

# Das “Abstract Window Toolkit” (AWT)

Und einige komplexere Beispiele,  
die Softwareentwicklung von der  
objektorientierten Analyse über das Design bis  
zur Implementierung erläutern

Foliensatz von A. Weber zur Vorlesung Informatik I, Bonn, 2002/03  
Überarbeitet von W. Küchlin zu Informatik I, Tübingen 2003/04

- Das **Java AWT** (*abstract window toolkit*) ist ein **Framework** zum Programmieren von **graphischen Benutzeroberflächen** (*graphical user interfaces -- GUIs*)
- 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

- **Objektorientiertes Rahmenwerk**  
(*object-oriented framework*)
  - eine halbfertige Anwendung
    - Rahmen ist fertig, Inhalt fehlt
  - Inhalt wird vom Nutzer über **Haken** (*hooks*) eingehängt
    - Über Ableitung von Haken-Klassen oder Delegation an implementierte Interfaces
  - Framework hat die Kontrolle über die Ausführung
    - ruft den Code des Benutzers über die Haken auf  
(*don't call us – we'll call you*)

- Wir können relativ leicht ein GUI mit AWT 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**
- Über **höhere objektorientierte Mechanismen** (Vererbung und dynamisches Binden) wird der **spezielle Code nahtlos** in den durch das **AWT** gegebenen **Rahmen integriert**
  - und man kann damit den **Code** des AWT Rahmenwerks **wieder verwenden**
  - Das **Framework ruft** den **Darstellungscode** des **Benutzers auf**

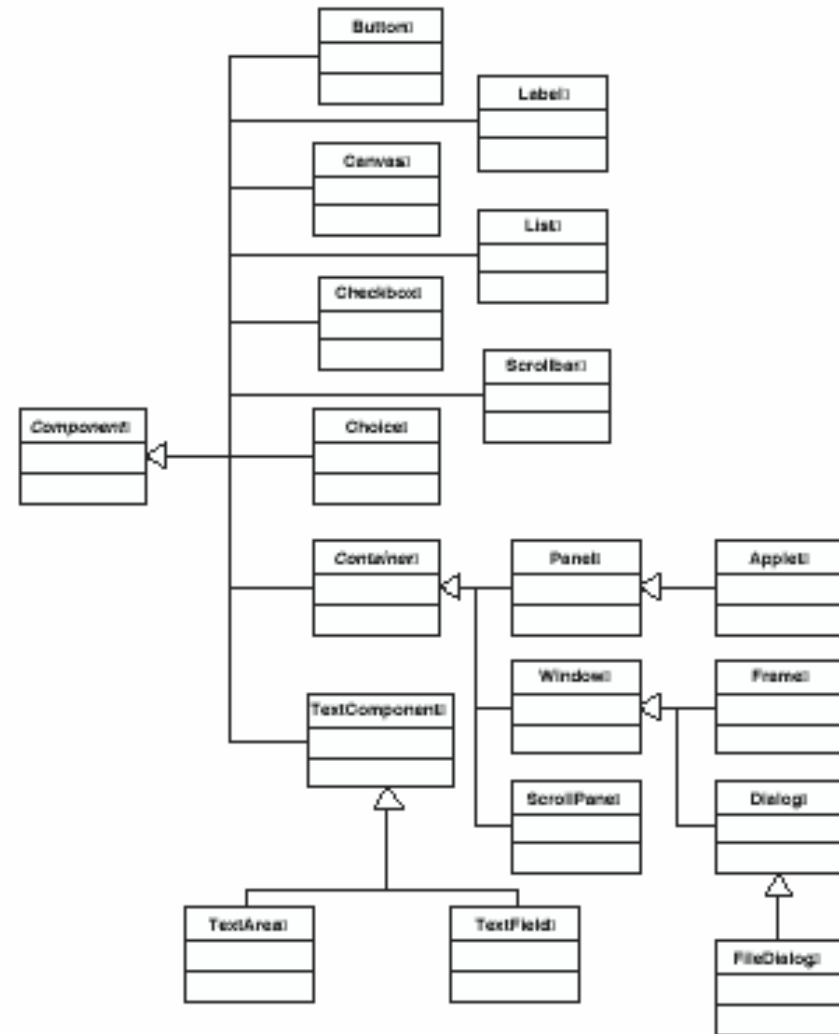
- **Bemerkung:**

- Wir verwenden das AWT, obwohl es **neuere Klassenbibliotheken** für die **GUI-Programmierung in Java** gibt
  - Eclipse **SWT** (Standard Widget Toolkit): *system flavored look-and-feel*
  - Die Java Standard-Bibliothek **Swing**
  - Swing ist eine Erweiterung des AWT und bietet technische Vorteile, wie etwa ein **systemunabhängiges** aber **benutzerdefinierbares** von graphischen Elementen, (*system independent look-and-feel*)
- Aufgrund seiner einfacheren Struktur ist das AWT aber für unsere Zwecke geeigneter als Swing
- Die **Prinzipien des Entwurfs graphischer Komponenten** und der **Ereignisbehandlung** sind in AWT und Swing **gleich**.

