**JAVA**

**Was ist JAVA?**

Java ist eine objektorientierte Programmiersprache und gleichzeitig auch eine eigene Plattform. Die Definition von Plattform ist die Kombination aus Computer und Prozessor. Da Java eine objektorientierte Programmiersprache ist, arbeitet man in Java nahezu nur mit Objekten. Sogenannte Klassen sind Blaupausen für diese Objekte!

**Warum JAVA?**

Da wir seit der dritten Klasse das Programmieren mit Java lernen, auch in der Freizeit einiges in Java programmieren und dadurch bereits viel Erfahrung im Umgang mit Java gesammelt haben, entschieden wir uns für diese Programmiersprache. Ein weiterer Grund ist die große Unterstützung dieser Sprache im Internet, und die Plattformunabhängigkeit.

**Vor- und Nachteile?**

|  |  |
| --- | --- |
| Vorteile | Nachteile |
| Java ist gratis | Java ist speicheraufwändig |
| Java läuft auf jedem Betriebssystem mit einer installierten Java virtuellen Maschine. | Java ist ein wenig langsamer als andere Programmiersprachen |
| In Java gibt es einigen bereits vorprogrammierten Code | Ein paar Klassenbibliotheken sind noch nicht fertig ausprogrammiert |
| Die Java Community ist die größte Programmiercommunity weltweit. |  |

**Alternativen?**

* C++
* C#
* Python

**Genaueres zum Code?**

**DBManager**

Funktion dieser Klasse ist es die Verbindung zur Datenbank zu „managen“. Dabei ist die Art der Datenbank egal. Diese wird nämlich in einer Properties-Datei bestimmt. Mehr dazu später. Der DBManager verwaltet die Tabellen „sensordaten“ und „users“. Mittels diverser Methoden des DBManagers wird uns das schnelle speichern und auslesen von Sensordaten und Userdaten ermöglicht.

**Wie funktioniert das Aufbauen einer Datenbankverbindung in Java?**

Dazu ist ein .jar-Container für die jeweilige Datenbank nötig. Dieser beinhaltet den jeweiligen Code zum Aufbauen einer Verbindung zur jeweiligen Datenbank. Ein Beispiel dafür wäre z.B. der „MYSQL-Connector“ oder „SQLITE-JDBC“.

Code der beim Erstellen eines DBManager-Objekts aufgerufen wird:

Connection con;

public DBManager()

{

Class.forName(“com.mysql.jdbc.Driver”);

con = (Connection) DriverManager.getConnection(

“jdbc:mysql://localhost:3306/fish”,”root”,””);

}

**Wie funktioniert die Kommunikation mit dem AVR-NetIO-Board?**

Diese wird mithilfe des SocketManagers durchgeführt.

**SocketManager**

Aufgabe des SocketManagers ist es Socket-Verbindungen zu “managen“. Socket-Verbindungen werden dazu benötigt um eine Netzwerkverbindung mit unserem   
AVR-NetIO-Board aufzubauen. Die Methoden des Socket-Managers ermöglichen uns das schnelle Senden und Empfangen von Nachrichten. Beispiele für Befehle die wir dem AVR-NetIO-Board senden:

* GETADC 1 🡪 Liefert die Spannung am ADC1 als ganzzahligen Wert der zwischen 0 und 1023 liegt.
* SETPORT 1 1 🡪 Schließt das Relais 1 der am AVR-NetIO-Board angeschlossenen K8IO-Relaiskarte

**Wie funktioniert das Einlesen von Sensorwerten?**

Unser AVR-NetIO-Board sendet uns die Spannung als digitalen ganzzahligen Wert mithilfe des GETADC-Befehls, welcher von unserem SocketManager gesendet wird. Dieser Wert wird dann von unserer Tool-Klasse in eine Spannung umgerechnet. Das geschieht folgendermaßen:

Wobei die Referenzspannung in unserem Fall 5,0V und die Bitbreite 10 ist.

