

# Objektorientiert Programmieren I
# Klassen

Stephan Neuhaus

Lehrstuhl Softwaretechnik
Universität des Saarlandes, Saarbrücken









# *Überblick*

Drei Vorlesungen „Objektorientiertes Programmieren“

Für Leute, die Nachholbedarf haben oder auffrischen wollen

Beispiele hauptsächlich in C++, auch in Java oder (selten) ML

Kenntnisse in prozeduralem Programmieren werden *vorausgesetzt*!

1. Klassen und Objekte
2. Vererbung
3. Spezialitäten









# *Objekte*

Objekte sind Dinge wie „dieses Fenster“, „jene Person“ usw.

Wir werden jetzt nicht auf die philosophischen Fragestellungen nach der Wirklichkeit von Objekten eingehen, sondern einfach annehmen, daß es Objekte tatsächlich gibt

Wer sich für so etwas interessiert, dem sei Platos „Höhlengleichnis“, Kants „Kritik der reinen Vernunft“ und Schopenhauers Anmerkungen dazu empfohlen

Aus programmiertechnischer Sicht sind zwei Fragen interessant:

- Welche Daten hat ein Objekt?
- Welche Operationen kann ich mit dem Objekt ausführen?



# *Beispiele für Objekte*

Dieses Fenster hat einen Griff, zwei Scharniere, eine Doppelglasscheibe und einen Rahmen (und noch viel mehr Dinge, wie eine Vielzahl von Atomen) (Attribute)

Dieses Fenster kann man öffnen und schließen (Methoden)

(Ein kaputtes Fenster kann man möglicherweise nicht immer schließen!)

Die hier anwesenden Studierenden sind (aus diesem Blickwinkel betrachtet!) allesamt Objekte



# *Klassen*

Eine Klasse ist die Zusammenfassung von Objekten unter einem Namen

Bei der Modellierung einer Klasse wird man sich auf solche Merkmale beschränken, die bei Objekten dieser Klasse relevant verschieden sind (also z.B. die Scharniere, nicht aber z.B. die Farbe oder die genaue molekulare Konfiguration)

Eine Klasse ist daher eine *Abstraktion*

Alle Fenster haben Scharniere, Scheiben, Rahmen usw. (gemeinsame Attribute): dieses Attribut ist bei verschiedenen Fenstern unterschiedlich

Alle Fenster kann man öffnen und schließen (Methoden)



# *Klassen*

Manchmal sind bestimmte Methoden (z.B. „Fenster schließen“) bei bestimmten Objektzuständen (z.B. „Fenster kaputt“) nicht möglich → *Ausnahmebehandlung*

Andere Dinge, die man im wirklichen Leben mit Fenstern machen kann, werden vielleicht nicht modelliert (z.B. „einschmeißen“) (Ein Modell, das so genau ist, daß es wirklich alle Aspekte des Originals enthält, ist mindestens so kompliziert wie dieses.)

Ein einfaches Loch in der Wand ist kein Fenster, weil ihm Scharniere und die Glasscheibe fehlen

Nicht alle Phänomene eignen sich für eine solche Modellierung (z.B. „Sonnenaufgang“)



# *Woher kommen Klassen?*

Code und Daten sind nicht getrennt, sondern konzeptionell oft zusammen.

Beispiel aus dem X Window System: Zum struct `Display` gehören Funktionen wie `XOpenDisplay()` und `XCloseDisplay()`, aber nicht `XOpenWindow()`, `XCreateGC()` oder `XConfigureEvent()`

X Windows kommt aus einer Zeit, als C++ noch nicht weit verbreitet war. Heute würde man `Display` als Klasse und `XOpenDisplay()` als eine Methode dieser Klasse modellieren

Nicht alle Probleme eignen sich für eine solche Modellierung, z.B. das $n$-Damen-Problem



# *Klassen in C++ und Java*

Klassen sind aufgeteilt in eine *Deklaration* (Interface) und eine *Definition* (Implementierung)

Die Deklaration bestimmt, welche Methoden und Attribute einem Benutzer der Klasse zur Verfügung stehen

In der Definition wird gesagt, wie diese Methoden und Attribute genau implementiert sind

C++: Beide getrennt und in der Regel auch auf mehrere Dateien aufgeteilt. Deklaration: Header-File (Endung `.h`), Definition: Source-File (Endung `.cc`, `.C`, `.cxx` o.ä.)

