Objektorientierte Rahmenwerke

Am Beispiel des Java „Abstract Window Toolkit“ (AWT)

universitätbonn

Bibliotheken vs. Rahmenwerke

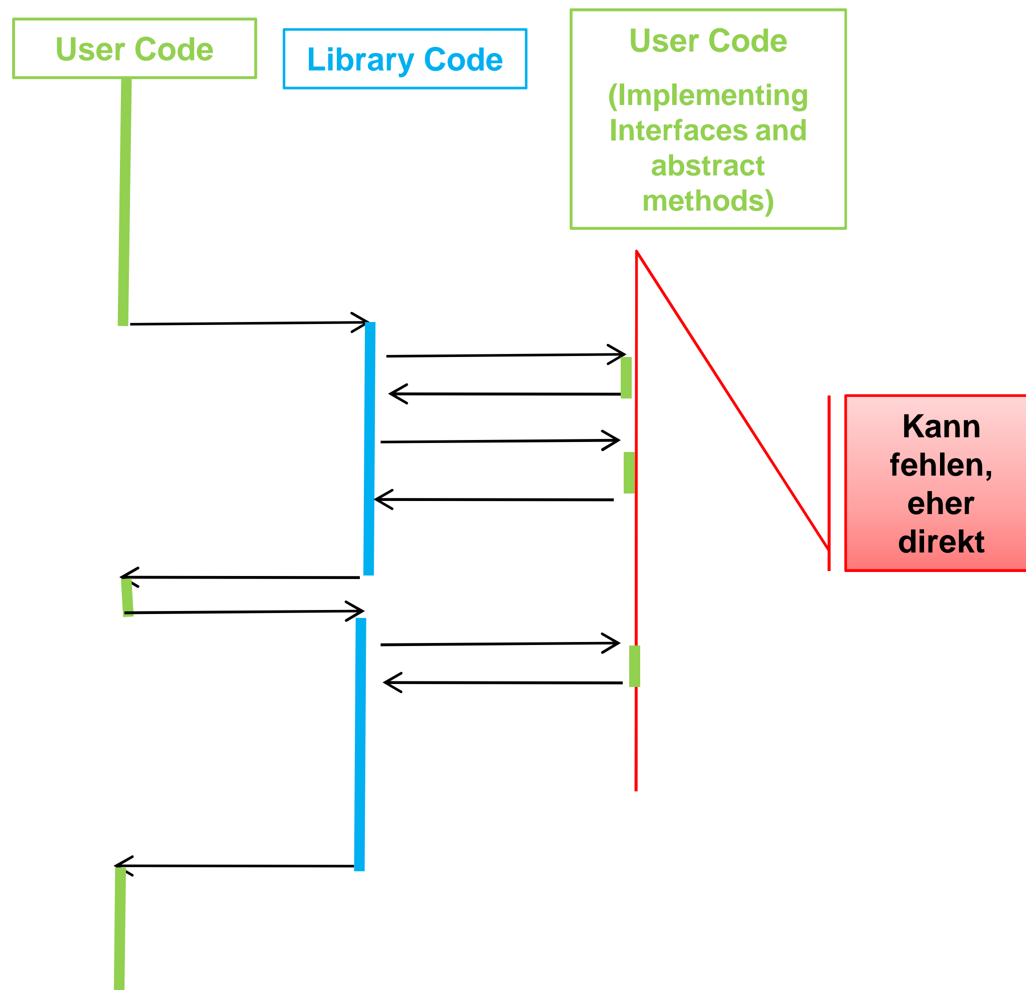
- Erstellung wiederbenutzbarer Softwarebibliotheken eine der Attraktionen OOP
  - Etwa für komplexe Datenstrukturen
    - Vgl. letztes Kapitel
    - Die häufig auch „generisch" für verschiedene Typen einsetzbar sind
  - Ablaufkontrolle liegt bei Benutzerprogramm
    - Aus dem Methoden der Bibliothek explizit aufgerufen werden
- Paradigmen des OOP liefern aber auch Grundlage für „Rahmenwerke" (frameworks)
  - Bibliotheken, bei denen die Ablaufkontrolle (im Wesentlichen) bei der Bibliothek liegt
    - ***„Inversion of control"***
  - Code des Benutzers wird durch „Haken" (hooks) eingehängt
    - Beim OOP durch Überschreiben von Methoden, die in Basisklassen und Interfaces des Frameworks definiert sind

