

Engine Alpha Electronics

Während sich die (ursprüngliche) Engine Alpha um das einfache Erstellen und animieren von 2D-Grafiken zum Zwecke der Spiele-Programmierung bemüht, ist das Ziel der **Engine Alpha Electronics** das Ansprechen von realer Hardware mittels eines **Arduino-Mikrokontrollers**. Somit ist es möglich, die Spiele um **Joysticks** zu bereichern. Man kann aber auch **LEDs, Taster, Lichtschranken** sowie **diverse Sensoren** (Schall, Licht, Temperatur, diverse Gase, Erschütterung, Bewegung ...) und **Motoren** (DC-Motor, Servo-Motor, ...) ansprechen und damit sehr kreativ mit der Umwelt interagieren.

Leider ist es derzeit noch nötig, dass das Arduino über ein USB-Kabel dauerhaft mit dem PC verbunden bleibt. Hierdurch resultieren derzeit noch 2 Nachteile:

- Die digitalen Pins 0 und 1 werden durch den USB-Bus belegt und können NICHT verwendet werden.
- Autonome Fahrzeuge sind derzeit wegen des USB-Kabels noch NICHT möglich.

Wenn du nicht weißt, was genau ein Arduino ist bzw. wie man es für die Benutzung mit JAVA vorbereitet, dann lies das Dokument:

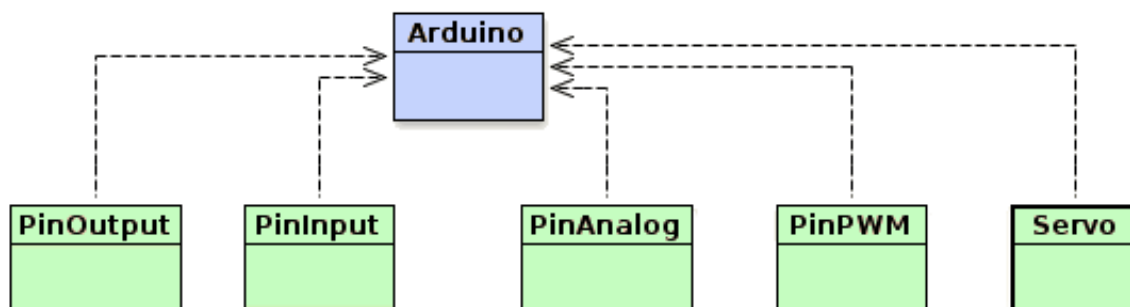


[Arduino_Quick_Guide.pdf](#)



[Arduino_mit_JAVA.pdf](#)

Konzeptuelles Klassen-Diagramm



Das Klassen-Modell versucht, ein Arduino aus Sicht eines Benutzers abzubilden:

- So gibt es eine zentrale **Klasse Arduino**, in welcher alle technischen Grundlagen zusammen gefasst wurden. Mit dieser Klasse hat man als Anfänger eigentlich nichts zu tun. Sie dient Entwicklern dazu, grundlegend neue Funktionalität hinzu zu fügen. Für den reinen Anwender stellt sie einige wenige statische Methoden zur Verfügung. (Dazu später mehr)
- Für den Einsteiger werden 5 weitere Klassen angeboten, welche die Grundfunktionalitäten eines Arduino widerspiegeln:
 - **PinOutput:** digitaler Ausgangs-Pin (ein- / ausschalten von Geräten)
 - **PinInput:** digitaler Eingangs-Pin (einfache Sensor-Abfrage: JA / NEIN)
 - **PinAnalog:** analoger Eingangs-Pin (Sensor-Abfrage mit 1024 Stufen)
 - **PinPWM:** Pulsweiten-Modulations-Pin (generieren von Rechteck-Spannungen)
 - **Servo:** Ansteuerung von Servo-Motoren

Die Basis-Klassen im Detail

Ein kleiner Überblick soll helfen, die Klassen später sinnvoll einzusetzen.