Java: Deklaration und Definition in einer Datei (Endung `.java`)

In C++ und Java heißen Attribute „member“



# Attribut- und Methodenzugriff in C++

```
#include <iostream>

class MyWindow {            // Declare class: MyWindow
public:
  int handle;               // Declare member: MyWindow::handle
  void open();              // Declare method: MyWindow::open
  void close();             // Declare method: MyWindow::close
};

void MyWindow::open() { handle = 1; }  // Define MyWindow::open()
void MyWindow::close() { handle = 0; } // Define MyWindow::close()

int main(int argc, const char *argv[]) {
  MyWindow window;          // Create MyWindow object "window"

  window.open();            // Call method MyWindow::open on "window"
  std::cout << window.handle << std::endl;
  window.close();           // Call method MyWindow::close on "window"
  std::cout << window.handle << std::endl;
  return 0;                 // 0 = success
}
```









# Attribut- und Methodenzugriff in Java

```
// Must be in file called MyWindow1.java
public class MyWindow1 {  // Declare and define class: MyWindow1
  public int handle;

  public void open() { handle = 1; }  // Define MyWindow1.open()
  public void close() { handle = 0; } // Define MyWindow1.close()

  public static void main(String[] args) {
    // Create MyWindow1 object "window"
    MyWindow1 window = new MyWindow1();
    window.open();       // Call method MyWindow1.open on "window"
    System.out.println(window.handle);
    window.close();      // Call method MyWindow1.close on "window"
    System.out.println(window.handle);
  }
}
```









# C++: Zugriff über Pointer

```
#include <iostream>

class MyWindow { public: int handle; void open(); void close(); };
void MyWindow::open() { handle = 1; }
void MyWindow::close() { handle = 0; }

int main(int argc, const char *argv[]) {
  // Construct anonymous MyWindow object dynamically. The
  // "new" operator is analogous to malloc(), only it's type safe
  MyWindow *window = new MyWindow;

  if (window == 0)          // When "new" encounters an error,
    return 1;               // we don't proceed (1 = failure)

  window->open();           // Call method MyWindow::open on "*window"
  std::cout << window->handle << std::endl;
  (*window).close();        // Alternative to window->close()
  std::cout << window->handle << std::endl;
  delete window;            // Delete is analogous to free()
  return 0;
}
```







# *Sichtbarkeit*

Manche Methoden oder Attribute einer Klasse sollen von außen sichtbar sein, manche nicht.

Angenommen z.B., das Attribut `handle` dürfte nur die Werte $0$ (für geschlossen), $1$ (für offen) und $2$ (für Kippstellung) annehmen.

„Vertrauen ist gut, Kontrolle ist besser“ — W. I. Lenin (?)

Schutz des Attributs vor direktem Zugriff, Attributsänderung nur mit speziellen Methoden









# Objektschutz

```
#include <iostream>

class MyWindow {
public:          // Accessible from outside
  void open();
  void close();
  void tilt();
  int getHandle();

private:         // Not accessible from outside
  int handle;    // This is line 11
};

void MyWindow::open() { handle = 1; }
void MyWindow::close() { handle = 0; }
void MyWindow::tilt() { handle = 2; }
int MyWindow::getHandle() { return handle; }
```









# *Objektschutz: Benutzung*

```
int main(int argc, const char *argv[]) {
  MyWindow window;

  window.handle = 1; // This is line 22
  return 0;
}
```

Benutzung von `private` Attributen führt zum Compilerfehler:

```
% g++ -o Win2 Win2.cc
Win2.cc: In function ‘int main(int, const char **)’:
Win2.cc:11: ‘int MyWindow::handle’ is private
Win2.cc:22: within this context
```









# *Objektschutz:* `const`*-Methoden*

Eine `const` Methode ändert den Zustand des Objekts nicht (reine Auskunftsmethode)

```
class MyWindow {
public:
  MyWindow() { x = 0; }; // Explanation later
  int getX() const { return x; };
  void setX(int newX) { x = newX; };
private:
  int x;
};
int main(int argc, const char *argv[]) {
  MyWindow w1;
  const MyWindow w2;

  w1.getX(); w2.getX();
  w1.setX(3); // OK
  w2.setX(3); // Error: non-const member function on const object
  return 0;
}
```









# *Fehlermeldung zu* const

```
% g++ -c Const1.cc
Const1.cc: In function ‘int main(int, const char **)’:
Const1.cc:16: passing ‘const MyWindow’ as ‘this’ argument of
    ‘void MyWindow::setX(int)’ discards qualifiers
```

Bedeutet: Würde man die Methode `setX()` auf `w2` anwenden (das ist das „‘`this`’ argument“), würde dadurch die `const`-Qualifizierung des Objekts `w2` verloren gehen, weil die anzuwendende Methode nicht `const` ist

Interpretation von Fehlermeldungen manchmal mit viel Fantasie!









