



FH MÜNSTER  
University of Applied Sciences

# Programmieren in C++

## Teil 2 – Objektorientierung 1 | Objekte und Klassen

Prof. Dr. Kathrin Ungru  
Fachbereich Elektrotechnik und Informatik

[kathrin.ungru@fh-muenster.de](mailto:kathrin.ungru@fh-muenster.de)

# Objektorientierung 1

## Inhalt



FH MÜNSTER  
University of Applied Sciences

- Einführung in die Objektorientierung
  - Was ist eine Klasse?
  - Was ist ein Objekt?
- Einfache Klassen in C++ definieren
- Objekte in C++ erzeugen
- Begriffe: Deklaration, Definition, Initialisierung



# Objektorientierung 1

## Einführung in die Objektorientierung

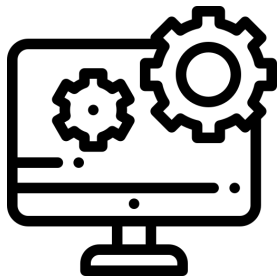


FH MÜNSTER  
University of Applied Sciences

# Einführung in die Objektorientierung

## Anforderungen an Software

### Software...



#### ... in der Anwendung:

- Software muss korrekt sein
- Software muss benutzerfreundlich sein
- Software muss effizient und performant sein

#### ... als Kostenfaktor:

- Software muss effizient mit Ressourcen umgehen
- Software soll günstig sein
- Geringer Schulungsaufwand

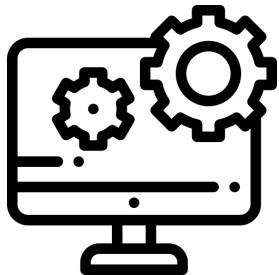
#### ... in der Entwicklung:

- Software muss sich schnell anpassen lassen
- Software soll in möglichst kurzer Zeit fertiggestellt werden
- Software soll über Jahre genutzt werden ... und verändert werden können

# Einführung in die Objektorientierung

## Anforderungen an Software: Zusammengefasst

### Software ...



... soll genau das tun, was von ihr erwartet wird.

➤ **Korrektheit**

... soll einfach und intuitiv zu benutzen sein.

➤ **Benutzerfreundlichkeit**

... soll mit wenigen Ressourcen auskommen und performant sein.

➤ **Effizienz**

... soll mit wenig Aufwand erweiterbar und änderbar sein.

➤ **Wartbarkeit**

**Problem: Komplexität von Software**

**Lösung: Objektorientierte Programmierung**

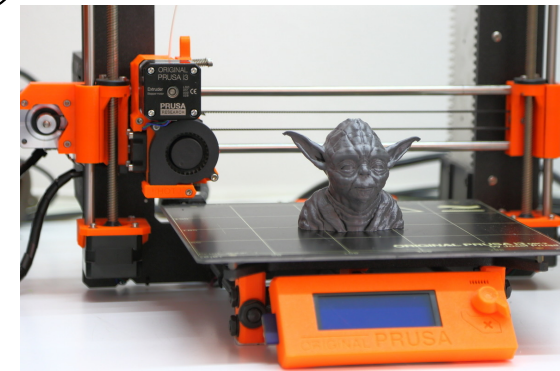
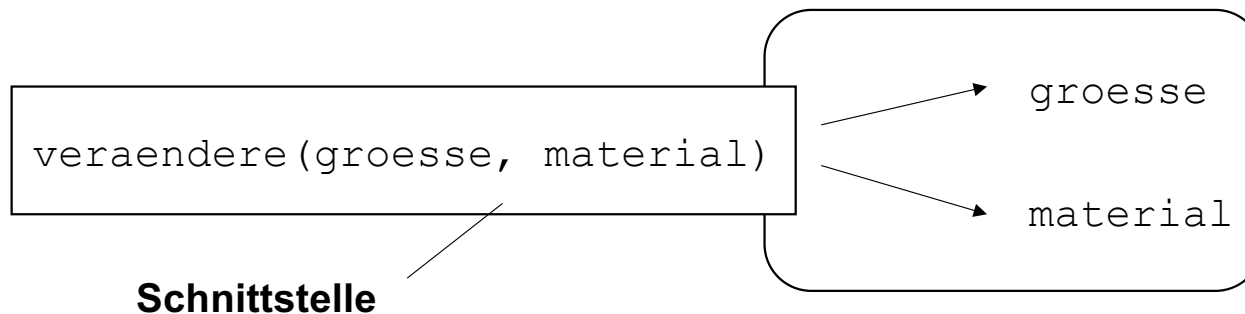
# Einführung in die Objektorientierung

## Warum Objektorientierung?

- Objektorientierung bietet einen Werkzeugkasten, der es erlaubt die Anforderungen an Software in der Entwicklung zu berücksichtigen. Basiswerkzeuge:
  - **Datenkapselung**
  - **Vererbung**
  - **Polymorphie**