Die Klasse Arduino

Arduino
public static int ANALOG public static int HIGH public static int INPUT public static int LOW public static int OUTPUT public static int PULLUP public static int PWM public static int SERVO static IODevice device
private Arduino() public static void addEventListener(int pin, PinEventListener pel) public static IODevice getDevice() public static void pause(int milli_sec) public static long read_analog(int pin) public static boolean read_digital(int pin) public static void removeEventListener(int pin, PinEventListener pel) public static void setMode(int pin, int mode) public static void setup() public static void stop() public static void write(int pin, int value)

Das Arduino ist nach dem Singleton-Pattern implementiert. Das heißt, immer wenn man ein Objekt (irgend einen Pin, Servo) erzeugt, wird automatisch im Hintergrund ein Arduino-Objekt mit erzeugt und es gibt genau EIN solches Objekt. **Man kann also nur EIN Arduino anstecken.**

Der **Standard-Anwender** wird nur folgende Methode benötigen:

```
pause ( int milli_sec )
```

Lässt jedes Programm an dieser Stelle für eine gewisse Zeit warten, bis der nächste Befehl ausgeführt wird. Diese Methode muss statisch aufgerufen werden: **Arduino.pause(...)**

Folgende Methoden sind **für Entwickler** gedacht:

- `setup()` , `getDevice()` , `stop()`
- `setMode(int pin , int mode)` , `readAnalog(int pin)` , `readDigital(int pin)`
- `write(int pin , int value)`
- `addEventListener(...)` , `removeEventListener(...)`

[Code in den vorgegebenen Klassen lesen und nach `firmata4j` googlen]

Klasse *PinOutput*

Diese Klasse repräsentiert einen digitalen Ausgangs-Pin. Damit kann man externe „Geräte“ ein und aus schalten. Es ist empfehlenswert bei einfachen „Geräten“ mit nur einem benötigten Pin von dieser Klasse zu erben.

PinOutput
private Pin output_pin private boolean state
public PinOutput(int pin) public void high() public boolean isHigh() public boolean isLow() public void low() public void toggle()

Konstruktor

```
PinOutput ( int pin )
```

Man übergibt einfach die Nummer des Pins, an dem das „Gerät“ angeschlossen ist. Am Arduino UNO sind das die Pins 0-13.

VORSICHT : Die **Pins 0 und 1 sind verboten**, sie werden vom USB-Bus verwendet.

Weitere Methoden

```
high ( )
```

Setzt das Ausgangs-Signal auf HIGH (+5V) und schaltet damit das „Gerät“ ein.

```
low ( )
```

Setzt das Ausgangs-Signal auf LOW (GND , 0V) und schaltet damit das „Gerät“ aus.

```
toggle ( )
```

Wechselt das Ausgangs-Signal von LOW nach HIGH und umgekehrt.

```
isHigh ( )
```

Gibt *true* zurück, wenn der Pin auf HIGH gesetzt ist, sonst *false*.

```
isLow ( )
```

Gibt *true* zurück, wenn der Pin auf LOW gesetzt ist, sonst *false*.

Klasse PinInput

Diese Klasse repräsentiert einen digitalen Eingangs-Pin. Damit kann man den Zustand von externen digitalen Sensoren abfragen. Digitale Sensoren senden entweder ein HIGH oder ein LOW, aber keinerlei Zwischen-Werte.

Es ist empfehlenswert, einfache digitale Sensoren mit nur einem benötigten Pin davon erben zu lassen.

Konstruktor

```
PinOutput ( int pin )
```

Man übergibt einfach die Nummer des Pins, an dem der Sensor angeschlossen ist. Am Arduino UNO sind das die Pins 0-13.

VORSICHT : Die **Pins 0 und 1 sind verboten**, sie werden vom USB-Bus verwendet.