Bibliotheken vs. Rahmenwerke

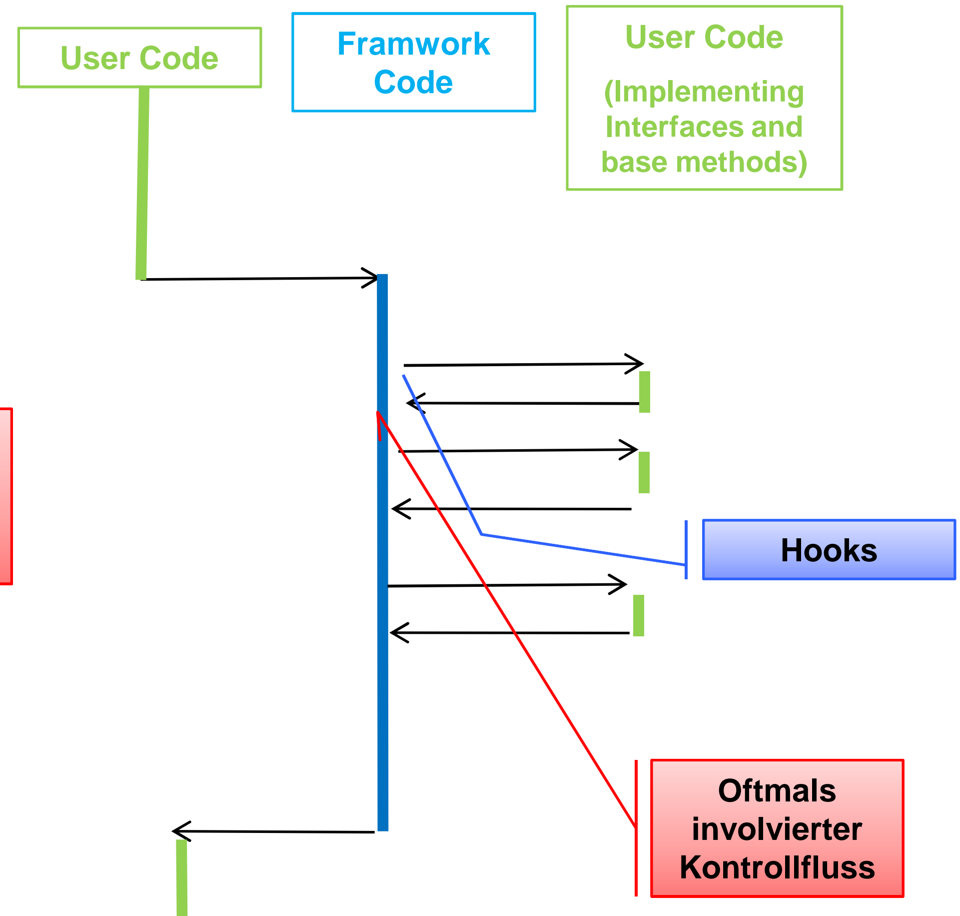
- Übergang zwischen Bibliotheken und Rahmenwerken fließend
  - Auch in Bibliotheken wird häufig Benutzercode aufgerufen, der abstrakte Methoden von Interfaces realisiert
    - ⇨ Vgl. etwa die Methode compareTo, die in Bibliotheksfunktionen von Listen aufgerufen wird
      - – Und Code des Benutzers ausführt
  - Bei Bibliotheken aber im Allgemeinen häufiger Aufruf von Bibliotheksfunktionen aus Benutzerprogramm
  - Bei Rahmenwerken oftmals nur wenige (oder ein) Aufruf des Rahmenwerks aus Benutzercode
    - ⇨ Und viele Aufrufe der Benutzermethoden über die Hooks des Rahmenwerks
    - ⇨ Die oftmals erst nach einem sehr komplexen internen Kontrollfluss innerhalb des Frameworks aufgerufen werden

Bibliotheken vs. Rahmenwerke

- Typischer Kontrollfluss bei einer „klassischen“ Bibliothek



- Typischer Kontrollfluss bei einem Objekt-orientierten Rahmenwerk



Bibliotheken vs. Rahmenwerke

- Wollen diese zunächst etwas abstrakt erscheinende Konzepte an einem Beispiel näher erläutern
  - Dem Java „Abstract Window Toolkit“ (AWT)
- Werden Funktionalität des AWT auch in einem finalen größeren Beispiel benötigen
  - Siehe nächstes Kapitel
    - ⇨ In dem wir von der Objekt-orientierten Analyse über die Verfeinerungs- und Revisionsstufen des Objekt-orientierten Designs zu einer Implementierung in Java gelangen

universitätbonn

Das „Abstract Window Toolkit“ (AWT)

AWT als Beispiel eines Frameworks

Graphische Komponenten

Frames und Applets

Ereignisse

Größeres Beispiel: Ein FunctionPlotter

Java AWT

- Das Java **AWT** (abstract window toolkit) ist eine Klassenbibliothek, mit deren Hilfe graphische Benutzeroberflächen (graphical user interfaces -- **GUIs**) programmiert werden können
  - Wichtigste Bestandteile des AWT sind
    - ⇨ Klassen zur Darstellung graphischer Komponenten in einem Fenstersystem und
    - ⇨ Mechanismen, die eine Interaktion mit dem Benutzer ermöglichen, indem sie es erlauben, auf **Ereignisse** (events) wie z. B. Mausklicks zu reagieren

Java AWT

- Mit der Klassenbibliothek des AWT ist ein **objektorientiertes Rahmenwerk** (object-oriented framework) realisiert
- Wir können relativ leicht eine graphische Benutzeroberfläche programmieren,
  - indem wir unsere Klassen von speziellen AWT-Klassen ableiten
  - und nur relativ wenige Methoden, die in den Basisklassen des AWT definiert sind, durch eigenen Code überschreiben
    - ⇨ Solche Methoden sind **Haken** (hooks) im Rahmenwerk, an denen benutzerspezifischer Code „eingehängt“ wird
- Über höhere objektorientierte Mechanismen wie Vererbung und dynamisches Binden wird der spezielle Code nahtlos in den durch das AWT gegebenen Rahmen integriert
  - und man kann den damit den Code wieder verwenden, den das Rahmenwerk des AWT liefert
  - Im Gegensatz zu einer herkömmlichen Klassenbibliothek liegt die Kontrolle hier beim Framework, das den speziellen Darstellungscode des Benutzers aufruft