# Einführung in die Objektorientierung

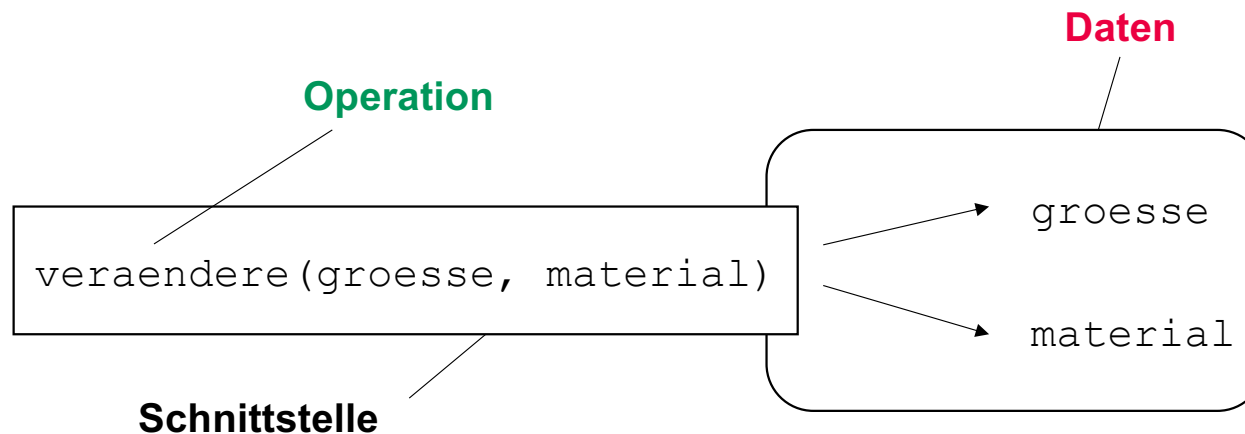
## Prinzip der Datenkapselung



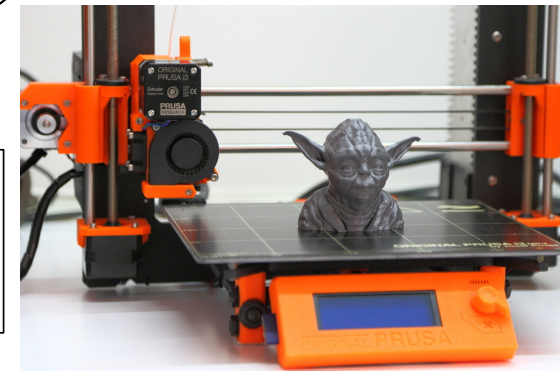
<https://www.musenkuss-muenchen.de/angebote/erwachsenenkurs-3d-kick-off>, September 2020

# Einführung in die Objektorientierung

## Prinzip der Datenkapselung



**Daten** repräsentieren den internen **Zustand** eines Objektes.  
**Daten** eines Objektes können nur über die **Operationen** des Objektes von anderen Objekten verändert werden.



<https://www.musenkuss-muenchen.de/angebote/erwachsenenkurs-3d-kick-off>, September 2020

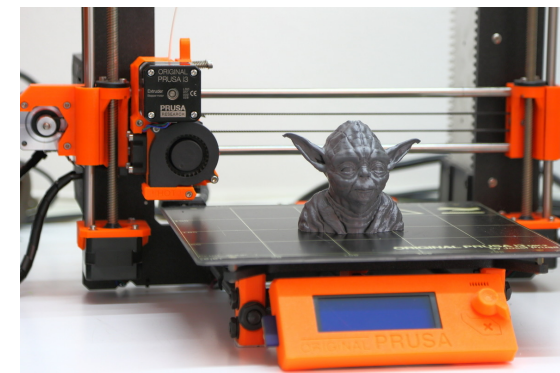


# Einführung in die Objektorientierung

## Was ist ein Objekt?

- Ein **Objekt** kapselt **Daten** und legt diese als Werte im Speicher ab (**Zustand**).
- Der Zustand eines **Objektes** kann sich durch Aktivitäten des **Objektes** ändern, d.h. durch **Operationen**, die die Daten und damit den Zustand des **Objektes** verändern.
- Ein **Objekt** hat **Eigenschaften (Attribute)**, die von außen erfragt werden können. **Eigenschaften** können direkt auf Daten abgebildet werden oder aus Daten berechnet werden.
- Ein **Objekt** wird nach einem bestimmten Bauplan erzeugt.

➤ Dieser Bauplan nennt sich **Klasse**.

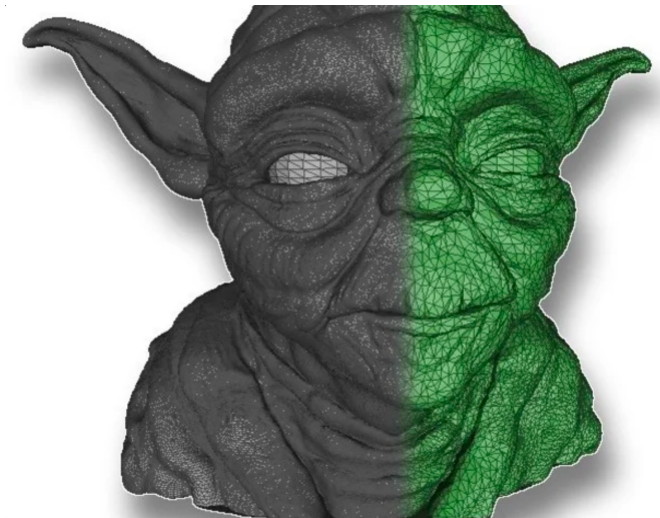


<https://www.musenkuss-muenchen.de/angebote/erwachsenenkurs-3d-kick-off>, September 2020

# Einführung in die Objektorientierung

## Was ist eine Klasse?

- Eine **Klasse** enthält den **Bauplan** eines **Objektes**, d.h. in ihr sind alle Eigenschaften und Operationen eines Objektes beschrieben.
- Eine **Klasse** ist ein **abstrakter Datentyp** in einer Programmiersprache.
- In einer Programmiersprache dient eine **Klasse** dazu, dem Compiler die Beschreibung von später zu definierenden **Objekten** mitzuteilen.
- Beschreibt eine **Klasse** auch die Implementierung einer **Operation**, so nennt man diese Implementierung **Methode**.
- Eine **Klasse** belegt keinen Speicher.