PinInput
protected boolean changed
protected int debounce_interval
private Pin pin_input
protected boolean state
public PinInput(int pin)
public boolean isHigh()
public boolean isLow()
public void onModeChange(IOEvent event)
public void onReceiveChange()
public void onReceiveHigh()
public void onReceiveLow()
public void onValueChange(IOEvent event)
public void setDebounce(int milli_sec)

Weitere einfache Methoden

```
isHigh ( )
```

Gibt *true* zurück, wenn der Pin auf HIGH gesetzt ist, sonst *false*.

```
isLow ( )
```

Gibt *true* zurück, wenn der Pin auf LOW gesetzt ist, sonst *false*.

```
setDebounce ( int milli_sec )
```

Manche Sensoren senden leider nicht nur ein einzelnes Signal sondern „**prellen**“. Darunter versteht man eine Folge von mehreren, schnell aufeinander folgenden HIGHs und LOWs. Dies kommt insbesondere bei billigen Tastern oder bei Schall-Sensoren vor.

Mit dieser Methode kann man ein Zeitintervall in Milli-Sekunden (Tausendstel Sekunden) setzen, so dass nach einem erkannten Signal für diesen Zeitraum NICHT auf weitere Signale reagiert wird, so dass die folgenden „Preller“ ignoriert werden.

Methoden, die nur intern aufgerufen werden

Die Methoden **onModeChange(...)** und **onValueChange(...)** werden intern durch das Observer-Pattern aufgerufen. **Es ist NICHT sinnvoll, diese Methoden von Hand aufzurufen!**

(weitere wichtige Methoden auf der nächsten Seite)

Methoden, die überschrieben werden müssen

Um automatisch über Signal-Wechsel (*HIGH* → *LOW* oder *LOW* → *HIGH*) der Sensoren informiert zu werden, muss man von dieser Klasse erben und in der Sub-Klasse eine oder mehrere der folgenden Methoden überschreiben. Der Code dieser Methoden wird dann bei entsprechenden Ereignissen automatisch aufgerufen. (Technisch wird das automatische Aufrufen durch das Observer-Pattern realisiert.)

```
onReceiveHigh ( )
```

Was soll geschehen, wenn an diesem Pin ein **LOW** → **HIGH** Ereignis stattfindet.

```
onReceiveLow ( )
```

Was soll geschehen, wenn an diesem Pin ein **HIGH** → **LOW** Ereignis stattfindet.

```
onReceiveChange ( )
```

Was soll geschehen, wenn an diesem Pin ein **beliebiger Signal-Wechsel** stattfindet.

Klasse PinAnalog

Diese Klasse repräsentiert einen analogen Eingangs-Pin. Damit kann man den Zustand von externen analogen Sensoren abfragen. Analoge Sensoren senden beliebige Signal-Werte fließend von 0V (*GND*) bis +5V (*Vcc*). Ein analoger Arduino-Pin wandelt diese Spannungs-Werte in ganzzahlige Werte von 0 (*GND*, 0V) bis 1023 (+5V) um.

PinAnalog
private Pin analog_pin
public PinAnalog(int pin)
public long readValue()

Konstruktor

```
PinAnalog ( int pin )
```

Man übergibt einfach die Nummer des Pins, an dem der Sensor angeschlossen ist. Am Arduino UNO sind das die Pins 14-19 bzw. A0-A5.

Weitere Methode

```
readValue ( )
```



Gibt einen Wert von 0 (*GND*, 0V) bis 1023 (+5V) zurück.

Klasse PinPWM

PWM steht für **Pulsweiten-Modulation**. Man versteht darunter eine Rechteck-Spannung: Das Signal ist eine gewisse Zeit (sog. duty cycle) auf HIGH und danach wieder auf LOW und das geschieht periodisch immer wieder.

Damit kann man z.B. LEDs dimmen oder mit Hilfe eines Motor-Treibers die Geschwindigkeit eines Gleichstrom-Motors regulieren.

Wenn dir noch nicht ganz klar ist, was Pulsweiten-Modulation ist, so lies folgende Dokumente durch:

-  [Arduino_Quick_Guide.pdf](#)
-  [Motortreiber_H-Bruecke.pdf](#)

PinPWM
private int duty_percentage
private Pin pwm_pin
public PinPWM(int pin)
public int getDutyCyclePercentage()
public void setDutyCyclePercentage(int percentage)

Konstruktor

```
PinPWM ( int pin )
```

Man übergibt einfach die Nummer des Pins, an dem man das PWM-Signal erzeugen möchte.
Am Arduino UNO stehen nur 6 Pins zur Verfügung, die PWM unterstützen.
Dies sind die Pins 3 , 5 , 6 , 9 , 10 , 11 .