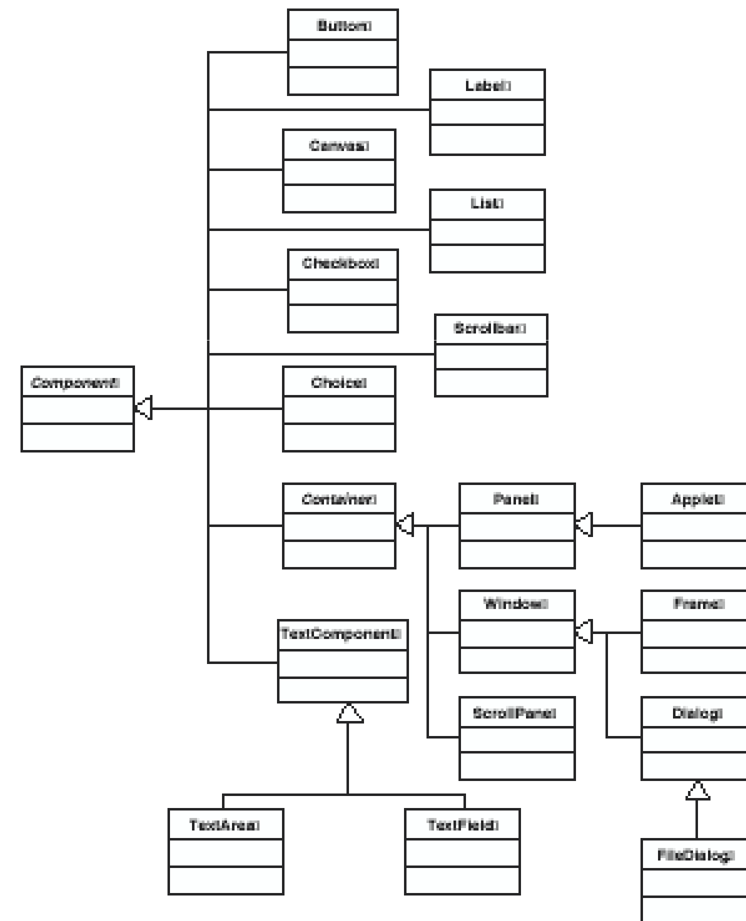
Java AWT

- **Bemerkung:**
  - Wir verwenden für die größeren Beispiele in diesem Kapitel weiterhin das AWT, obwohl es inzwischen neuere Klassenbibliotheken für die GUI-Programmierung in Java gibt
    - ⇨ Insbesondere die mit **Swing** bezeichnete Bibliothek
    - ⇨ Diese ist eine Erweiterung des AWT und bietet gewisse technische Vorteile, wie etwa ein systemunabhängiges aber benutzerdefinierbares „look and feel“ von graphischen Elementen, auf die an dieser Stelle nicht näher eingegangen werden soll
    - ⇨ Aufgrund seiner etwas einfacheren Struktur ist das AWT aber für unsere Zwecke geeigneter als Swing
  - Die Prinzipien des Entwurfs graphischer Komponenten und der Ereignisbehandlung sind in beiden Bibliotheken die gleichen
    - ⇨ Und uns kommt es im wesentlichen auf die Grundstrukturen des Programmierens an

Graphische Komponenten

Vererbungshierarchie für `Component` in `java.awt`

Mit Ausnahme von **`Applet`** gehören alle Klassen zum Paket **`java.awt`**. Die Klasse **`Applet`** ist im Paket **`java.applet`** enthalten.
Alle Klassen erben von der abstrakten Basisklasse ***`Component`***.



Graphische Komponenten: Funktionalität von Component

- Die unten angegebene Funktionalität wird von der Klasse `Component` zur Verfügung gestellt, d. h. sie besitzt folgende (virtuellen) Methoden; in einer vom Benutzer definierten Ableitung können diese entsprechend überschrieben werden
- Grundlegende Zeichenfunktionen:
  - `void paint(Graphics g)` ist die Hauptschnittstelle („Haken“) zum Anzeigen eines Objekts
    - ⇨ Diese Methode wird in einer abgeleiteten Klasse überschrieben, wobei die Methoden, die das Graphics-Objekt g zur Verfügung stellt, benutzt werden
  - `void update()`
  - `void repaint()`
- Funktionen zur Darstellung:
  - `void setFont(Font f)` setzt die **Schriftart** (font), in der Textstücke innerhalb der Graphik geschrieben werden.
  - `void setForeground(Color c)`
  - `void setBackground(Color c)`

Graphische Komponenten: Funktionalität von Component

- Funktionen zur Größen- und Positionskontrolle
  - `Dimension getMinimumSize()`
  - `Dimension preferredSize()`
  - `void setSize(int width, int height)` setzt die Größe der Komponente (in Pixeleinheiten)
  - `void setSize(Dimension d)`
  - `Dimension getSize()`

Die Klasse Graphics

- Die `paint`-Methode von AWT-Komponenten besitzt einen Parameter vom Typ `Graphics`
- Die Klasse `Graphics` ist die abstrakte Basisklasse für alle Klassen, die graphische Ausgabeobjekte realisieren
  - wie z. B. Treiberklassen für verschiedene Bildschirme, Drucker, . . .
  - Sie stellt einen sogenannten **Graphik-Kontext** (graphics context) zur Verfügung

Die Klasse Graphics

- Ein Graphik-Kontext wird vom Rahmenwerk des AWT erzeugt, nicht unmittelbar im Anwendungsprogramm
  - Über die „Haken“, die ein Graphics-Objekt als Parameter besitzen, kann im Anwendungsprogramm aber auf den vom AWT erzeugten Graphik-Kontext zugegriffen werden
    - ⇨ Ein wichtiger Haken ist etwa die Methode **`paint(Graphics g)`** der Basisklasse **`Component`**
- Die Klasse **`Graphics`** spezifiziert eine Vielzahl von Methoden, mit denen in dem Graphik-Kontext „gezeichnet“ werden kann
  - Viele der Argumente, die vom Typ **`int`** sind, bezeichnen Koordinaten, die in Einheiten von **Bildpunkten** (picture element, pixel) gegeben sind, an denen ein spezielles Objekt gezeichnet werden soll
    - ⇨ Dabei hat die *linke obere* Ecke die Koordinate (0,0)
    - ⇨ Die x-Koordinate wächst nach *rechts*, die y-Koordinate nach *unten*