# 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 abstrakter Basisklasse **Component**.



### *Graphische Komponenten: Funktionalität von Component*

- Die Klasse **Component** 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 **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

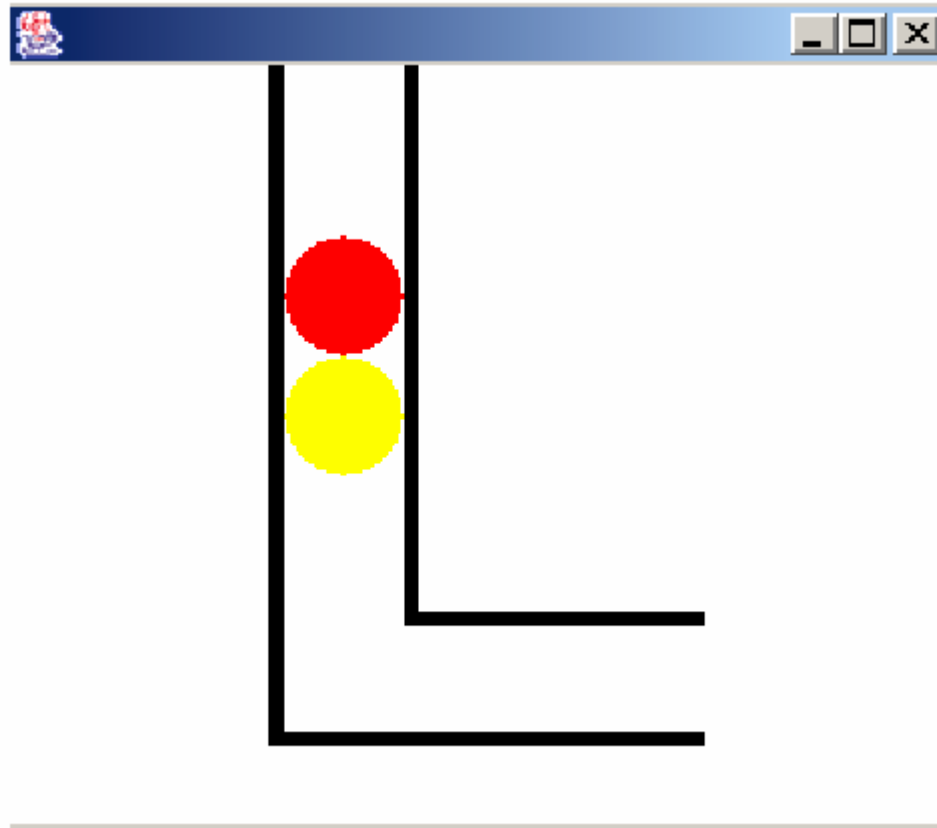
- Ein **Graphik-Kontext** wird vom **Rahmenwerk** des **AWT** erzeugt, (das die Kontrolle hat)
  - Über die „**Haken**“, die ein Graphics-Objekt als Parameter besitzen, kann im **Anwendungsprogramm** aber auf den vom **AWT erzeugten Graphik-Kontext** zugegriffen werden
    - Ein wichtiger Haken ist etwa 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 (vom Typ **int**) 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*

- 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**

- 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  $y+height-1$
  - **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.

- 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

Ausgabe des Hauptprogramms von **FrameBeispiel**



- 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 Fenster der Groesse 320 x 280.
     */
    FrameBeispiel() {
        setSize(320,280);
        setVisible(true);
    }

    /**
     * Zeichne einige einfache graphische Objekte.
     * Ergeben 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);
    }

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

- 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

- 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

- 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

- 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

- 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** 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

- 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.
- 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

```
/**
 * 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);

        // Vier Buttons an den Rändern
        //f.setLayout(new BorderLayout());
        // überflüssig, da default
        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);

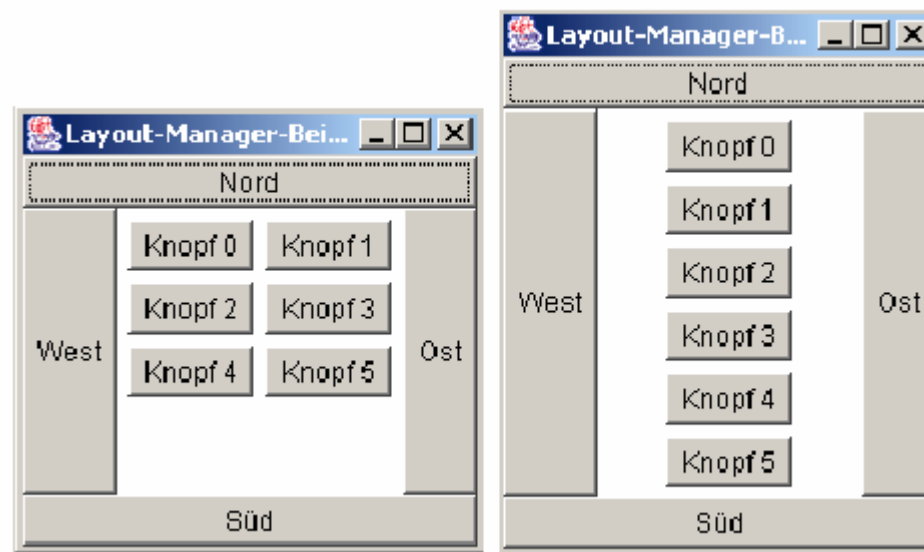
        // Im Zentrum ein Container mit Flowlayout
        Container c = new Container();
        c.setLayout(new FlowLayout());
        for (int i=0; i<6; i++) {
            c.add(new Button("Knopf "+i));
        }
        c.setVisible(true);

        f.add(c,BorderLayout.CENTER);

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

## 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

- 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 **addTypeListener()** 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
  - Jeder Empfänger implementiert dies individuell so, dass die für ihn typische Bearbeitung des übergebenen Ereignisses stattfindet
- In einem großen über viele Rechner verteilten System kann der vor Ort installierte **event listener** das Ereignis auch erst einmal vorverarbeiten und danach die relevante Information über das Netz zum wirklichen Bearbeiter weiterschicken

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

### Ereignisquellen und Ereignisempfänger: Die Schnittstelle WindowListener

- 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

- 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 spezifizierten 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  $f: \mathbb{R} \rightarrow \mathbb{R}$
- 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  $\rightarrow$  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 Aufrufchnittstelle sichtbar
      - Mögliches Problem, siehe weiter vorne
  - Die Methode **paint** ist somit eine **Funktion höherer Stufe** (higher-order function)

### *Ein Beispiel: Ein Rahmen zum Zeichnen reeller Funktionen*

- Code der abstrakten Basisklasse FunctionPlotter