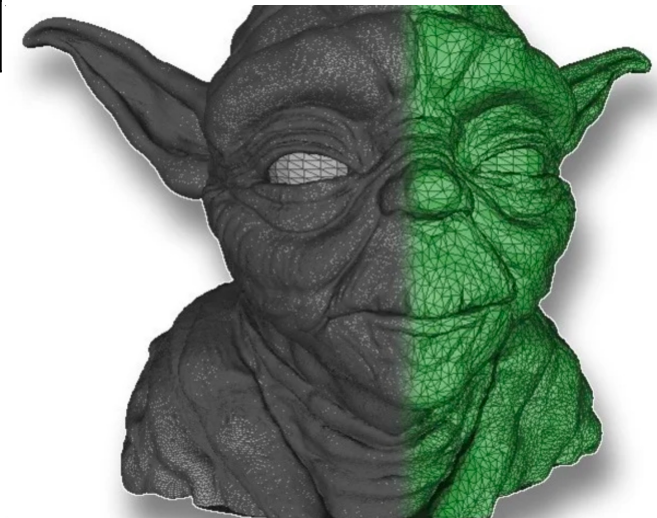


<https://www.thingiverse.com/thing:10752>, November 2020

# Einführung in die Objektorientierung

## Die Klasse: Bauplan eines Objektes

<b>Name der Klasse</b>	<b>Yoda</b>
<b>Eigenschaften</b>	<ul style="list-style-type: none"> <li>- groesse : double</li> <li>- material : MaterialTyp</li> </ul>
<b>Operationen</b>	+ veraendere(groesse, material)

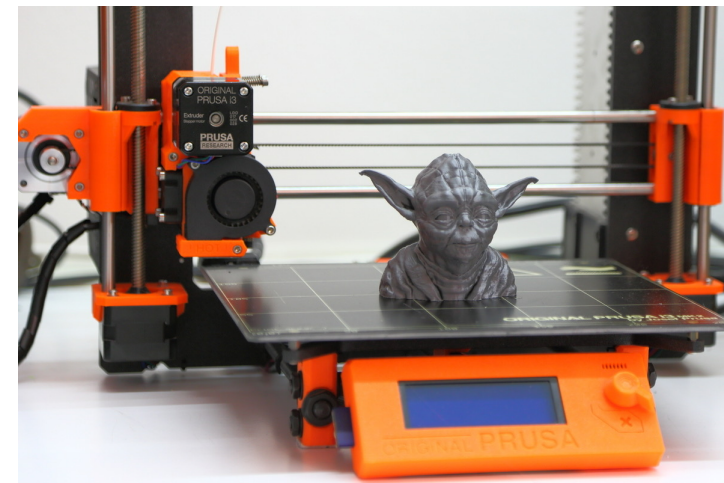
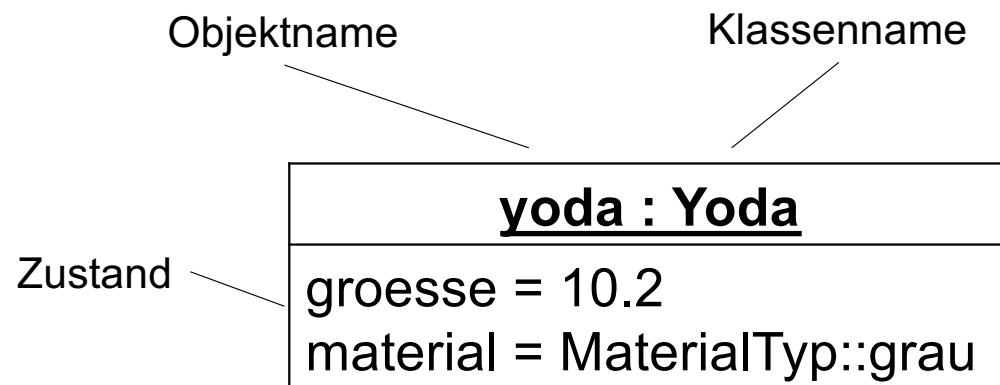


<https://www.thingiverse.com/thing:10752>, November 2020

# Einführung in die Objektorientierung

## Objekterzeugung

- **Objekte** werden nach dem **Bauplan** einer **Klasse** erzeugt
- Die Daten eines **Objektes** haben Werte, die den **Zustand** beschreiben
- Dies ist ein **Objekt** der **Klasse Yoda**:

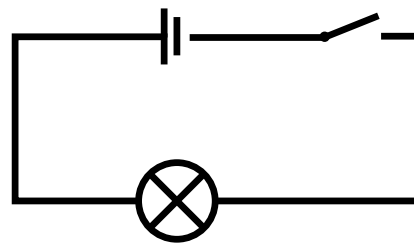


<https://www.musenkuss-muenchen.de/angebote/erwachsenenkurs-3d-kick-off>, September 2020

# Einführung in die Objektorientierung

## Eigenschaften vs. Daten

- Beispiel Stromkreis:



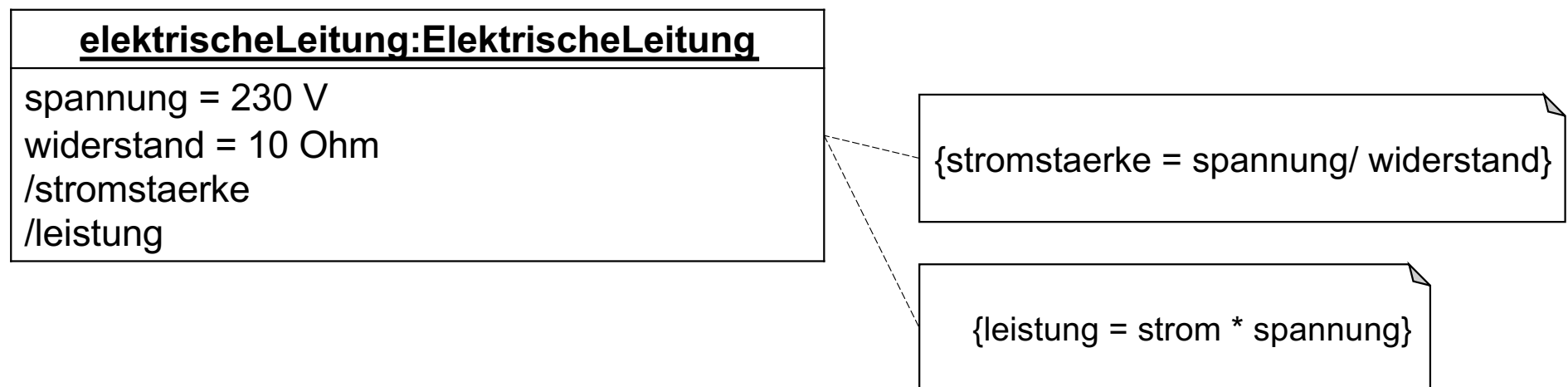
$$I = U / R$$

- Der Stromkreis hat die **Eigenschaften**: Spannung (U), Widerstand (R) und Stromstärke (I)
- Widerstand und Spannung liegen als **Daten** im Objekt vor.
- Die Stromstärke berechnet sich aus Widerstand und Spannung.

# Einführung in die Objektorientierung

## Objekte und Klassen in UML (Unified Modeling Language)

- Darstellung der inneren Struktur eines **Objektes** in UML
- / bezeichnet Eigenschaften, die von Daten abgeleitet werden



# Einführung in die Objektorientierung

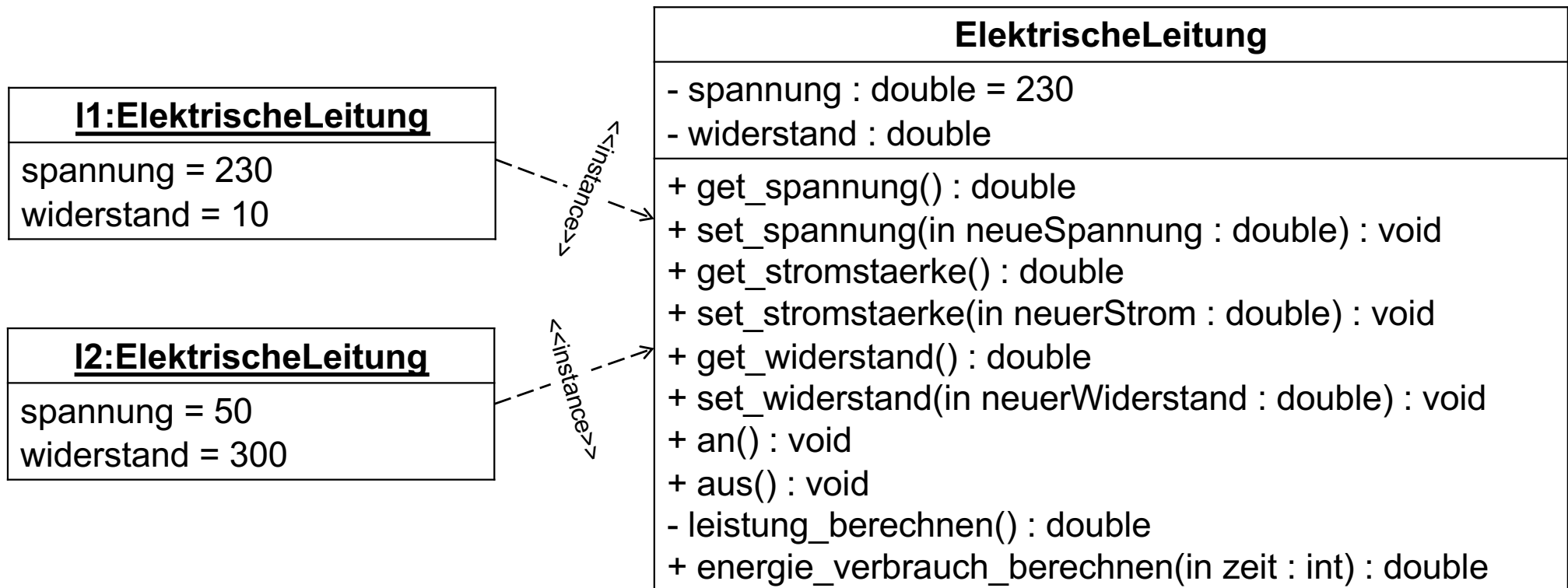
## Objekte und Klassen in UML (Unified Modeling Language)

- Darstellung einer **Klasse** in UML.
- Der **nicht-öffentliche Teil** (engl. **private**) einer Klasse wird in UML mit "-" gekennzeichnet.
- Der **öffentliche Teil** (engl. **public**) einer Klasse wird in UML mit "+" gekennzeichnet.

