen awt-Klassen
public [Klassenname] extends java.applet.Applet {
// Variablen-Deklaration und Definition von Methoden
    public void init() { ... } // beim Laden des Applets
    public void start() { ... } // beim Start des Applets
    public void stop() { ... } // wenn HTML-Seite mit Applet
                                // verlassen wird
    public void destroy() { ... } // Löschen des Applets und
                                // Freigeben der Ressourcen
    public void paint(Graphics g) { ... } // wird aufgerufen,
                                // wenn Applet gezeichnet wird
    public void run() { ... } // bei Multithreading anstelle
                                // von paint()
    ...
}
```

Folie 36

Die **drawLine()**-Methode

Mit dieser Methode kann man eine einfache Linien zwischen zwei vorgegebenen Koordinatenpunkten zeichnen.

```
import java.awt.Graphics;
public class ZieheLinie extends java.applet.Applet {
    public void paint(Graphics g) {
        // Zeichnen einer Linie zwischen Anfangspunkt
        // (42,42) und Endpunkt (100,100)
        g.drawLine(42,42,100,100);
    }
}
```

Folie 37

Die **drawRect ()**-Methode

Für das Zeichnen von Rechtecken kann die **drawRect ()**-Methode genutzt werden. Als Parameter werden die Koordinaten des oberen linken Punktes, die Breite und die Höhe übergeben.

```
import java.awt.Graphics;
public class MaleRec extends java.applet.Applet {
    public void paint(Graphics g) {
        // Rechteck mit links oberer Koordinate (150,100),
        // einer Breite von 200 und Höhe von 120 Pixeln
        g.drawRect(150, 100, 200, 120);
    }
}
```

Folie 38

Beispiele zur **Graphics ()**-Klasse #1

gefülltes Rechteck mit links oberer Koordinate (10,100), einer Breite von 200 Pixeln und einer Höhe von 42 Pixeln.

```
import java.awt.Graphics;
public class MaleRecFil extends java.applet.Applet {
    public void paint(Graphics g) {
        g.fillRect(10, 100, 200, 42);
    }
}
```

Folie 39

Beispiele zur **Graphics()**-Klasse #2

Zeichnet ein dreidimensionales Rechteck. Der 5. Parameter ist ein boolescher Wert, der angibt, ob das Rechteck als erhöht dargestellt werden soll.

```
import java.awt.Graphics;
public class MaleRec3D extends java.applet.Applet {
    public void paint(Graphics g) {
        g.draw3DRect(10, 100, 200, 42, true);
    }
}
```

Folie 40

Beispiele zur **Graphics()**-Klasse #3

Zeichnet ähnlich wie **drawRect()** ein Rechteck, aber mit abgerundeten Ecken. Die beiden ersten Parameter geben den Winkel der Abrundung in der horizontalen und der vertikalen Ebene an.

```
import java.awt.Graphics;
public class MaleRecRound extends java.applet.Applet {
    public void paint(Graphics g) {
        g.drawRoundRect(10, 100, 50, 42, 5, 5);
        g.drawRoundRect(110, 200, 100, 42, 5, 50);
        g.drawRoundRect(150, 250, 25, 42, 50, 25);
    }
}
```

Folie 41

Beispiel: Polygonzug

```
import java.awt.Graphics;
import java.awt.Polygon;
public class ZeichnePolygon2 extends java.applet.Applet {
    // Definiere ein Array mit X-Koordinaten
    int[] xCoords = { 10, 40, 60, 300, 10, 30, 88 };
    // Definiere ein Array mit Y-Koordinaten
    int[] yCoords = { 20, 0, 10, 60, 40, 121, 42 };
    // Bestimme Anzahl Ecken über Methode length
    // des Array-Objekts mit x-Koordinaten
    int anzahlEcken = xCoords.length;
    public void paint(Graphics g) {
        Polygon poly = new Polygon(xCoords, yCoords, anzahlEcken);
        // Zeichne ein 7-Eck
        g.drawPolygon(poly);
    }
}
```

Folie 42

Beispiel: Zeichnen von Kreisen und Ellipsen