Beispiel (ADC-Wert ist 256):

Diese Spannung muss nachher in einen Temperaturwert umgerechnet werden. Da bei unserem Temperatursensor (KTY-222) im Bereich zwischen 0 und 50° Celsius ein lineares Verhältnis zwischen Widerstand und Temperatur besteht, ist es möglich die Temperatur mittels einer arithmetischen Gleichung erster Ordnung zu bestimmen.

Berechnung der Geradengleichung:

Durch Einsetzen in die Gleichung ist es uns möglich die Werte k und d zu bestimmen.

🡪 k=15,5

🡪 d=1635

**Wie funktioniert das Schreiben/Lesen von Dateien?**

Dazu schrieben wir uns zwei Methoden (read, write) in unsere Tool-Klasse.

Read-Methode:  
Diese Methode hat als Parameter den Dateinamen der Datei und als Rückgabewert ein String-Array. Im Grunde genommen wird die Datei zeilenweise ausgelesen und in die einzelnen Zeilen in ein Array geschrieben. Das zu retournierende Array wird so vorbereitet, dass es nur so groß ist wie die Datei Zeilen beinhaltet.

Write-Methode:  
Diese Methode hat als Parameter den Dateinamen der Datei und ein String-Array. Das String-Array beinhaltet die einzelnen Zeilen, die dann ausgelesen werden und in eine Datei geschrieben werden.

**Property-Files**

In Java ist es durchaus üblich sogenannte Property-Files zu erstellen. Diese Dateien erlauben es das Programm komfortabel zu konfigurieren. Als „Properties“ (deutsch Eigenschaften) werden statische Variablen konfiguriert wie z.B. Datenbank-Adressen, IP-Adressen usw. Property-Files haben üblicherweise über die Dateiendung „.properties“.

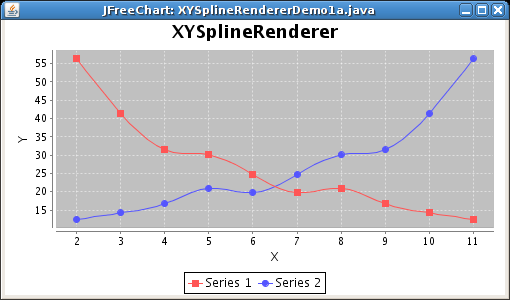
**log4j**

log4j ist ein Tool zum Loggen von Nachrichten (Informationen, Warnungen, Fehler…) in Java. Verwendet wird dieses in nahezu jedem größeren Open-Source-Java-Projekt. Es ist viel komfortabler dieses Tool zu verwenden als die von Java mitgelieferten Logger-Klassen. Der log4j-Logger kann mittels Property-Files konfiguriert werden.

Code zum Erstellen eines Loggers und ausgeben einer Informations-Nachricht:

public logger=Logger.getLogger(„Logger“);  
logger.info(“Hello World!”);

**JFreeChart**

Das JFreeChart Projekt wurde im Jahre 2000 von David Gilbert ins Leben gerufen. Heutzutage ist JFreeChart die meist genutzte Graphenerstellungs-Software weltweit. Wie der Name schon sagt ist die Benutzung von JFreeChart kostenlos. JFreeChart lässt sich optimal in den Java Code integrieren.

Alternativen zu JFreeChart:

* Simile Widgets
* rrdTool

Ähnlich wie JFreeChart sind diese Graphenerstellungs-Programme in der Lage Daten grafisch darzustellen. Der Nachteil bei diesen Anwendungen ist jedoch, dass sie nicht für Java entworfen wurden. Sie sind daher relativ umständlich in ein Java-Programm integrierbar. Deshalb fiel unsere Wahl auf JFreeChart.

**Klassen**

**Klasse Tool**

Diese Klasse beinhaltet beinahe alle Funktionen, die für unsere Software benötigt werden. Die Funktionen sind alle als „static“ deklariert, sodass kein Objekt der Klasse Tool erzeugt werden muss um die Funktionen aufzurufen.