ElektrischeLeitung
- spannung : double = 230 - widerstand : double
+ get_spannung() : double + set_spannung(in neueSpannung : double) : void + get_stromstaerke() : double + set_stromstaerke(in neuerStrom : double) : void + get_widerstand() : double + set_widerstand(in neuerWiderstand : double) : void + an() : void + aus() : void - leistung_berechnen() : double + energie_verbrauch_berechnen(in zeit : int) : double

# Einführung in die Objektorientierung

## Objekte und Klassen in UML (Unified Modeling Language)





# Objekte und Klassen in C++

## Klassen definieren

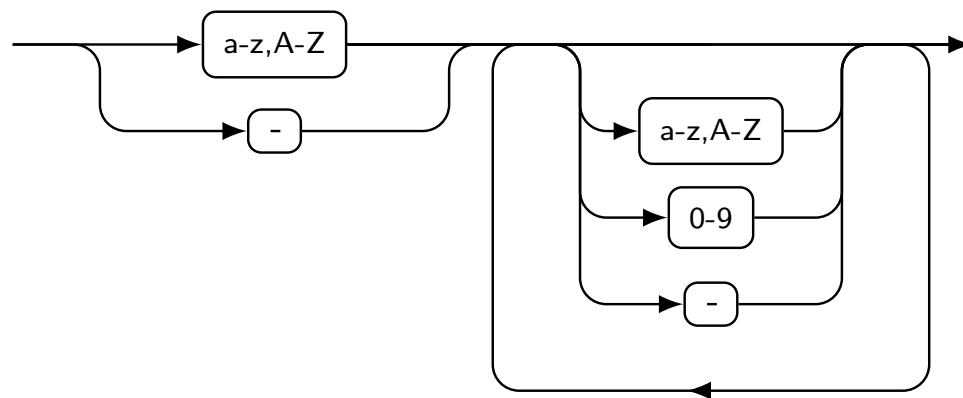


FH MÜNSTER  
University of Applied Sciences

# Namensregeln und -konventionen

## Namensregeln in C++

- Groß- und Kleinschreibung wird unterschieden: *Beispiel: Lisa ist etwas anderes als LISA*
- Syntaxgraph:



Übung: Welche Variablennamen sind richtig?

```

int _;
int _variable;
int 4variable;
int variable;
int v123;
int var iable;

```

- Es gibt mehr als 70 festgelegte Schlüsselwörter, die nicht als Namen verwendet werden dürfen. Schlüsselwörter wiederum bestehen nur aus Kleinbuchstaben.

# Namensregeln und -konventionen

## Schlüsselwörter in C++-20

<code>alignas</code>	<code>const</code>	<code>dynamic_cast</code>	<code>long</code>	<code>short</code>	<code>typedef</code>
<code>alignof</code>	<code>constexpr</code>	<code>else</code>	<code>mutable</code>	<code>signed</code>	<code>typeid</code>
<code>asm</code>	<code>constexpr</code>	<code>enum</code>	<code>namespace</code>	<code>sizeof</code>	<code>typename</code>
<code>auto</code>	<code>constinit</code>	<code>explicit</code>	<code>new</code>	<code>static</code>	<code>union</code>
<code>bool</code>	<code>const_cast</code>	<code>export</code>	<code>noexcept</code>	<code>static_assert</code>	<code>unsigned</code>
<code>break</code>	<code>continue</code>	<code>extern</code>	<code>nullptr</code>	<code>static_cast</code>	<code>using</code>
<code>case</code>	<code>co_await</code>	<code>false</code>	<code>operator</code>	<code>struct</code>	<code>virtual</code>
<code>catch</code>	<code>co_return</code>	<code>float</code>	<code>private</code>	<code>switch</code>	<code>void</code>
<code>char</code>	<code>co_yield</code>	<code>for</code>	<code>protected</code>	<code>template</code>	<code>volatile</code>
<code>char8_t</code>	<code>decltype</code>	<code>friend</code>	<code>public</code>	<code>this</code>	<code>wchar_t</code>
<code>char16_t</code>	<code>default</code>	<code>goto</code>	<code>register</code>	<code>thread_local</code>	<code>while</code>
<code>char32_t</code>	<code>delete</code>	<code>if</code>	<code>reinterpret_cast</code>	<code>throw</code>	
<code>class</code>	<code>do</code>	<code>inline</code>	<code>requires</code>	<code>true</code>	
<code>concept</code>	<code>double</code>	<code>int</code>	<code>return</code>	<code>try</code>	

Zusätzlich gibt es noch Alternativausdrücke für Operatoren wie `and`, `or`, `xor`, `compl` usw.

Diese Schlüsselwörter werden hier nicht aufgeführt, da wir im Folgenden die entsprechenden Operatoren nutzen werden, um Konfusion zu vermeiden.

# Namensregeln und -konventionen

## Namenskonventionen

- **Namenskonvention** bedeutet „Best Practice“ für die Namensgebung von Objekt- und Variablennamen, sowie Namen von Klassen und Methoden.
- Häufig auftretende Namenskonventionen:
  - Der erste Buchstabe von Variable, Objekten und Methoden wird klein geschrieben
  - Der erste Buchstabe einer Klasse wird groß geschrieben
  - **Pascal-Case:** `DiesIstEineKlasse`
  - **Camel-Case:** `diesIstEinObjekt`
  - **Snake-Case:** `dies_ist_eine_methode`
  - Bezeichner von Konstanten werden oft groß geschrieben und wenn nötig mit `_` getrennt  
`DIES_IST_EINE_KONSTANTE`

# Begriffe

## Definition, Deklaration, Initialisierung

### Rückblick

*Deklaration:* Vergabe von Namen und Typ



*Definition:* Reservierung von Speicher



*Initialisierung:* Definition und Zuweisung eines Wertes in einem Schritt.