Die Klasse Graphics

- Einige der wichtigsten Methoden sind die folgenden
  - `void setColor(Color c)`
    - ⇨ legt die in den folgenden Operationen verwendete **Farbe** fest
  - `void setFont(Font f)`
    - ⇨ legt die in den folgenden Text-Operationen verwendete **Schriftart** (font) fest
  - `void drawString(String str, int x, int y)`
    - ⇨ schreibt den `String str` in dem gerade gültigen Font (und in der gerade gültigen Farbe) an den Punkt (x, y)
  - `void drawLine(int x1, int y1, int x2, int y2)`
    - ⇨ zeichnet eine **Strecke** (in der gerade gültigen Farbe) vom Punkt (x1, y1) zum Punkt (x2, y2) im Koordinatensystem des Graphik-Kontexts

Die Klasse Graphics

- Einige der wichtigsten Methoden (Forts.)
  - `void drawRect(int x, int y, int width, int height)`
    - zeichnet den Umriss eines **Rechtecks** (in der gerade gültigen Farbe), das durch die Parameter spezifiziert ist
    - Der linke und rechte Rand des Rechtecks sind bei x und x+width
    - Der obere und untere Rand sind bei y und y+height
  - `void fillRect(int x, int y, int width, int height)`
    - zeichnet ein **Rechteck**, das mit der gerade gültigen Farbe **gefüllt** ist
    - Der linke und rechte Rand des Rechtecks sind bei x und x+width-1
    - Der obere und untere Rand sind bei y und y+height-1

Die Klasse Graphics

- Einige der wichtigsten Methoden (Forts.)
  - **`void drawOval(int x, int y, int width, int height)`**
    - ⇨ zeichnet den Umriss einer **Ellipse** (in der gerade gültigen Farbe), die in das Rechteck eingepasst ist, das durch die Parameter gegeben ist
  - **`void fillOval(int x, int y, int width, int height)`**
    - ⇨ zeichnet eine **Ellipse**, die mit der gerade gültigen Farbe **ausgefüllt** ist
    - ⇨ Die Ellipse ist in das Rechteck eingepasst, das durch die Parameter gegeben ist.

Frames

- Die von ***`Component`*** abgeleitete Klasse **`Frame`** aus **`java.awt`** dient zum Zeichnen von Fenstern mit Rahmen
  - In einem Frame-Objekt kann ein **Menü-Balken** (menu bar) verankert sein, über den Dialog-Menüs verankert sein können
- AWT-Frames haben folgende wichtige spezifische Funktionalität
  - **`void setTitle(String title)`** schreibt einen Titel in die obere Leiste des Fensters
  - **`void setMenuBar(MenuBar mb)`**
  - **`MenuBar getMenuBar()`**
- Um im Fenster, das durch ein Frame-Objekt erzeugt wird, etwas anzeigen zu können, muss - wie bei allen AWT-Komponenten - die **`paint`**-Methode überschrieben werden

Frames: Beispiel

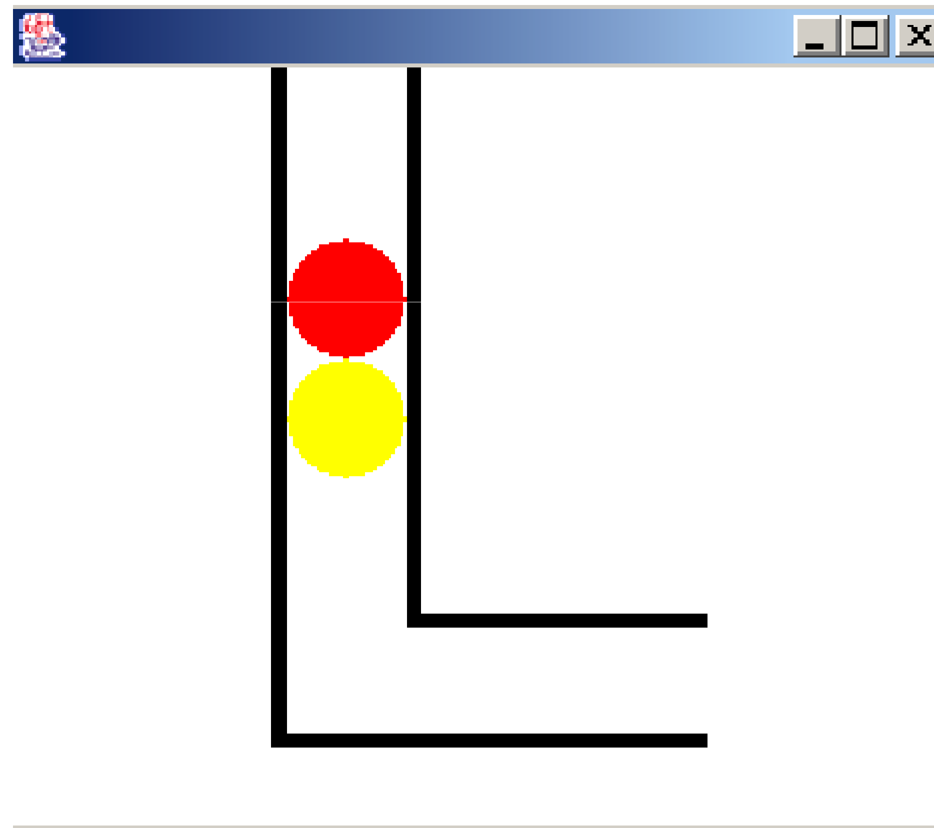
- Das folgende Programm zeichnet in einem Fenster einen Teil der „Vereinzelungseinheit“

```
/**
 * Zeichnet statisch einen Teil der
 * Vereinzelungseinheit in ein Fenster.
 */
import java.awt.*;

public class FrameBeispiel extends Frame {
    /**
     * Erzeugt ein Fenster
     * der Groesse 320 x 280.
     */
    FrameBeispiel() {
        setSize(320,280);
        setVisible(true);
    }

    public static void main(String[] args) {
        new FrameBeispiel();
    }
```

```
    /**
     * Zeichne einige einfache
     * graphische Objekte. Diese
     * ergeben einen Teil der
     * Vereinzelungseinheit.
     */
    public void paint(Graphics g) {
        // Röhre
        g.setColor(Color.black);
        g.fillRect(90,5,5,240);
        g.fillRect(135,5,5,200);
        g.fillRect(90,245,145,5);
        g.fillRect(135,205,100,5);

        // Bälle
        g.setColor(Color.red);
        g.fillOval(95,80,40,40);
        g.setColor(Color.yellow);
        g.fillOval(95,120,40,40);
    }
}
```