```
public abstract class FunctionPlotter
    extends java.awt.Frame {

    private int noSamples;
        // number of samples to use for discretisation
    private double minx;
        // left border of x-range
    private double maxx;
        // right border of x-range
    private double miny;
        // minimal value of f() within the specified range
    private double maxy;
        // maximal value of f() within the specified range

    // other attributes, e.g. for Window size

    // Constructors, other Methods,
    // e.g. getPixelXfromWorldX, getPixelYfromWorldY
    // ...

    /**
     * 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;
        int x1,y1,x2,y2; // Pixel coordinates
        double step=(maxx-minx)/(noSamples-1);
        double x;

        x1=getPixelXfromWorldX(minx);
        y1=getPixelYfromWorldY(f(minx));
        // here we use the abstract f
        x=minx+step;

        for (i=1; i<noSamples; i++,x+=step,x1=x2,y1=y2) {
            x2=getPixelXfromWorldX(x);
            y2=getPixelYfromWorldY(f(x));
            // here we use the abstract f
            g.drawLine(x1,y1,x2,y2);
        }
    }
}
```

## *Ein Beispiel: Ein Rahmen zum Zeichnen reeller Funktionen*

- Um eine Funktion, wie z. B.  $x \rightarrow x \cdot \cos(x)$  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)

### *Ein Beispiel: Ein Rahmen zum Zeichnen reeller Funktionen*

```
import java.awt.Frame;
import java.awt.Graphics;
import java.awt.Event;
import java.awt.AWTEvent;
import java.awt.event.WindowListener;
import java.awt.event.WindowEvent;

public abstract class FunctionPlotter
    extends Frame
    implements WindowListener {

    public FunctionPlotter
        (int nsamples, double min, double max)
    { // ...

        // add as WindowListener to react to
        // WINDOW_DESTROY events
        addWindowListener(this);
    }

    // The attributes and methods of FunctionPlotter.
    // ...

    // The attributes and methods of FunctionPlotter.
    // ...

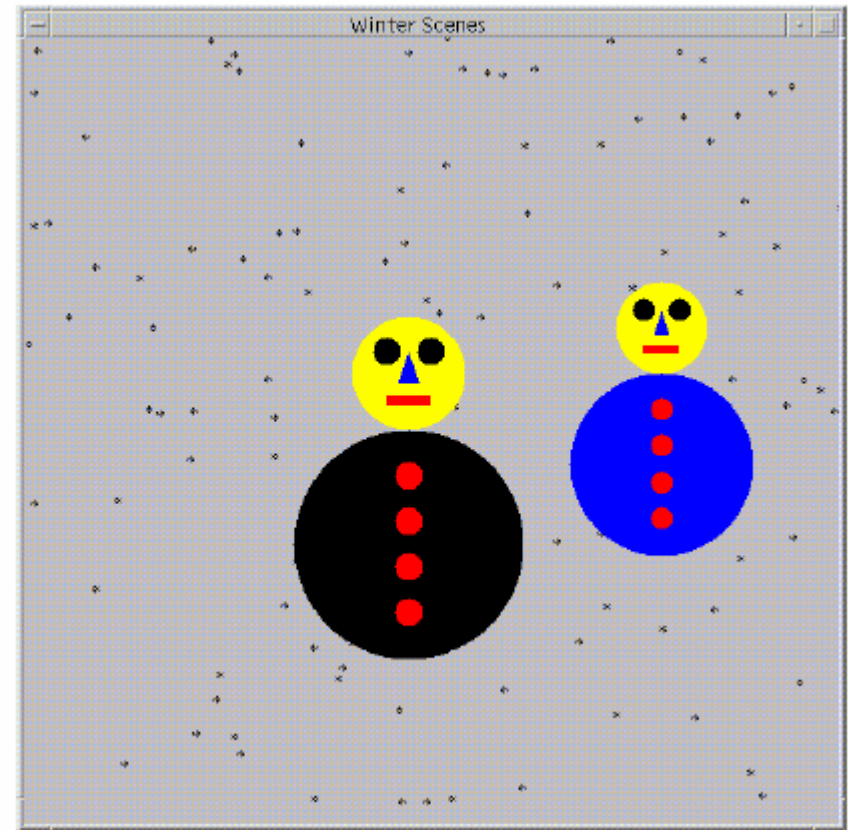
    /**
     * Event handler of the window to process
     * WINDOW_DESTROY events, which should cause
     * the application to terminate.
     */
    public void windowClosing(WindowEvent e) {
        setVisible(false);
        dispose(); // destroy the window
        System.exit(0); // terminate execution
    }