**Methoden der Klasse Tool**

**Methode „write“**

void write(String path, String[] str)  
Schreibt das Array „str“ in eine Datei mit dem Pfad „path“.  
Wird in unserem Fall für das Speichern der Konfiguration (Uhrzeiten der geplanten Fütterung) verwendet. Methode „write“ wird für den jetzigen Stand der Programmierung nicht mehr benötigt.

**Methode „read“**

String[] read(String path)  
Liest eine Datei mit dem Pfad „path“ ein und gibt den Inhalt als String-Array zurück.  
Wird in unserem Fall für das Einlesen der Konfiguration (Uhrzeiten der geplanten Fütterung) verwendet. Methode „read“ wird für den jetzigen Stand der Programmierung nicht mehr benötigt.

**Methode „readFishConfig“**

String[] readFishConfig()  
Ruft die Methode „read“ mit dem Pfad der Fischfütterungsdatei auf, falls diese überhaupt existiert und gibt den Inhalt als String-Array zurück. Methode „readFishConfig“ wird für den jetzigen Stand der Programmierung nicht mehr benötigt.

**Methode „writeFishConfig“**

void writeFishConfig(String[] str)  
Ruft die Methode „write” mit dem Pfad der Fischfütterungsdatei auf, falls diese überhaupt existiert. Existiert sie nicht, wird eine neue Datei erstellt. In die Datei wird ein String-Array geschrieben. Methode „writeFishConfig“ wird für den jetzigen Stand der Programmierung nicht mehr benötigt.

**Methode „fishConfigExists“**

boolean fishConfigExists()  
Schaut nach, ob die Fischfütterungsdatei existiert und returniert „true“ falls diese existiert, ansonsten wird „false“ returniert. Methode „fishConfigExists“ wird für den jetzigen Stand der Programmierung nicht mehr benötigt.

**Methode „getTemperatureFromVoltage“**

float getTemperatureFromVoltage(float voltage)  
Berechnet und returniert die resultierende Temperatur aus der gemessenen Spannung.

**Methode „getTemperature“**

float getTemperature(SocketManager sman, int adc)  
Ruft die Methode „getVoltageFromDigital” und „getTemperatureFromVoltage” auf, um die Temperatur als float-Variable zu returnieren.

**Methode „round“**

float round(float d, int decimalPlace)  
Diese Funktion rundet die Zahl „d“ ab der Kommastelle „decimalPlace“ auf oder ab. Es handelt sich hier um algebraisches Runden. Zurückgegeben wird eine float-Variable.

**Methode „getVoltageFromDigital“**

float getVoltageFromDigital(int i)  
Berechnet und returniert die resultierende Spannung aus dem digitalen Wert „i“.

**Methode „wait“**

void wait(int milliseconds)  
Wartet „milliseconds” Millisekunden. Die Exceptions werden bereits in der Methode selbst behandelt.

**Methode „ping“**

boolean ping()  
Pingt das AVR-NetIO-Board und returniert „true“ falls der Ping erfolgreich war. Ansonsten wird false returniert.

**Methode „sendMail“**

void sendMail(String subject, String text)  
Startet den “JavaMailThread”, welcher ein E-Mail mit dem Titel “subject” und dem Inhalt “text” an den konfigurierten Empfänger versendet.

**Methode „SgetTime“**

String SgetTime(String format)  
Gibt die aktuelle Zeit als String mit dem Format „format” zurück.

**Methode „IgetTime“**

int IgetTime(String format)  
Gibt die aktuelle Zeit als Integer mit dem Format „format“ zurück.

**Methode „fetch“**

void fetch(SocketManager sman)  
Baut mithilfe des SocketManagers eine Socket-Verbindung mit dem AVR-NetIO-Board auf und liest die Wasser- und Lufttemperatur ein. Ist ein Wert kritisch wird eine Warnnachricht als E-Mail versendet. Zusätzlich werden die Daten in die SQLite-Datenbank gespeichert.