Frames: Beispiel

Ausgabe des Hauptprogramms von **`FrameBeispiel`**



Applets

- Eine der großen Attraktionen der Architektur von Java mit JVM und Byte-Code ist es, dass Java-Programme auch sehr einfach über das Internet geladen und ausgeführt werden können, da sie nur von der Java Laufzeitumgebung abhängen
- Ein Web-Browser mit integrierter JVM, der in einer Web-Seite die Adresse eines geeigneten Java-Programms findet, kann den Byte-Code übers Netz laden und ihn sofort ausführen
- Die Ausgabe des Programms kann er in der Web-Seite dort anzeigen, wo die Adresse stand
  - Dadurch werden Web-Seiten „lebendig":
    - ⇨ Sie wandeln sich von statischen Dokumenten zu Dokumenten mit dynamischem und auch interaktiv nutzbarem Inhalt

Applets

- Damit ein Java-Programm aber sinnvoll in einem Browser leben kann, muss es die spezielle Form eines **Applet** haben
  - Wohl ein Kunstwort für „kleine Anwendung“ (application)
- Applets stellen insbesondere eine Möglichkeit dar, AWT-Komponenten in eine Web-Seite zu integrieren
  - Da die Klasse **`Applet`** von der Klasse **`Panel`** des Pakets **`java.awt`** erbt, diskutieren wir Applets hier im Rahmen des AWT, obwohl sie durch das separate Paket **`java.applet`** bereitgestellt werden
- Wenn in einer Web-Seite eine „APPLET“-Kennzeichnung gefunden wird, lädt der Browser den compilierten Java-Byte-Code für die angegebene Klasse, die von der Klasse **`Applet`** abgeleitet sein muss, von einer Internet-Adresse (URL -- uniform resource locator), die ebenfalls in der Web-Seite angegeben ist
  - Dann erzeugt er eine Objekt-Instanz dieser Klasse, reserviert einen zweidimensionalen Bereich in der Web-Seite, den das Objekt kontrolliert, und ruft schließlich die **`init`**-Methode des Objekts auf

Applets

- Die Laufzeitumgebung für Applets ist i. a. eingeschränkter als die für allgemeine Java-Programme
- Aus Sicherheitsgründen sollen Applets in einem fest abgegrenzten sogenannten **Sandkasten** (sandbox) ablaufen
  - In dem potentiell gefährliche Operationen (wie Zugriffe auf Dateien und Netzwerke, oder das Ausführen lokaler Programme) eingeschränkt werden
  - Hierzu gibt es einen in dem Browser definierten **Sicherheitsdienst** (security manager)
    - ⇨ Verifikation der Klassen, die die sandbox definieren, wichtige Aufgabe der Programmverifikation

Applets

- Die `init`-Methode ist die erste von vier Methoden, die für den Lebenszyklus eines Applets definiert sind
- Die Methoden `start` und `stop` werden jedes mal aufgerufen, wenn ein Benutzer die Web-Seite besichtigt oder wieder verlässt
  - Indem etwa der „Forward“ oder „Back“ Knopf im Browser betätigt wird
  - Diese Methoden können also im Gegensatz zu `init` mehrfach aufgerufen werden
  - Wenn eine Seite nicht mehr besichtigt werden kann, wird die `destroy`-Methode des Applets aufgerufen, um evtl. belegte Ressourcen wieder freizugeben
- Bei einem Applet sind also (nach dem Konstruktor der Klasse) die `init` und `start` Methoden Einstiegspunkte der Systemumgebung in den Java-Code und nicht die `main`-Methode wie bei anderen Java-Programmen, die direkt -- ohne über das Internet geladen worden zu sein -- von einem Byte-Code-Interpreter (wie dem java-Programm im SDK) ausgeführt werden
- Um im Fenster, das durch das Applet erzeugt wird, etwas anzeigen zu können, muss wiederum die `paint`-Methode überschrieben werden

Applets: Beispiel

- Das folgende Applet gibt die Zeichenkette „Hello world!“ in dem Fenster aus, das durch das Applet erzeugt wird und zwar an der Position (50, 25) im Koordinatensystem des Fensters

```
import java.applet.Applet;
import java.awt.Graphics;

public class HelloWorld extends Applet {
    public void paint(Graphics g) {
        g.drawString("Hello world!", 50, 25);
    }
}
```

Container

- `Container` dienen zur Gruppierung von AWT-Komponenten
  - Sind selber eine AWT-Komponente
    - ⇨ „Rekursion“ daher möglich: Gruppierungen von gruppierten Komponenten
- Jeder Container besitzt einen `LayoutManager`, der für die Anordnung der AWT-Komponenten verantwortlich ist
  - Der Programmierer spezifiziert die relativen Positionen der Komponenten
  - Ihre absolute Positionierung und Dimensionierung bleibt dem Layout-Manager überlassen
    - ⇨ Dieser versucht, eine „günstige“ Lösung in Abhängigkeit von der Fenstergröße und den abstrakten Vorgaben des Programmierers zu finden

Container

- Folgende grundlegende Methoden werden zur Verfügung gestellt
  - **`void setLayout(LayoutManager m)`** setzt den für den Container verantwortlichen LayoutManager
  - **`void add(Component c, . . . )`** fügt in Abhängigkeit vom **`LayoutManager`** die Komponente in den Container ein
  - **`void remove(Component)`** entfernt die Komponente aus dem Container.