    // Other methods a WindowListener needs to implement.
    // Implemented as dummies since WINDOW_DESTROY is the
    // only window event we want to react to.

    public void windowActivated(WindowEvent e) { }
    public void windowClosed(WindowEvent e) { }
    public void windowDeiconified(WindowEvent e) { }
    public void windowIconified(WindowEvent e) { }
    public void windowOpened(WindowEvent e) { }
    public void windowDeactivated(WindowEvent e) { }
}
```

## *Ein größeres Beispiel: Darstellung einer Winterlandschaft*

- Wir wollen eine Klassenbibliothek schreiben, die es uns ermöglicht, eine Winterlandschaft, wie sie rechts dargestellt ist, in einem Fenster zu zeichnen



## *Ein größeres Beispiel: Darstellung einer Winterlandschaft*

- **Anforderungsanalyse**

- Die Klassenbibliothek soll verschiedene Objekte, die zu einer Winterlandschaft gehören - wie etwa Schneemänner und Schneeflocken - in einem Fenster darstellen
- Ein **Schneemann** besteht aus **einem Gesicht** und **einem Bauch**
  - Der **Bauch** eines **Schneemanns** ist **rund** und **enthält einige runde Knöpfe**
  - Das **runde Gesicht** besteht aus **zwei kreisförmigen Augen**, **einem rechteckigen Mund** und **einer dreieckigen Nase**

## *Ein größeres Beispiel: Darstellung einer Winterlandschaft*

- **Objektorientierte Analyse und Design**
  - Analyse relativ einfach, wird hier übersprungen
  - Konzentrieren uns hier auf die Verfeinerung in der Design-Phase
    - Der wir später die Implementierung folgen lassen
  - Die Winterlandschaft wird als Objekt einer Klasse **SnowScene** repräsentiert,
    - die ein **Graphik-Fenster** enthält, in dem die Winterszene gezeichnet wird,
    - und eine **Liste von geometrischen Objekten**, die gezeichnet werden sollen
  - Das Graphik-Fenster realisieren wir in einer Klasse **DrawWindow**, die die Klasse **Frame** aus **java.awt** erweitert
  - Die virtuelle Funktion **paint** von **Frame** müssen wir in **DrawWindow** entsprechend implementieren
  - Damit ein Fenster, das ein Objekt der Klasse **DrawWindow** erzeugt, wieder per Mausklick geschlossen werden kann, realisiert **DrawWindow** auch noch das Interface **WindowListener** aus **java.awt.event**

## *Ein größeres Beispiel: Darstellung einer Winterlandschaft*

- Objektorientierte Analyse und Design (Forts.)
  - Um unsere Winterszenen einfach erweitern zu können, wollen wir eine beliebige Liste von geometrischen Objekten zeichne
  - Wir führen deshalb eine (abstrakte) Basisklasse **Shape** ein, von der alle unsere geometrischen Objekte in der **Winterlandschaft** abgeleitet werden sollen
    - Diese Klasse besitzt die (virtuelle) Methode **draw**, die in den Klassen, die **Shape** erweitern, entsprechend implementiert sein muss