# this

Die Spezialvariable `this` verweist in C++ und Java jeweils auf „dieses“ Objekt (andere Sprachen haben da z.B. `self`)

In C++ hat `this` einer Methode der Klasse X den Typ `X *`, in einer `const`-Methode den Typ `const X *`

In Java hat `this` einer Klasse X den Typ X

```
void MyWindow::setX(int x) {
  this->x = x;  // Help disambiguate member and parameter
}

MyWindow &MyWindow::incX() {
  x++;
  return *this; // Help write things like w.incX().incX()
}
```









# *Polymorphismus, Overloading*

Unter *Polymorphismus* versteht man hier: Die Möglichkeit zwei Methoden denselben Namen zu geben

Klingt zunächst bescheuert. Warum macht man sowas trotzdem?

Analog aus der Mathematik: Was bedeutet das Pluszeichen in $a + b$? Z.B. $+ : \mathrm{R} \longrightarrow \mathrm{R}$. Oder $+ : \mathbf{N} \longrightarrow \mathbf{N}$. Kommt also drauf an! Trotzdem ist es sinnvoll, dasselbe Zeichen für verschiedene Funktionen zu benutzen.

Auch bereits in Programmiersprachen: a + b wird abhängig von den Datentypen von a und b übersetzt

Der Compiler findet anhand der unterschiedlichen Signatur heraus, welche Methode aufgerufen werden soll









# *Signatur*

Zur *Signatur* einer Methode gehören (u.a.)

- Name der Funktion
- Typ und Reihenfolge der formalen Parameter
- Sichtbarkeit der Methode
- Typ des Rückgabewerts
- `const` und andere Qualifizierer
- Ein paar Sachen, die wir noch nicht hatten. . .

Die genauen Regeln, welche Funktion genau aufgerufen wird, wenn mehrere gleichen Namens zur Wahl stehen, sind kompliziert und werden vom Compiler bestimmt besser beherrscht als vom Programmierer









# Overloading: Was geht und was nicht

```
void print(double x) {}
void print(long l) {}

void f() {
  print(1L);  // print(long)
  print(1.0); // print(double)
  print(1);   // error: ambiguous
}

void g() {}
int g() { return 1; } // Error: Ambiguous definition

void h(char c) {}
void h(int i) {}

void k() {
  char c = 0;
  h(c);         // h(char)
  h(0);         // h(int)
  h((char) 0); // h(char)
}
```









# *Referenzen*

Referenzen sind alternative Namen für Objekte. Ihr Hauptzweck liegt in der Spezifikation von Argumenten und Rückgabewerten von Funktionen. Referenzen müssen bei der Definition (nicht notwendig bei der Deklaration) initialisiert werden.

```
int i = 3;
int &ref1 = i; // Ref1 and i refer to the same int
int &ref2;     // Error: must be initialized on definition

ref1++;        // Both ref1 and i now have the value 4
```

Nach der Initialisierung kann die Referenz nicht mehr geändert werden (nur noch der Referent). Von einer Referenz kann die Adresse nicht bestimmt werden.



# *Referenzen als Funktionsparameter*

C++ unterstützt call by reference

```
void increment(int &i) { i++; }

void f() {
  int i = 1;

  increment(i);
  // i now has the value 2
}
```

Man kann sich Referenzen als besondere (konstante) Zeiger vorstellen, die automatisch bei jeder Benutzung dereferenziert werden









# *Lebenszyklus eines Objekts*

1. Objekt wird *erzeugt* (die Klasse wird *instanziiert*): Bereitgestellter Speicher wird initialisiert und das Objekt wird in einen Initialzustand versetzt (man sagt „Objekt $x$ ist Instanz der Klasse $X$“)
2. Objekt wird *verwendet*: Durch Aufruf von Methoden wird der Attribut des Objekts bzw. seiner Unterobjekte, bzw. der Aufrufumgebung verändert
3. Objekt wird *zerstört*: Der Speicher, der vom Objekt verwendet wird, wird freigegeben und das Objekt hört auf zu existieren









# *Lebenszyklus-Management*

Besondere Methoden, die bei besonderen Punkten während der Lebenszeit eines Objekts aufgerufen werden.

*Konstruktor*: Attribute initialisieren initialisieren, so daß das Objekt nach Ausführung des Konstruktors vollständig initialisiert ist.

*Destruktor*: Aufruf, kurz bevor das Objekt zu Leben aufhört. Freigabe von Ressourcen, die in unmittelbarer Kontrolle des Objekts sind.