Container

- Unterstützt werden folgende Layout-Typen:
  - **`BorderLayout`** kann zur Gruppierung von Komponenten an den Rändern des Containers benutzt werden
    - ⇨ Die vier Ränder werden mit **`NORTH`**, **`EAST`**, **`SOUTH`** bzw. **`WEST`** bezeichnet, wobei diese Himmelsrichtungen der Anordnung auf Landkarten entsprechen
      - – Heutigen Landkarten, bei denen Norden oben ist, Osten rechts usw. ☺
    - ⇨ Das Innere eines solchen Containers wird mit **`CENTER`** bezeichnet
  - **`CardLayout`** ergibt eine spielkartenförmige Anordnung der Komponenten
  - **`FlowLayout`** ergibt eine „fließende“ Anordnung, die linksbündig (**`FlowLayout.LEFT`**), zentriert (**`FlowLayout.CENTER`**) oder rechtsbündig (**`FlowLayout.RIGHT`**) sein kann
  - **`GridLayout`** richtet die Komponenten an einem Gitter aus
  - **`GridBagLayout`** dient zur einer flexiblen horizontalen und vertikalen Anordnung von Komponenten, die nicht alle von der gleichen Größe zu sein brauchen

Graphische Komponenten: Beispiel zu Layout-Managern

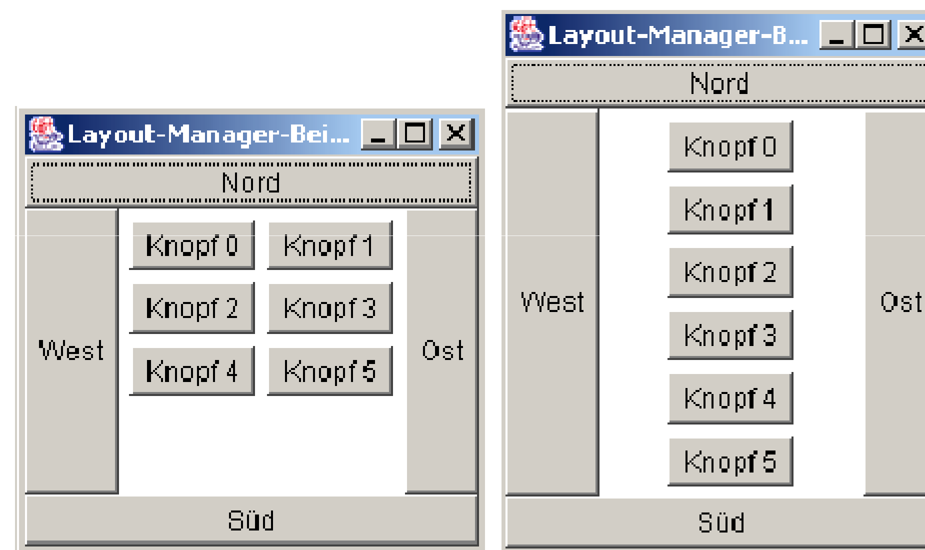
```java
/**
 * Beispielprogramm zur Verwendung von Layout-Managern.
 */
import java.awt.*;

public class LayoutBeispiel extends Frame {
    public static void main(String[] args) {
        Frame f = new Frame("Layout-Manager-Beispiel");

        f.setSize(210,200);
        // Fenstergröße für zweites Beispiel
        // f.setSize(200,245);

        // überflüssig, da default:
        //f.setLayout(new BorderLayout());

        // Vier Buttons an der Rändern
        f.add(new Button("Nord"),BorderLayout.NORTH);
        f.add(new Button("Ost"),BorderLayout.EAST);
        f.add(new Button("Süd"),BorderLayout.SOUTH);
        f.add(new Button("West"),BorderLayout.WEST);
```

```java
        // Im Zentrum ein Container
        // mit Flow-Layout
        Container c = new Container();
        c.setLayout(new FlowLayout());
        for(int i=0; i<6; i++)
            c.add(new Button("Knopf "+i));
        c.setVisible(true);
        f.add(c,BorderLayout.CENTER);

        f.setVisible(true);
    }
}
```


universitätbonn

Graphische Komponenten: Beispiel zu Layout-Managern

- Ausgaben von **`LayoutBeispiel`**



Links: Ergebnis nach `f.setSize(210,200);` Rechts: Ergebnis nach `f.setSize(200,245);`

Ereignisse (events)

- Für die Interaktion mit einem Benutzer ist die Behandlung äußerer **Ereignisse** (events) wesentlich
  - Dies kann z. B. eine Mausbewegung, ein Mausklick oder das Drücken einer Taste sein usw.
- Hierfür stellt das AWT einen entsprechenden Rahmen zur Verfügung, den wir im folgenden skizzieren wollen
  - Das zugrunde liegende Modell wurde im JDK 1.1 gegenüber der Vorgängerversion im JDK 1.0.2 verändert und lieferte eine der Hauptquellen für Inkompatibilitäten zwischen den unterschiedlichen Versionen
- Verschiedene Klassen von Ereignissen sind in Java 1.1 (und höher) durch verschiedene Java-Klassen repräsentiert
  - Jede Ereignisklasse ist eine Unterklasse von **`java.util.EventObject`**
    - ⇨ Ereignisobjekte tragen gegebenenfalls weitere nützliche Information in sich, die das Ereignis weiter charakterisiert
      - – z. B. die X- und Y-Koordinate bei **`MouseEvent`**, das zu einem Mausklick gehört

Ereignisse (events)