      - Da diese Methode eine virtuelle Funktion darstellt, können wir die **paint** Methode in **DrawWindow** implementieren, ohne alle möglichen Erweiterungen der Klasse **Shape** zu kennen!

## *Ein größeres Beispiel: Darstellung einer Winterlandschaft*

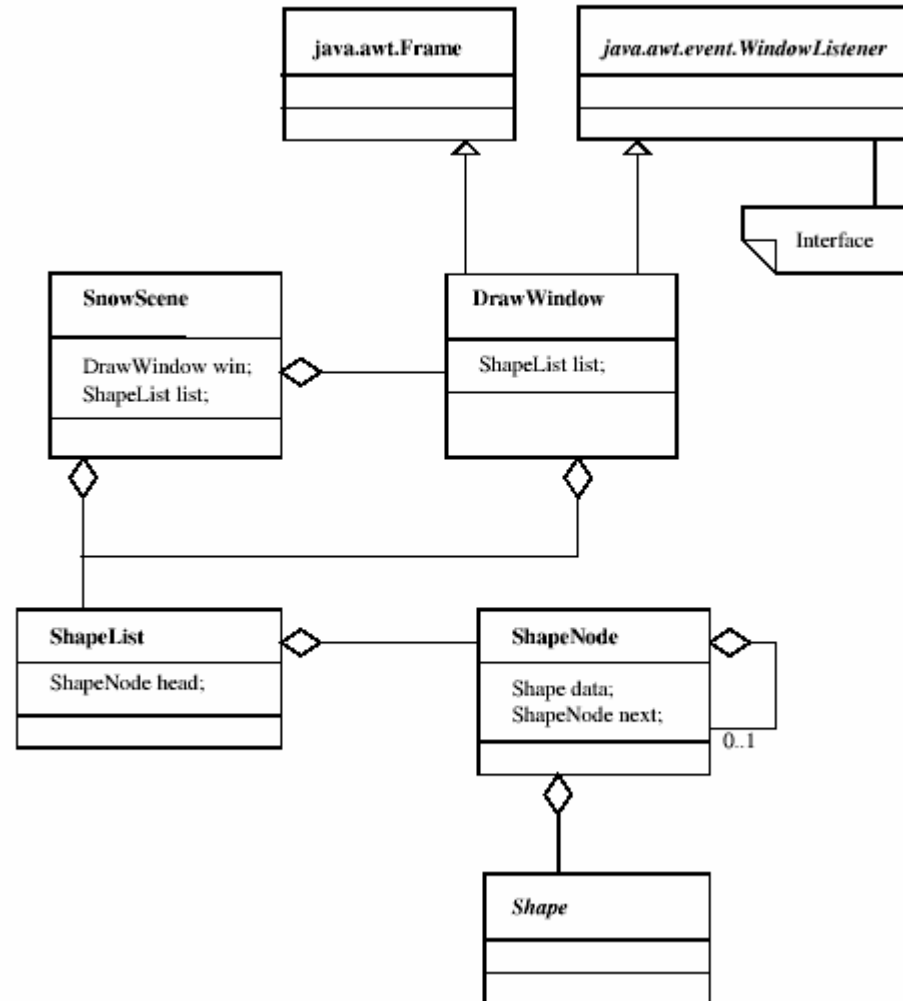
- **Objektorientierte Analyse und Design (Forts.)**
  - Die Liste der geometrischen Objekte vom Typ **Shape** realisieren wir als **einfach verkettete lineare Liste**
  - Entsprechend früher gewählten **Namenskonvention** hat diese **Klasse** den **Namen ShapeList**, und jede solche Liste besteht aus Knoten vom Typ **ShapeNode**
  - In unserer Winterlandschaft benötigen wir bislang Objekte vom Typ **Snowman** und vom Typ **Snowflake**
  - Ein Objekt vom Typ **Snowman** **aggregiert** zwei Felder,
    - eines für **Body**
    - und eines für **Face**
  - Diese beiden Klassen sind auch **geometrische Objekte**, die wir zeichnen wollen, sie sind also auch **Erweiterungen der abstrakten Basisklasse Shape**

## Ein größeres Beispiel: Darstellung einer Winterlandschaft

- Objektorientierte Analyse und Design (Forts.)
  - Ein Objekt vom Typ **Body** aggregiert einen **Kreis** für den eigentlichen Körper und einige (kleinere) Kreise für die Knöpfe
  - Wir benötigen daher auch eine Klasse **Circle**
    - Da die **Anzahl** der **Knöpfe** **nicht fest bestimmt** ist, notieren wir eine allgemeine **1:n-Beziehung** zwischen **Body** und **Circle**
  - Das **Gesicht** - ein Objekt vom Typ **Face** - wird durch einen umfassenden **Kreis** dargestellt, der **zwei kleinere Kreise** als Augen enthält
    - Die **Aggregation** zwischen **Face** und **Circle** ist also vom Typ **1:3**
    - Das **Gesicht** enthält ferner einen **rechteckigen Mund** und eine **dreieckige Nase**
    - Wir benötigen daher noch Erweiterungen von **Shape** zu einer Klasse **Rectangle** und zu einer Klasse **Triangle**
      - Da ein **Gesicht** **genau einen Mund** und **genau eine Nase** enthält, sind diese **Aggregationen** vom Typ **1:1**

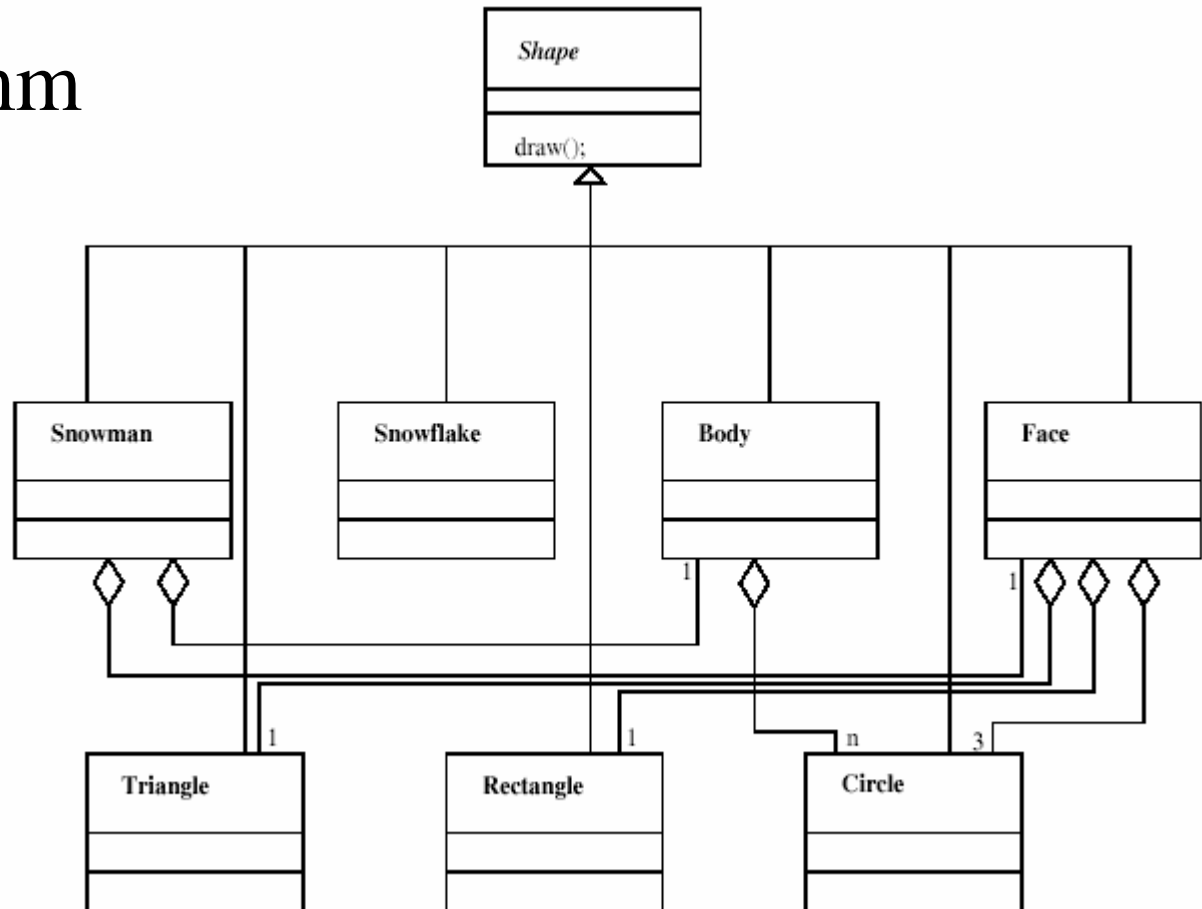
## Ein größeres Beispiel: Darstellung einer Winterlandschaft

Klassendiagramm zu  
„Winterlandschaft“  
(**SnowScene**)  
verfeinert durch die Design-  
Phase



## Ein größeres Beispiel: Darstellung einer Winterlandschaft

- Klassendiagramm zu „Shape“



## *Ein größeres Beispiel: Darstellung einer Winterlandschaft*

- Bei der **Implementierung** der **Shape-Klassen** beginnen wir bei der **abstrakten Basisklassen**
- Die Shape-Klassen, die keine anderen Shape-Klassen aggregieren, können danach unabhängig voneinander implementiert werden
- Die Shape-Klassen, die andere aggregieren (wie etwa Face) sind sinnvollerweise erst danach zu implementieren

### *Ein größeres Beispiel: Darstellung einer Winterlandschaft*

- Code für abstrakte Basisklasse **Shape**