Java später









# Vollständiger Lebenszyklus: Deklaration

```
#include <iostream>

class MyWindow {
public:
  MyWindow();                 // Default constructor
  MyWindow(const MyWindow&); // Copy Constructor (overloaded)
  MyWindow(int x);            // Ordinary constructor (overloaded)
  ~MyWindow();                // Destructor

  void setX(int newX);
  int getX();

private:
  int x;
};
```









# *Vollständiger Lebenszyklus: Definition*

Das ist *kein* Beispiel für gute Formatierung!

```
MyWindow::MyWindow() { x = 0;
  cout << "Default constructor called\n";
}
MyWindow::MyWindow(const MyWindow& a) { x = a.x;
  cout << "Copy constructor called with x = " << x << endl;
}
MyWindow::MyWindow(int x) { this->x = x;
  cout << "Regular constructor called with x = " << x << endl;
}
MyWindow::~MyWindow() {
  cout << "x = " << x << " destroyed!\n";
}
void MyWindow::setX(int newX) { x = newX;
  cout << "State change (setter) with x = " << x << endl;
}
int MyWindow::getX() { return x; }
```



# Vollständiger Lebenszyklus: Verwendung

```
int main(int argc, const char *argv[]) {
  MyWindow a1;
  a1.setX(2);

  MyWindow a2 = a1;
  MyWindow a3 = 1;

  cout << a3.getX() << endl;

  MyWindow *a4 = new MyWindow;

  return 0;
}
```









# *Vollständiger Lebenszyklus: Ausführung*

```
% g++ -o Lifecycle1 Lifecycle1.cc
% ./Lifecycle1
Default constructor called
State change (setter) with x = 2
Copy constructor called with x = 2
Regular constructor called with x = 1
1
Default constructor called
x = 1 destroyed!
x = 2 destroyed!
x = 2 destroyed!
%
```

Nanu, da fehlt doch ein Destruktor?









# Beispiel in Java

```
public class TestMe {
  private int x;

  private TestMe(int x) { this.x = x; }
  private int getX() { return x; }
  private void setX() { this.x = x; }

  public static void main(String[] args) {
    // Create object "a" of type "TestMe"
    TestMe a = new TestMe(1);

    // Change state of object "a"
    a.setX(2);

    // Change state of object "System.out" by outputting getX()
    System.out.println(a.getX());

    // Implicitly delete object "a" by garbage collection
  }
}
```









# Namespaces

Namespaces dienen dazu, gleiche Namen (z.B. `Window`) auseinanderzuhalten

```
namespace a {
  void foo ()
  {
  }
}

namespace b {
  void foo ()
  {
  }
}

void foo()
{
  a::foo();
  b::foo();
}
```









# Namespace-Vereinfachung

```
namespace a {
  void foo ()
  {
  }
}

namespace b {
  void foo ()
  {
  }
}

// When I say "foo", I mean "a::foo"
using a::foo;

void bar()
{
  foo();    // means a::foo();
  b::foo(); // disambiguate
}
```









# Klassen in C++: Deklaration mit Namespace

```
#include <X11/X.h>
#include <X11/Xlib.h>
#include <string>

namespace MyProject {
  class Window {
  public:
    // ... stuff deleted

  private:
    // :: means: use global "Window" defined in <X11/Xlib.h>
    ::Window window;
    Display *display;
  };
}
```







# Klassendefinition in C++

```
#include "Window.h"

// +----------------------- Namespace
// |          +------------ Class
// |          |       +---- Method (Constructor)
// |          |       |
// v          v       v
MyProject::Window::Window()
{
  window = 0;  // Assignment to member
  display = 0; // Assignment to member
}

// Namespace    Therefore, no namespace
// opens here   necessary here ---+
// |                              |
// v                              v
MyProject::Window::Window(const Window &w)
{
  window = w.window; display = w.display;
}
```



# Guter Stil I

- Entweder alles englisch oder alles deutsch. Gut: `getNextTarget()`. Auch gut: `holeNaechstesZiel()`. Schlecht: `getNaechstesZiel()`.
- Entweder `member_function()` oder `memberFunction()`
- Klassennamen ohne Unterstriche mit Groß/Kleinschreibung
- Egal welcher Einrückstil, aber bloß einer und konsistent
- Sparsam aber gezielt kommentieren (am besten elektronisch weiterverarbeitbar, siehe Javadoc)
- Keine Abkürzungen: Nicht `mxBfSz`, sondern `maxBufferSize` (Ausnahmen bei Standardabkürzungen wie „max“ o.ä.)
- Namespaces verwenden

# *Guter Stil II*

- In der Kürze liegt die Würze: Also `maxBufferSize` statt `bufferSizeForTheSmallBufferThatWasDeclaredAboveForThePur`
- Kommentare und Code tendieren dazu, auseinanderzulaufen. Daher: Kommentare knapp halten und dabei ausschließlich auf nicht offensichtliche Effekte oder externe Dokumentation (z.B. Requirements) hinweisen. „Only source code never lies“
- Manche *de-facto*-Konventionen entsprechen nicht den eigenen Vorlieben.
- Nicht dagegen wehren
- Die meisten interessanten Programmieraufgaben sind Teamaufgaben, also soziale Tätigkeiten, bei denen es darauf ankommt, verstanden zu werden.