- AWT-Events
  - AWT-Ereignisse sind Unterklassen von **`java.awt.AWTEvent`**
    - ⇨ Sie sind im Paket **`java.awt.event`** zusammengefasst
  - AWT-Komponenten können folgende Ereignisse erzeugen:

```
ActionEvent      AdjustmentEvent
ComponentEvent   ContainerEvent
FocusEvent       ItemEvent
KeyEvent         MouseEvent
TextEvent        WindowEvent
```

Ereignisquellen und Ereignisempfänger

- Jedes Ereignis wird von einer **Ereignisquelle** (event source) generiert
  - dies ist ein anderes Objekt, das man mit `getSource()` erhält
- Die Ereignisquelle liefert ihre Ereignisse an interessierte Parteien, die **Ereignisempfänger** (event listener) aus, die dann selbst eine geeignete Behandlung vornehmen
- Die Zuordnung zwischen Quelle und Empfängern darf nicht im Programmcode statisch fixiert werden, sondern sie muss sich zur Laufzeit dynamisch ändern können
  - Dazu verwendet man ein elegantes Prinzip, das wir (in ähnlicher Form) schon beim generischen Programmieren kennen gelernt hatten
- Die Ereignisempfänger müssen sich bei der Quelle an- und abmelden
  - Ereignisquelle implementiert hierzu eine geeignete Schnittstelle
  - Die Ereignisquelle unterhält eine Liste von angemeldeten Ereignisempfängern
  - Hat Ereignisquelle ein Ereignis generiert, dann schreitet sie die Liste der Empfänger ab und ruft auf jedem Empfänger eine Methode auf, der sie das Ereignisobjekt (per Referenz) übergibt
    - ⇨ Hierzu ist für jede Ereignisklasse eine entsprechende Schnittstelle für Empfänger (event listener interface) spezifiziert

Ereignisquellen und Ereignisempfänger

- **Beispiel:** Bei einer AWT-Komponente können ein oder mehrere event listener mittels einer Methode **`add`*`Type`*`Listener()`** registriert werden
  - Im folgenden Programmfragment wird ein **`ActionListener`** bei einer **`Button`**-Komponente registriert
- Ein event listener interface definiert Methoden, die beim Auftreten eines Ereignisses automatisch aufgerufen werden und die man so implementieren kann, dass sie das Ereignis behandeln
- Jeder Empfänger muss das zum Ereignis gehörige Interface implementieren, damit die Quelle ihn aufrufen kann
- In einem großen über viele Rechner verteilten System kann der vor Ort installierte event listener das Ereignis auch erst einmal vor verarbeiten und danach die relevante Information über das Netz zum wirklichen Bearbeiter weiterschicken
- Jeder Empfänger implementiert dies individuell so, dass die für ihn typische Bearbeitung des übergebenen Ereignisses stattfindet

```
import java.awt.*;
import java.awt.event.*;
// ...
Button button;
// ...
button.addActionListener(this);
```

Ereignisquellen und Ereignisempfänger

- Als Beispiel wollen wir das zum Paket **`java.awt.event`** gehörige Interface **`WindowListener`** beschreiben
- In der Schnittstelle sind die folgenden Methoden spezifiziert:
  - **`void windowOpened(WindowEvent e)`**
  - **`void windowClosing(WindowEvent e)`** Diese Methode wird aufgerufen, wenn vom Benutzer ein sogenannter Close-Request abgesetzt wurde
    - ⇨ Wie ein Close-Request genau abgesetzt wurde, hängt von der Laufzeitumgebung und dem Fenstersystem ab; es könnte z. B. das Schließe-Symbol des umgebenden System-Fensters angeklickt worden sein
    - ⇨ Das Fenster muss explizit mit der dispose-Methode des Fensters geschlossen werden
  - **`void windowClosed(WindowEvent e)`**
  - **`void windowIconified(WindowEvent e)`**
  - **`void windowDeiconified(WindowEvent e)`**
  - **`void windowActivated(WindowEvent e)`**
  - **`void windowDeactivated(WindowEvent e)`**
- In den folgenden größeren Beispielen wird das **`WindowListener`**-Interface jeweils von einer Erweiterung der **`Frame`**-Klasse implementiert, um das Fenster, in dem gezeichnet wird, wieder schließen zu können

Adapter-Klassen im AWT

- Bei der Verwendung eines Listener-Interfaces müssen wie bei allen Interfaces immer alle Methoden implementiert werden
  - Wenn man sich nur für bestimmte Events interessiert, kann man die anderen Methoden als sog. „Hohlkopf"- oder „Atrappen"-Methoden (dummy method) mit leerem Rumpf implementieren
    - ⇨ Da keine der in den Listener-Interfaces spezifzierten Methoden einen Rückgabewert besitzt
      - – Sonst müsste ein „Dummy"-Rückgabewert vom geeigneten Typ zurückgegeben werden
- Um dem Programmierer die Arbeit zu erleichtern, stellt das AWT für alle Interfaces, die mehr als eine Methode definieren, eine **Adapter-Klasse** zur Verfügung
  - Adapter-Klassen haben alle Interface-Methoden als leere Methoden implementiert
  - Bei der Verwendung von Adapter-Klassen müssen also nur die benötigten Methoden überschrieben werden
  - Die eigene Klasse muss von der Adapter-Klasse abgeleitet werden, so dass Adapter-Klassen nur dann verwendet werden können, wenn die eigene Klasse nicht von einer anderen Klasse erben soll

Ein Beispiel: Ein Rahmen zum Zeichnen reeller Funktionen

- Wir definieren eine abstrakte Klasse ***`FunctionPlotter`***, mit deren Hilfe wir eine mathematische Funktion zeichnen können
  - genauer gesagt eine eindimensionale Funktion f: IR → IR
- Diese Klasse kann insbesondere zur Veranschaulichung von Funktionen aus einer begleitenden Mathematik-Vorlesung (speziell Analysis I) verwendet werden
  - Sie eignet sich auch sehr gut für eigene Erweiterungen zur Übung, z. B. auf das Zeichnen mehrerer Funktionen im gleichen Bild
- Da wir beliebige einstellige reelle Funktionen zeichnen wollen, definieren wir in der Klasse ***`FunctionPlotter`*** eine abstrakte Funktion **`f`**: **`double`** → **`double`**