```
import java.awt.Graphics;
import java.awt.Color;

/**
 * This is a class that defines an
 * abstract shape. It ensures that every subclass
 * implements a draw method.
 */
public abstract class Shape {
    protected Color
        color = Color.black; // color of the object

    /**
     * Gives back the color.
     * @return The color of the Shape.
     */
    public Color getColor() {
        return color;
    }
}
```

```
/**
 * Sets the color of the Shape.
 * @param col --
 */
public void setColor(Color col) {
    color = col;
}

/**
 * An abstract method which ensures that every
 * subclass of Shape implements a draw method.
 * @param g --
 */
public abstract void draw(Graphics g);
}
```

## Ein größeres Beispiel: Darstellung einer Winterlandschaft

- Einfache Erweiterungen der Shape-Klasse

*Kreise.*

```
import java.awt.Graphics;
import java.awt.Point;

/**
 * This extension of Shape defines a Circle
 * by point and radius.
 */
public class Circle extends Shape {
    private Point m; // center
    private int r; // radius

    /**
     * Creates a new Circle.
     * @param midpoint --
     * @param radius radius >= 0.
     */
    public Circle(Point midpoint, int radius) {
        m = midpoint;
        r = radius;
    }

    /**
     * Draw method for Circle. Paints the Circle
     * as a filled oval.
     * @param g --
     */
    public void draw(Graphics g) {
        g.setColor(color);
        g.fillOval(m.x-r,m.y-r,2*r,2*r);
    }
}
```

### *Ein größeres Beispiel: Darstellung einer Winterlandschaft*

- Rechtecke

#### *Dreiecke.*

Die Implementierung für Dreiecke ist analog zu der für Kreise und Rechtecke und soll hier nicht wiedergegeben werden

#### *Rechtecke.*