Weitere Methoden

```
setDutyCyclePercentage ( int percentage )
```

Setzt den prozentualen Anteil von HIGH an einer Periode des PWM-Signals im Bereich von 0 bis 100.

```
getDutyCyclePercentage ( )
```

Nennt den aktuellen Anteil von HIGH an einer Periode des PWM-Signals.

Klasse Servo

Diese Klasse repräsentiert einen Servo-Motor. Ein Servo ist ein Motor, der wie ein Scheibenwischer in einem Bereich von etwa 180° positioniert werden kann. Ein Servo hat 3 Anschlüsse: GND (Minus), Vcc (Plus) und ein PWM-Signal, das die Stellung des Servo angibt.

Servo
private Pin servo_pin
public Servo(int pin) public void center() public long getValue() public void left() public void right() public void setValue(int value)

Konstruktor

`Servo (int pin)`

Man übergibt einfach die Nummer des Pins, an dem man das PWM-Signal erzeugen möchte.
Am Arduino UNO stehen nur 6 Pins zur Verfügung, die PWM unterstützen.
Dies sind die Pins 3 , 5 , 6 , 9 , 10 , 11 .

Weitere Methoden

`center ()`

Stellt den Servo in etwa in die mittige Position.

`left ()`

Dreht den Servo ganz auf eine Seite.
Ob das links oder rechts ist, hängt davon ab, von welcher Seite man den Servo anschaut.

`right ()`

Dreht den Servo ganz auf die andere Seite.
Ob das links oder rechts ist, hängt davon ab, von welcher Seite man den Servo anschaut.

`setValue (int value)`

Stellt den Servo auf eine beliebige Position.
Es sind Werte von **0 bis 180** zulässig. Die Werte entsprechen in etwa Winkelgraden.

`getValue ()`

Nennt den aktuellen Wert der Position.

Auf der nächsten Seite finden sich noch einige Methoden, um die Klasse Servo an einige spezielle Servo-Typen anzupassen bzw. um die Laufgeschwindigkeit des Servos anzupassen:

setDelay (int delay)

Stellt ein, wie schnell der Servo sich auf eine neue Position bewegt. Standard-Wert ist 20. Kleinere Werte bewirken schnelleres Laufen, größere Werte bewirken langsames Laufen.

setLeft (int left)

Stellt den Wert ein, der den Servo ganz nach links dreht. Standard-Wert = 1.
Wenn der Servo ganz links brummt, sollte dieser Wert erhöht werden, bis es nicht mehr brummt. (Der Wert für „Mitte“ wird automatisch mit angepasst.)
Benötigt man den Servo „auf dem Kopf stehend“, kann man damit links und rechts vertauschen, so dass die Richtungen wieder stimmen.

setRight (int right)

Stellt den Wert ein, der den Servo ganz nach rechts dreht. Standard-Wert = 179.
Wenn der Servo ganz rechts brummt, sollte dieser Wert erhöht werden, bis es nicht mehr brummt. (Der Wert für „Mitte“ wird automatisch mit angepasst.)
Benötigt man den Servo „auf dem Kopf stehend“, kann man damit links und rechts vertauschen, so dass die Richtungen wieder stimmen.