*Beispiel:*



// Definition (und gleichzeitig Deklaration)

**int** x;

**int** x, y, z;

**double** dx, dy;

Definition und Deklaration



// Initialisierung bekannt aus C

**int** x = 3;

Initialisierung im C-Stil



// Typsichere Initialisierung mit {}

**int** x {3};

Initialisierung im C++-Stil (ab C++11)

# Eine einfache Klasse definieren

## Definition einer Klasse

- Eine Klasse (**class**) ist ein (Daten-)Typ der vom Benutzer selbst definiert werden kann
- Sie enthält Methoden (auch Member-Funktionen genannt) sowie Variablen (Member-Variablen) die dieser Klasse zugeordnet werden.
- Es können öffentliche (**public**) und nicht-öffentliche (**private**) Teile definiert werden.
- Werden diese Teile nicht angegeben, sind alle Variablen und Methoden standardmäßig **private**.

### Beispiel:

```
class Testklasse {  
    // öffentlicher Teil  
    public:  
        // Methode (Member-Funktion)  
        void ändern(int x, int y)  
        {  
            xKoordinate = x;  
            yKoordinate = y;  
        };  
    // privater Teil  
    private:  
        // Member-Variablen  
        // die nur innerhalb der Klasse "sichtbar" sind  
        int xKoordinate;  
        int yKoordinate;  
};
```

# Eine einfache Klasse definieren

## Definition einer Klasse

- Die **Implementierung** einer Methode kann **direkt** in der Klasse erfolgen, aber:
  - Dies wird sehr unübersichtlich bei langen Methoden!

*Beispiel:*

```
class Testklasse {  
    // öffentlicher Teil  
    public:  
        // Methode (Member-Funktion)  
        void ändern(int x, int y)  
        {  
            xKoordinate = x;  
            yKoordinate = y;  
        };  
    // privater Teil  
    private:  
        // Member-Variablen  
        // die nur innerhalb der Klasse "sichtbar" sind  
        int xKoordinate;  
        int yKoordinate;  
};
```


# Eine einfache Klasse definieren

## Definition einer Klasse


- Besser: saubere **Trennung** zwischen **Deklaration** und **Implementierung** der Klasse!

Beispiel:

```
class Testklasse {  
    // öffentlicher Teil  
    public:  
        // Deklaration der Methode  
        void ändern(int x, int y);  
    // privater Teil  
    private:  
        // Member-Variablen,  
        // die nur innerhalb der Klasse "sichtbar" sind  
        int xKoordinate;  
        int yKoordinate;  
};
```



```
// Definition der Methode  
void Testklasse::ändern(int x, int y)  
{  
    xKoordinate = x;  
    yKoordinate = y;  
}
```





# Zusammenfassung

## Deklaration und Definition in Klassen

```
class A; // Klassen-Deklaration

class A { // Klassen-Definition
public:
    A(); // Konstruktor-Deklaration
    ~A(); // Destruktor-Deklaration

    void foo(); // Deklaration der Methode foo
    void gib_werte_aus();
// privater Teil
private:
    // Member-Variable
    double dings;
};
```

# Objekte und Klassen in C++

## Objekte erzeugen



FH MÜNSTER  
University of Applied Sciences

# Objekte erzeugen

## Definition eines Objektes

- Ein **Objekt** einer Klasse kann wie eine Variable erzeugt werden.
- Die Klasse aus der das Objekt erzeugt wird ist dabei der **Objekttyp**
  - Analog: Datentyp einer Variablen
- Häufig wird ein Objekt einer Klasse auch **Instanz** oder (seltener) **Exemplar** einer Klasse genannt.
- Wird ein Objekt erzeugt, wird dies **Instanziieren** genannt.

### Beispiel:

```
int main()
{
    // Instanziiert das Objekt test
    Testklasse test;
    /* Das Zustand des Objektes kann mit Hilfe der
    oeffentlichen Methode aendern() veraendert
    werden */
    test.aendern(1, 2);

    return 0;
}
```

# Konstruktor und Destruktor

Objekte initialisieren und Speicher freigeben

## Konstruktor:

- regelt die Initialisierung des Objektes
- wird (automatisch) bei der Instanziierung eines Objektes aufgerufen

## Beispiel:

```
int main()
{
    // Instanziiert das Objekt test
    Testklasse test;
    /* Das Zustand des Objektes kann mit Hilfe der
    oeffentlichen Methode aendern() veraendert
    werden */
    test.aendern(1, 2);

    return 0;
}
```

# Konstruktor und Destruktor

Objekte initialisieren und Speicher freigeben

## Konstruktor:

- regelt die Initialisierung des Objektes
- wird (automatisch) bei der Instanziierung eines Objektes aufgerufen

## Destruktor:

- kontrolliert das aufräumen des Objektes und gibt gegebenenfalls reservierten Speicher wieder frei
- wird (automatisch) aufgerufen, wenn der Speicher des Objektes z.B. am Ende eines Anweisungs-Blocks wieder freigegeben wird