```
import java.awt.Graphics;
import java.awt.Point;

/**
 * Rectangle, defined by two corners.
 */
public class Rectangle extends Shape {
    private Point min; // lower left corner
    private Point max; // upper right corner

    /**
     * Creates a Rectangle through two points.
     * @param pmin The lower left corner.
     * @param pmax The upper right corner.
     */
    public Rectangle(Point pmin, Point pmax) {
        min = pmin;
        max = pmax;
    }

    /**
     * Draw method for Rectangle. The Rectangle is
     * painted and filled.
     * @param g --
     */
    public void draw(Graphics g) {
        g.setColor(color);
        g.fillRect(min.x,min.y,max.x-min.x,max.y-min.y);
    } }
```

### *Ein größeres Beispiel: Darstellung einer Winterlandschaft*

- Erweiterungen von **Shape**, die **andere Shape-Objekte** aggregieren

```
import java.awt.Graphics;
import java.awt.Point;
import java.awt.Color;
/**
 * This extension of Shape defines the Face
 * of a Snowman with eyes, nose and mouth.
 */
public class Face extends Shape {
    private Circle face;    // the face itself
    private Circle leftEye; // one eye
    private Circle rightEye; // the other eye
    private Triangle nose; // the nose
    private Rectangle mouth; // the mouth

    /**
     * Creates a new Face.
     * @param midpoint --
     * @param radius radius >= 0.
     */
    public Face(Point midpoint, int radius) {
        // The different components are placed relative
        // to the midpoints. Their sizes depend only
        // on the radius.

        // Face
        face = new Circle(midpoint, radius);

        // Eyes
        Point leftEyePos = new Point(midpoint.x,
                                     midpoint.y);
        Point rightEyePos = new Point(midpoint.x,
                                      midpoint.y);
        leftEyePos.translate(-(int)(0.4*radius),
                             -(int)(0.4*radius));
        rightEyePos.translate((int)(0.4*radius),
                              -(int)(0.4*radius));

        leftEye = new Circle(leftEyePos, radius/4);
        rightEye = new Circle(rightEyePos, radius/4);
        leftEye.setColor(Color.yellow);
        rightEye.setColor(Color.yellow);
    }
}
```

### *Ein größeres Beispiel: Darstellung einer Winterlandschaft*

- Fortsetzung des Codes zu **Face**

```
// Nose
Point p1 = new Point(midpoint.x, midpoint.y);
Point p2 = new Point(midpoint.x, midpoint.y);
Point p3 = new Point(midpoint.x, midpoint.y);

p1.translate(-(int)(0.2*radius),
             (int)(0.2*radius));
p2.translate((int)(0.2*radius),
             (int)(0.2*radius));
p3.translate(0, -(int)(0.4*radius));

nose = new Triangle(p1, p2, p3);
nose.setColor(Color.blue);

// Mouth
Point min = new Point(midpoint.x, midpoint.y);
Point max = new Point(midpoint.x, midpoint.y);

min.translate(-(int)(0.4*radius),
              (int)(0.4*radius));
max.translate((int)(0.4*radius),
              (int)(0.6*radius));

mouth = new Rectangle(min, max);
mouth.setColor(Color.red);
}
```

```
/**
 * Sets the color of the entire face.
 * @param col --
 */
public void setColor(Color col) {
    super.setColor(col);
    face.setColor(col);
}

/**
 * Sets the color for both eyes together.
 * @param col --
 */
public void setEyeColor(Color col) {
    leftEye.setColor(col);
    rightEye.setColor(col);
}

/**
 * Sets the color of the nose.
 * @param col --
 */
public void setNoseColor(Color col) {
    nose.setColor(col);
}

/**
 * Sets the color of the mouth.
 * @param col --
 */
public void setMouthColor(Color col) {
    mouth.setColor(col);
}
```

### *Ein größeres Beispiel: Darstellung einer Winterlandschaft*

- Fortsetzung des Codes zu **Face**

```
/**
 * Draw method for Face. Paints all components
 * face, eyes, nose, mouth.
 * @param g --
 */
public void draw(Graphics g) {
    face.draw(g);
    leftEye.draw(g);
    rightEye.draw(g);
    nose.draw(g);
    mouth.draw(g);
} }
```

## *Ein größeres Beispiel: Darstellung einer Winterlandschaft*

- *Schneemann*
  - Der Schneemann besteht aus einem Gesicht und einem Kreis als Bauch
  - Diese Shape-Klasse soll hier ebenfalls nicht wiedergegeben werden

### *Ein größeres Beispiel: Darstellung einer Winterlandschaft*

- **Ein Fenster mit einer Winterlandschaft**
  - Dem **Konstruktor** der folgenden Klasse **DrawWindow** kann neben der Größe des zu zeichnenden Fensters eine Liste von Shape-Objekten mitgegeben werden, die in dem Fenster gezeichnet werden

```
import java.awt.Frame;
import java.awt.Graphics;
import java.awt.Color;
import java.awt.Event;
import java.awt.AWTEvent;
import java.awt.event.WindowListener;
import java.awt.event.WindowEvent;