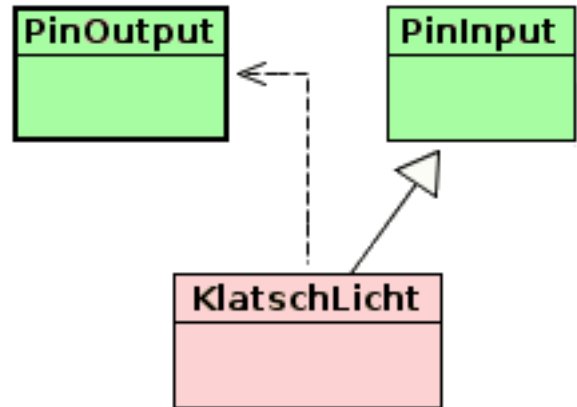
 [Servo.pdf](#)

Klassen zu speziellen „Geräten“

Um zu erläutern, wie man die elementaren Klassen verwenden kann, um reale Hardware abzubilden, existieren einige Demo-Klassen:

Klatsch-Licht

Ein Klatsch-Licht geht an bzw. wieder aus, wenn man laut in die Hände klatscht. Hierzu benötigt man einen Ausgangs-Pin (z.B. für eine LED oder für ein Relay, das eine richtige Lampe steuert) und einen Eingangs-Pin für einen Sensor, der auf Klatschen reagiert. Damit die Reaktion auf das Klatschen automatisch erfolgt, ohne dass wir im Programm ständig nach dem Zustand des Klatsch-Sensors fragen müssen, nutzen wir das Observer-Pattern des digitalen Input-Pins. Hierzu erbt die **Klasse KlatschLicht** von `PinInput` und überschreibt die Methode `onReceiveHigh()`. Um das „Prellen“ des Klatsch-Sensors zu beseitigen, wird das Debounce-Intervall des Input-Pins auf etwa 100 Milli-Sekunden gesetzt. Der Output-Pin wird im Gegensatz zum Input-Pin referenziert.



Details können im Code der Klasse nachgesehen werden ...

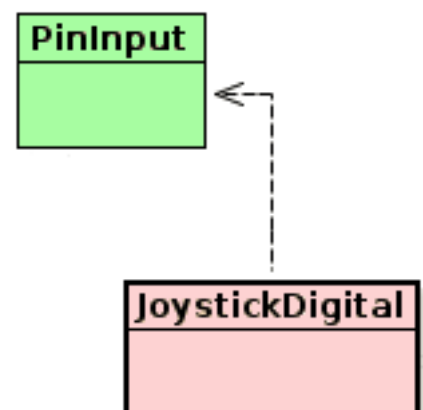
➔ [Schallsensor.pdf](#)

Digitaler Joystick

Ein digitaler Joystick besteht eigentlich nur aus 4 Push-Buttons (in der Regel mit `PULLUP` Widerstand), die über einen „Hebel mit Knauf“ bedient werden. Entsprechend referenziert diese Klasse im Prinzip 4 Objekte vom Typ `PinInput`.

Damit auch hier automatisch auf die Ereignisse der Input-Pins reagiert werden kann, verwendet die **Klasse JoystickDigital** eine interne Klasse `JoystickButton`, die von `PinInput` erbt und die Methode `onReceiveLow()` entsprechend überschreibt. Man hat mit dieser Klasse allerdings nichts zu tun.

Es gibt 4 Methoden `up()`, `down()`, `left()`, `right()`, die auf die 4 Buttons reagieren. Sinnvollerweise schreibt man eine Sub-Klasse zur Klasse `JoystickDigital` und überschreibt darin diese 4 Methoden um sinnvoll auf die Eingaben des Joysticks reagieren zu können.



Details können im Code der Klasse nachgesehen werden ...

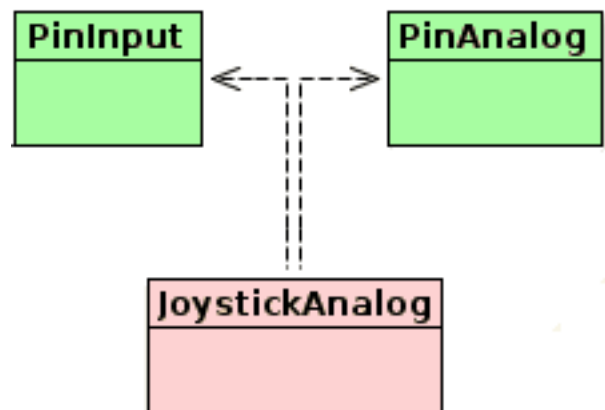
➔ [Joystick_Digital.pdf](#)

Analoger Joystick

Ein analoger Joystick besteht aus 2 Potentiometern, eines für die x- und das andere für die y-Richtung. Die Spannung dieser Potentiometer wird durch 2 analoge Eingangs-Pins abgefragt.