### Beispiel:

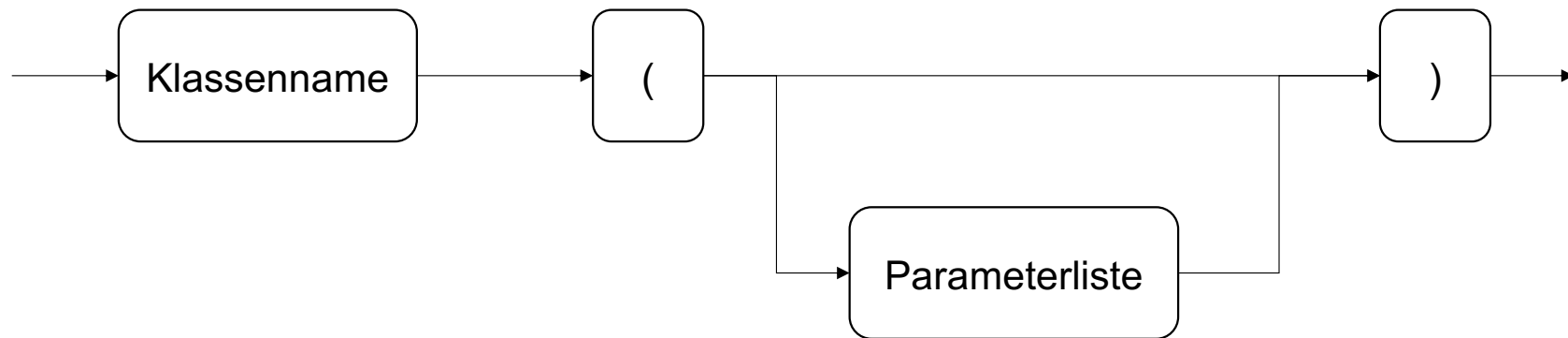
```
int main()
{
    // Instanziert das Objekt test
    Testklasse test;
    /* Das Zustand des Objektes kann mit Hilfe der
    oeffentlichen Methode aendern() veraendert
    werden */
    test.aendern(1, 2);

    return 0;
}
```

# Konstruktor und Destruktor

## Der Konstruktor

- Syntaxgraph



# Konstruktor und Destruktor

## Der Konstruktor

Beispiel:

```
class Testklasse {  
    // oeffentlicher Teil  
    public:  
        // Deklaration des ersten Konstruktors  
        Testklasse();  
        // Deklaration des zweiten Konstruktors  
        Testklasse(int x, int y);  
        // Deklaration der Methode  
        void ändern(int x, int y);  
        void gib_werte_aus();  
    // privater Teil  
    private:  
        // Member-Variablen  
        int xKoordinate;  
        int yKoordinate;  
};
```

- Eine Klasse kann **mehrere** Konstruktoren haben.
- Die verschiedenen Konstruktoren unterscheiden sich lediglich anhand der Signatur (Parameterliste).

```
// Definition des ersten Konstruktors  
Testklasse::Testklasse()  
{  
    xKoordinate = 0;  
    yKoordinate = 0;  
}  
// Definition des zweiten Konstruktors  
Testklasse::Testklasse(int x, int y)  
{  
    xKoordinate = x;  
    yKoordinate = y;  
}
```

**Zuweisung, keine Initialisierung!**

# Konstruktor und Destruktor

## Der Konstruktor

Beispiel:

```
class Testklasse {  
    // oeffentlicher Teil  
    public:  
        // Deklaration des ersten Konstruktors  
        Testklasse();  
        // Deklaration des zweiten Konstruktors  
        Testklasse(int x, int y);  
        // Deklaration der Methode  
        void ändern(int x, int y);  
        void gib_werte_aus();  
    // privater Teil  
    private:  
        // Member-Variablen  
        int xKoordinate;  
        int yKoordinate;  
};
```

- Eine Klasse kann **mehrere** Konstruktoren haben.
- Die verschiedenen Konstruktoren unterscheiden sich lediglich anhand der Signatur (Parameterliste).

```
// Definition des ersten Konstruktors  
Testklasse::Testklasse()  
{  
    xKoordinate = 0;  
    yKoordinate = 0;  
}  
// Definition des zweiten Konstruktors  
Testklasse::Testklasse(int x, int y):  
xKoordinate{x}, yKoordinate{y}  
{  
}
```

**Besser: Zuweisung über Initialisierungsliste**

Schneller, weil Definition und Zuweisung in einem Schritt, was einer echten Initialisierung entspricht!



# Konstruktor und Destruktor

## Konstruktor

*Beispiel:*

```
int main()
{
    // Instanziiert das Objekt test1
    // mit erstem Konstruktor
    Testklasse test1;
    // Gibt Werte der Member-Variablen aus
    test1.gib_werte_aus();
    /* Ausgabe:
       xKoordinate: 0, yKoordinate: 0
    */
    // Instanziiert das Objekt test2
    // mit zweitem Konstruktor
    Testklasse test2 {3, 5};
    // Gibt Werte der Member-Variablen aus
    test2.gib_werte_aus();
    /* Ausgabe:
       xKoordinate: 3, yKoordinate: 5
    */

    return 0;
}
```

Die Konstruktoren der Klasse **Testklasse** werden bei der Instanziierung der Objekte test1 und test2 aufgerufen

Initialisierung mit  
Initialisierungsliste in {}

# Exkurs: Typsichere Initialisierung

## ab C++ 11

- Durch die Initialisierung mit {} ist eine implizite Typumwandlung während der Initialisierung nicht möglich

*Beispiel:*

```
int x {}; // Initialisierung einer Variablen mit 0
int y {4}; // Initialisierung einer Variablen mit 4
Testklasse test {3, 5}; // Initialisierung eines Objektes mit zwei double Parametern
```

```
int x = 1.3; // funktioniert!
int x {1.3}; // erzeugt Fehlermeldung
Testklasse test (3.3, 5.4); // funktioniert!
Testklasse test {3.3, 5.4}; // erzeugt Fehlermeldung
```

➤ **Handlungsempfehlung:** Initialisierung mit {} sollte bevorzugt werden!!