/**
 * This class is a window for a ShapeList.
 * The Shapes are painted in a Frame.
 */
public class DrawWindow extends Frame
    implements WindowListener {

    private ShapeList list; // list of Shape elements
```

```
/**
 * Creates a new Frame for the Winter Scene.
 * @param width Width of the Frame. width >= 0.
 * @param height Height of the Frame. width >= 0.
 */
public DrawWindow(int width, int height,
                  ShapeList elements) {
    // Frame parameters
    setSize(width, height);
    setTitle("Winter Scenes");

    // this class handles the window events
    addWindowListener(this);

    // the ShapeList to be drawn
    list = elements;
    setVisible(true);
}
```

### *Ein größeres Beispiel: Darstellung einer Winterlandschaft*

- Fortsetzung des Codes von **DrawWindow**

```
/**
 * Paints the given list of Shapes.
 * @param g --
 */
public void paint(Graphics g) {
    // trivial case
    if (list==null)
        return;

    // go through the non-empty list
    ShapeNode x = list.getHead();
    while(x != null) {
        x.getData().draw(g);    // draw the Shape
        x=x.getNext();          // switch to the next
    }
    return;
}
```

```
/**
 * Closes the window if desired and exits
 * the whole program.
 * @param e --
 */
public void windowClosing(WindowEvent e) {
    setVisible(false);

    dispose();
    System.exit(0);
}

/** Dummy methods for implementing WindowListener */
public void windowActivated(WindowEvent e) { }
public void windowClosed(WindowEvent e) { }
public void windowDeiconified(WindowEvent e) { }
public void windowIconified(WindowEvent e) { }
public void windowOpened(WindowEvent e) { }
public void windowDeactivated(WindowEvent e) { }
}
```

### *Ein größeres Beispiel: Darstellung einer Winterlandschaft*

- Implementierungsalternative
  - Statt einer speziellen Liste vom Typ **ShapeList** hätten wir auch eine **generische Liste** zum Speichern von Shape-Objekten verwenden können
  - Wenn wir in der Klasse **DrawWindow** eine solche Liste vom Typ **List** statt vom Typ **ShapeList** verwenden, müssen wir in der Methode **paint** im Programmtext
    - Erst einmal **überprüfen**, ob das in einem Listen-Knoten gespeicherte Objekt vom Typ **Shape** ist
    - Es dann **explizit** zu einem **Shape**-Objekt **umwandeln**
    - Und dann wie zuvor die virtuelle Methode **draw** des Shape-Objekts aufrufen
  - Wie zuvor wird daraufhin **dynamisch** zur Laufzeit die **draw**-Methode des aktuellen Typs des Objekts benutzt (also die **draw**-Methode von **Snowman**, **SnowFlake**, **Circle**, **Rectangle** usw.).

### *Ein größeres Beispiel: Darstellung einer Winterlandschaft*

- Implementierungsalternative: Code

```
public class DrawWindow extends Frame
    implements WindowListener {
    private List list; // generic list of Shape elements
    // ...
    /**
     * Paints the given (generic) list of Shapes.
     */
    public void paint(Graphics g) {
        // trivial case
        if (list==null)
            return;

        // go through the non-empty list
        Node x = list.getHead();
        while(x != null) {
            Object tmpo = x.getData();
            // if the Node contains a Shape Object
            // then draw it
            if (tmpo instanceof Shape) {
                Shape tmps = (Shape) tmpo; // explicit type cast
                tmps.draw(g); // use dynamic binding for drawing
            }
            x=x.getNext(); // switch to the next
        }
        return;
    }
}
```

*Ein größeres Beispiel: Darstellung einer Winterlandschaft*

- Hauptprogramm **SnowScene**
  - Die Objekte einer ganzen Winterlandschaft können z. B. mit der main-Methode der folgenden Klasse **SnowScene** gezeichnet werden
    - Neben zwei Schneemännern werden 120 Schneeflocken gezeichnet

### *Ein größeres Beispiel: Darstellung einer Winterlandschaft*

```
import java.awt.Point;
import java.awt.Color;

/**
 * This class adds several Shape objects to a ShapeList
 * and calls DrawWindow to paint them.
 */
public class SnowScene {
    DrawWindow win;    // the window to paint in
    ShapeList list;    // Shapes to be painted

    /**
     * Creates a SnowScene with two snowmen and
     * some flakes and gives a ShapeList to DrawWindow.
     */
    public SnowScene() {
        list = new ShapeList(); // create empty list

        // Adding two Snowman objects to the list
        Snowman sm1 = new Snowman(new Point(200, 475), 250);
        list.insertFirst(sm1);

        Snowman sm2 = new Snowman(new Point(400, 400), 200);
        sm2.body.setBodyColor(Color.blue);
        list.insertFirst(sm2);

        // Adding Snowflake objects to the list
        for(int i = 0; i < 120; i++) {
            list.insertFirst(
                new Snowflake((int)(600*Math.random()),
                              (int)(600*Math.random())));
        }

        // Create a DrawWindow to paint the objects in list
        win = new DrawWindow(600, 600, list);
    }

    /**
     * Test of SnowScene
     */
    public static void main(String args[]) {
        SnowScene sc = new SnowScene();
    }
}
```