```
import java.awt.Graphics;
public class ZeichneOval extends java.applet.Applet {
    public void paint(Graphics g) {
        // Zeichne eine Ellipse
        g.drawOval(50, 100, 200, 120);
        // Zeichne einen Kreis
        g.drawOval(175, 175, 200, 200);
    }
}
```

Folie 43

Farben setzen mit **setColor()**

```
import java.awt.*;
public class Farbe extends java.applet.Applet {
    public void paint(Graphics g) {
        // Erzeugen von Farbobjekten
        Color pinkColor = new Color(255, 192, 192);
        Color irgendeineFarbe1 = new Color(255, 100, 92);
        Color irgendeineFarbe2 = new Color(5, 12, 120);
        // Zeichnen von Ellipsen
        g.setColor(irgendeineFarbe2);
        g.drawOval(5, 5, 150, 250);
        // Zeichne einen Kreis
        g.setColor(pinkColor);
        g.fillOval(250, 150, 150, 150);
        // direkte Verwendung einer Farbkonstanten
        g.setColor(Color.green);
        g.drawRect(40, 40, 100, 200);
    }
}
```

Folie 44

Beispiel **setBackground()**

Mit **setBackground(Color c)** und **setForeground(Color c)** kann man die Hintergrund- und Vordergrundfarbe setzen.

```
public void paint(Graphics g) {
    /* Hier wird auf einem rosa Hintergrund */
    /* ein grünes Rechteck gezeichnet */
    Color pinkColor = new Color(255, 192, 192);
    setBackground(pinkColor);
    g.setColor(Color.green);
    g.drawRect(40, 40, 100, 20);
}
```

Folie 45

Erstellen von Fontobjekten

```
import java.awt.Font;
import java.awt.Graphics;
public class FontTest extends java.applet.Applet {
    public void paint(Graphics g) {
        Font f = new Font("TimesRoman", Font.PLAIN, 18);
        Font fb = new Font("TimesRoman", Font.BOLD, 18);
        Font f2i = new Font("Arial", Font.ITALIC, 34);
        g.setFont(f);
        g.drawString("Normaler (plain) Font - TimesRoman", 10, 25);
        g.setFont(fb);
        g.drawString("Fetter (bold) Font - TimesRoman", 10, 50);
    }
}
```

Folie 46

Bilder laden und anzeigen

```
import java.awt.Image;
import java.awt.Graphics;
public class DrawImage2 extends java.applet.Applet {
    Image samImage;
    Image bild;
    public void init() {
        // Bild laden - ein jpg-File
        samImage = getImage(getDocumentBase(), "bild.jpg");
    }
    public void paint(Graphics g) {
        g.drawImage(samImage, 0, 0, this);
    }
}
```

Folie 47

Animationen unter Java

```
import java.awt.Image;
import java.awt.Graphics;
public class Animation1 extends java.applet.Applet {
    Image bild;
    public void init() {
        bild = getImage(getCodeBase(), "bild.gif");
        resize(600, 200); }
    public void paint(Graphics g) {
        for (int i=0; i < 1000; i++) {
            int bildbreite = bild.getWidth(this);
            int bildhoehe = bild.getHeight(this);
            int xpos = 10; // Startposition X
            int ypos = 10; // Startposition Y
            g.drawImage(bild, (int)(xpos + (i/2)),
                (int)(ypos + (i/10)), (int)(bildbreite * (1 +
                    (i/1000))), (int) (bildhoehe * (1 + (i/1000))),
                    this); } } }
```

Folie 48

Beispielapplikation: Rotes Rechteck #1

```
import java.awt.*;
import java.awt.event.*;
class MaleRechtFill extends Frame {
    public MaleRechtFill() {
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e){
                dispose();
                System.exit(0);
            }
        });
    }
}
```

Folie 49

Beispielapplikation: Rotes Rechteck #2

```
public static void main(String args[]) {  
    MaleRechtFill mainFrame = new MaleRechtFill();  
    mainFrame.setSize(400, 400);  
    mainFrame.setTitle("Test");  
    mainFrame.setVisible(true);  
}  
public void paint(Graphics g) {  
    g.setColor(Color.red);  
    g.fillRect(50, 70, 200, 100);  
}  
}
```