# Konstruktor und Destruktor

## Standard-Konstruktor

- Ist kein Konstruktor definiert erzeugt das System automatisch einen **Standard-Konstruktor** (implizite Konstruktor Deklaration).
- Die Parameterliste des Standard-Konstruktors ist leer und es werden keine Variablen initialisiert.
- Ist mindestens ein Konstruktor definiert, wird kein Standard-Konstruktor mehr erzeugt.
  - Kann dazu führen, dass bei der Instanziierung immer Parameter erforderlich sind.

```
class Klassenname {  
    public:  
        // Einziger Konstruktor in Klasse  
        Klassenname(int a, double b);  
    ...  
}
```

```
// Einzige Möglichkeit der Instanziierung  
Klassenname objekt {2, 1.0};
```

# Konstruktor und Destruktor

## Standard-Konstruktor

- Seit **C++ 11** kann die Erzeugung eines Standard-Konstruktors eingefordert werden:

```
class Klassenname {  
    public:  
        // Erzwungener Standard-Konstruktor  
        Klassenname() = default;  
        // Konstruktor mit Parameterliste  
        Klassenname(int a, double b);  
    ...  
};
```

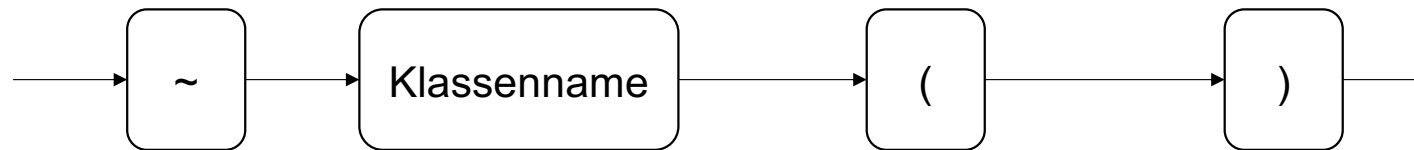
```
// Die zwei möglichen Instanziierungen  
Klassenname objekt1;  
Klassenname objekt1 {2, 1.0};
```

- Umgekehrt kann mit **delete** statt **default** auch verhindert werden, dass ein Standard-Konstruktor erzeugt wird.

# Konstruktor und Destruktor

## Der Destruktor

- Syntaxgraph



# Konstruktor und Destruktor

## Der Destruktor

### Beispiel:

```
class Testklasse {  
    // oeffentlicher Teil  
    public:  
        // Deklaration des ersten Konstruktors  
        Testklasse();  
        // Deklaration des zweiten Konstruktors  
        Testklasse(int x, int y);  
        // Deklaration des Destruktors  
        ~Testklasse();  
        // Deklaration der Methoden  
        void ändern(int x, int y);  
        void gib_werte_aus();  
    // privater Teil  
    private:  
        // Member-Variablen  
        int xKoordinate;  
        int yKoordinate;  
};
```

```
// Definition des Destruktors  
Testklasse::~~Testklasse()  
{  
    cout << "Der Destruktor von Testklasse." << endl;  
}
```

- Eine Klasse definiert genau einen Destruktor.
- Der Destruktor hat keine Parameterliste.
- Wird kein Destruktor definiert, erzeugt das System wie beim Konstruktor einen **Standard-Destruktor** (implizite Deklaration).
- Auch die Erzeugung des Standard-Destruktors kann mit `~Testklasse() = delete;` verhindert werden.

# Konstruktor und Destruktor

## Der Destruktor

- Destruktor von Testklasse wird bei Speicherfreigabe des Objektes *test* aufgerufen.

*Beispiel:*

```
int main()
{
    // Instanziiert das Objekt test
    // mit erstem Konstruktor der Klasse Testklasse
    Testklasse test;
    // Gibt Werte der Member-Variablen aus
    test1.gib_werte_aus();
    /* Ausgabe:
       xKoordinate: 0, yKoordinate: 0
    */

    return 0;
}
/* Ausgabe:
Der Destruktor von Testklasse."
*/
```

# Objektweite Gültigkeit

## Der this-Zeiger

- Das Schlüsselwort **this** erlaubt den Zugriff eines Objektes auf sich selbst.
- Mit **this** werden Member-Variablen und Methoden innerhalb der Klassenimplementierung angesprochen.
- **this** ist ein **Zeiger** und unterliegt dabei den Regeln der Dereferenzierung.
- Mit **this** können Member-Variablen und Methoden im Code kenntlich gemacht werden.
- Da **this** die Speicheradresse eines Objektes liefert, kann **this** zur Identifizierung eines Objektes genutzt werden.

```
class Klassenname {  
    public:  
        void set_variable(double variable) {  
            /* durch das vorangestellt this ist  
             klar, dass die linke Variable eine  
             Member-Variable der Klasse ist,  
             während die Rechte Variable eine  
             lokale Variable ist.*/  
            this->variable = variable;  
        }  
  
    private:  
        double variable;  
    ...  
}
```



# Objekte und Klassen in C++

## Schlüsselwörter in diesem Kapitel

alignas	const	dynamic_cast	long	short	typedef
alignof	constexpr	else	mutable	signed	typeid
asm	constexpr	enum	namespace	sizeof	typename
auto	constinit	explicit	new	static	union
bool	const_cast	export	noexcept	static_assert	unsigned
break	continue	extern	nullptr	static_cast	using
case	co_await	false	operator	struct	virtual
catch	co_return	float	<b>private</b>	switch	void
char	co_yield	for	protected	template	volatile
char8_t	decltype	friend	<b>public</b>	<b>this</b>	wchar_t
char16_t	default	goto	register	thread_local	while
char32_t	delete	if	reinterpret_cast	throw	
<b>class</b>	do	inline	requires	true	
concept	double	int	return	try	



FH MÜNSTER  
University of Applied Sciences

# Programmieren in C++

Prof. Dr. Kathrin Ungru

Fachbereich Elektrotechnik und Informatik

[kathrin.ungru@fh-muenster.de](mailto:kathrin.ungru@fh-muenster.de)