Ein Beispiel: Ein Rahmen zum Zeichnen reeller Funktionen

- Die Klasse `FunctionPlotter` wird von der Klasse `java.awt.Frame` abgeleitet
- Die virtuelle Funktion `paint` ist diejenige Funktion in der Klasse `Frame`, die vom Windows-System aufgerufen wird, wenn ein Frame-Objekt gezeichnet wird
  - Wir überschreiben daher diese Methode in `FunctionPlotter`
- In unserer Implementierung der Methode `paint` benutzen wir die rein virtuelle Funktion `f`
  - Damit in Spezialisierungen von `FunctionPlotter` diese abstrakte Methode durch verschiedene Realisierungen ersetzt werden kann, ohne dass `paint()` reimplementiert werden muss, hat `paint()` also die Funktion `f` als einen (impliziten) Parameter
    - ⇨ Dieser ist nicht in der Aufrufschnittstelle sichtbar
      - – Mögliches Problem, siehe weiter vorne
  - Die Methode `paint` ist somit eine **Funktion höherer Stufe** (higher-order function) in diesem Beispiel

Ein Beispiel: Ein Rahmen zum Zeichnen reeller Funktionen

- Dieser implizite Funktionsparameter `f` von `paint` kann aber im Methodenkopf nicht gekennzeichnet werden
  - Etwa durch einen Parameter vom Typ eines Interfaces, in dem der Funktionsparameter `f` spezifiziert ist
    - ⇨ Wie dies sonst sehr zu empfehlen ist
    - ⇨ Und auch wichtigen Prinzipien wie die des Information Hiding entspricht
  - Signatur von `paint` ist in Basisklasse des Frameworks definiert
    - ⇨ Und kann in abgeleiteter Klasse nicht geändert werden
  - Wichtig: zumindest in Dokumentationskommentaren impliziten Funktionsparameter genau erläutern
- Dieses Beispiel zeigt eines der Probleme (und Herausforderungen) bei der Benutzung von Frameworks

Ein Beispiel: Ein Rahmen zum Zeichnen reeller Funktionen

- Code der abstrakten Basisklasse ***FunctionPlotter***

```
Import java.awt.*;

public abstract class FunctionPlotter
        extends java.awt.Frame {
    // number of samples for discretisation
    private int noSamples;
    // range to be displayed
    private double minx;
    private double maxx;
    private double miny;
    private double maxy;

    /**
     * Abstract method to be implemented in
     * child classes to evaluate the function
     * which is to be plotted
     */
    public abstract double f(double x);
```

```
    /**This method is invoked automatically by
     * the runtime-environment whenever the
     * contents of the window have to be drawn.
     * (Once at the beginning after the window
     * is displayed on the screen and once
     * every time a redraw event occurs.)
     * Plotting of the graph of f() is
     * implemented here.
     */
    public void paint(Graphics g) {
        int i,x1,x2,y1,y2;
        double x;
        double step=(maxx-minx)/(noSamples-1);
        x1=getPixelXfromWorldX(minx);
        y1=getPixelYfromWorldY(f(minx));
        x = minx + step;
        for(i=1; i<noSamples;
                 i++,x+=step,x1=x2,y1=y2) {
            x2=getPixelXfromWorldX(x);
            y2=getPixelYfromWorldY(f(x));
            g.drawLine(x1,y1,x2,y2);
        }
    }
}
```

Ein Beispiel: Ein Rahmen zum Zeichnen reeller Funktionen

- Um eine Funktion, wie z. B. `x → x*cos(x)` plotten zu können, müssen wir nur eine Klasse definieren,
  - die `FunctionPlotter` erweitert und
  - in der `f` durch die gewünschte Funktion implementiert ist
- Außer einem entsprechenden Konstruktor muss in dieser Klasse keine weitere Methode definiert sein

Ein Beispiel: Ein Rahmen zum Zeichnen reeller Funktionen

- Beispiele zweier erweiternder Klassen

```
/**
 * Concrete Class which implements function
 * f() to calculate x times cos(x)
 */
public class XCosXPlotter
    extends FunctionPlotter {
    /**
     * @param min left border of x-range
     * @param max right border of x-range
     * @param nsamples Number of Samples
     *        used for discretisation of f().
     */
    public XCosXPlotter( int nsamples,
                   double min, double max) {
        super(nsamples,min,max);
    }
    /**
     * Implementation of the function whose
     * graph is to be plotted.
     */
    public double f(double x)
    { return x*Math.cos(x); }
}
```

```
/**
 * Concrete Class which implements function
 * f() to calculate sin(x)
 */
public class SinXPlotter
    extends FunctionPlotter {
    /**
     * @param min left border of x-range
     * @param max right border of x-range
     * @param nsamples Number of Samples
     *        used for discretisation of f().
     */
    public SinXPlotter( int nsamples,
                  double min, double max) {
        super(nsamples,min,max);
    }
    /**
     * Implementation of the function whose
     * graph is to be plotted.
     */
    public double f(double x)
    { return Math.sin(x); }
}
```

Ein Beispiel: Ein Rahmen zum Zeichnen reeller Funktionen

- Damit wir das Graphik-Fenster, das für die Klasse ***FunctionPlotter*** geöffnet wird, wieder schließen können, müssen wir die Methode **windowClosing** des Interfaces **WindowListener** implementieren
  - Es müssen alle in **WindowListener** deklarierten Methoden implementiert werden, egal ob wir auf die entsprechenden Ereignisse reagieren wollen oder nicht
  - Die nicht benötigten Methoden implementieren wir mit leerem Rumpf (dummy method)