Oft haben analoge Joysticks einen Push-Button, der über einen digitalen Eingangs-Pin abgefragt werden kann.

Die **Klasse JoystickAnalog** wird in einer anderen Klasse referenziert und die Methoden `getX()`, `getY()`, `isPressed()` benutzt.



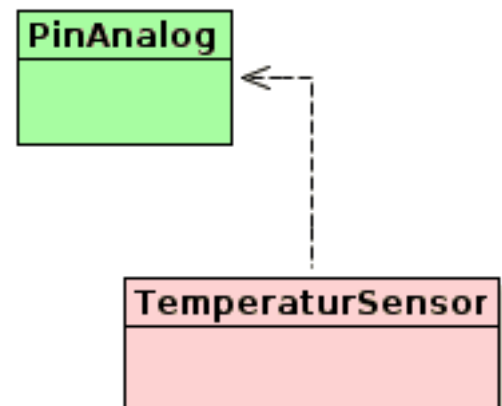
Details können im Code der Klasse nachgesehen werden ...

➔ [Joystick_Analog.pdf](#)

Analoger Temperatur-Sensor

Einen analogen Temperatur-Sensor kann man sehr leicht bauen aus einem Widerstand in Reihe zu einem NTC. Das analoge Signal braucht nur in eine Angabe in Grad Celsius umgerechnet zu werden.

Entsprechend erbt die **Klasse TemperaturSensor** von der Klasse `PinAnalog` und erweitert diese um die Methoden `getCelsius()` zum Nennen der Temperatur.



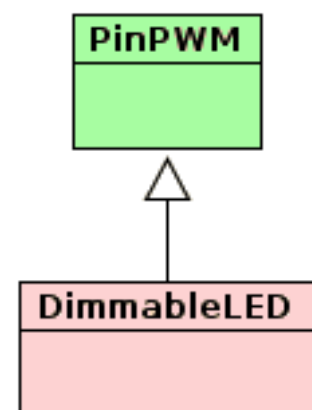
Details können im Code der Klasse nachgesehen werden ...

➔ [Temperatursensor_NTC.pdf](#)

Dimmbare LED

Eine dimmbare LED wird einfach über einen PWM-Pin angesteuert. Damit ist es naheliegend, die **Klasse DimmableLED** von `PinPWM` erben zu lassen und bei Bedarf um weitere sprechende Methoden zu erweitern.

Details können im Code der Klasse nachgesehen werden ...



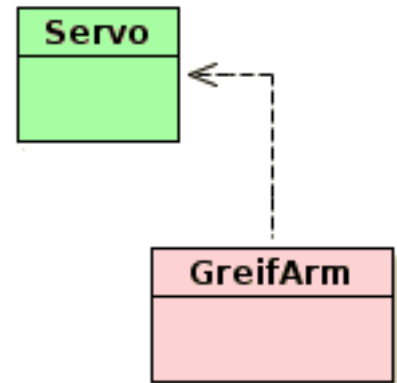
➔ [Arduino_Quick_Guide.pdf](#)

Greifarm

Einfache Greifarme bestehen aus Gelenken, die von Servo-Motoren bewegt werden. Referenziert man in einer **Klasse Greifarm** mehrere Servos und schreibt für jeden Servo sprechende Methoden wie z.B. `rauf()`, `runter()`, `links()`, `rechts()`, `hand_auf()`, `hand_zu()`, ... so kann man sehr leicht einfache Bewegungen mit dem Greifarm programmieren.

VORSICHT: Servos brauchen teilweise mehr Strom als das Arduino liefern kann. Hier muss man sich mit einer externen Stromversorgung behelfen ...

Details können im Code der Klasse nachgesehen werden ...



 [Servo_.pdf](#)