**Methode „feed“**

void feed(SocketManager sman)  
Baut mithilfe des Socket-Managers eine Socket-Verbindung mit dem AVR-NetIO-Board auf und startet den Motor und somit die Fütterung. Mit einer Zeitverzögerung von 500ms wird auch noch die Spannung an der Lichtschranke gemessen. Ändert sich diese wird der Motor gestoppt. Andernfalls haltet der Motor auf jeden Fall nach 4 Sekunden.

**Methode „timeFormat“**

String timeFormat(int i)  
Returniert eine “korrekt” formatierte Zeit. Beispielsweise wird diese Funktion zum Umformatieren von 9:15 in 09:15 verwendet.

**Methode „getGauge“**

String getGauge(float temperature)  
Ist für die grafische Darstellung der Temperatur beim Web-Interface verantwortlich. Dazu wird ein transparentes Pixel skaliert um ein Balkendiagramm zu erzeugen. Dieser ist grün wenn die Temperatur einen normalen Wert aufweist, rot wenn dieser zu heiß ist und blau wenn dieser zu kalt ist.

**Methode „md5“**

String md5(String input)  
Hat die Aufgabe den String „input“ mit dem MD5-Hash-Algorithmus zu hashen. Der gehashte Wert wird zurückgegeben. Da das Passwort zum Anmelden beim Web-Interface nur in gehashter Form in der Datenbank gespeichert ist, muss das eingegebene Passwort gehasht werden und kann erst dann mit dem in der Datenbank gespeicherten gehashten Passwort verglichen werden.

**Methode „StringToDate“**

Date StringToDate(String s)  
Wandelt einen Datums-String in ein Date-Object um und returniert dieses. Diese Methode muss beim Einlesen des Datums aus der SQLite-Datenbank verwendet werden. Das SQLite-Datums-Objekt ist nämlich nicht mit den Java-Datums-Objekten kompatibel.

**Methode „createThermometer“**

JFreeChart createThermometer(String s)  
Erstellt einen JFreeChart Thermometer-Chart für die Wasser- oder Lufttemperatur und returniert diesen. Er wird dann auf der „index.jsp“ mit dem jeweiligen Servlet angezeigt.

**Methode „createGraph“**

JFreeChart createGraph()  
Erstellt einen JFreeChart Timeseries-Chart für die Wasser- und Lufttemperatur und returniert diesen. Er wird dann auf der „graph.jsp“ Seite mithilfe des ChartViewer-Servlets angezeigt.

**Klasse „SocketManager“**

**Funktionen der Klasse „SocketManager“**

**Methode „init“**

void init()  
Erzeugt die Socket-Verbindung mit dem AVR-NetIO-Board und ermöglicht somit die Kommunikation.

**Methode „close“**

void close()  
Beendet die Socket-Verbindung.

**Methode „isConnected“**

boolean isConnected()  
Returniert “true” falls eine Verbindung mit dem AVR-NetIO-Board aufgebaut wurde. Andernfalls wird „false“ returniert.

**Methode „GETADC“**

int GETADC(int adc)  
Sendet den Befehl GETADC mit dem Parameter „adc“ an das AVR-NetIO-Board. Der empfangene String wird in einen Integer umgewandelt und returniert.

**Methode „SETPORT“**

void SETPORT(int port, int status)  
Sendet den Befehl SETPORT mit den Parametern „port“ und „status“ an das AVR-Net-IO-Board.

**Methode „schreibeNachricht“**

void schreibeNachricht(String nachricht)  
Schickt einen Befehl an das AVR-NetIO-Board.

**Methode „leseNachricht“**

String leseNachricht()  
Empfängt eine Nachricht vom AVR-NetIO-Board.

**Klasse DBManager**

**Methode „createDB“**

void createDB()  
Erzeugt eine neue Datenbank-Tabelle „sensordaten“, „users“ und „konfiguration falls diese noch nicht existieren. Weiters wird ein User „foo“ mit Passwort „bar“ erzeugt. Folgendes SQL-Statement wird verwendet:

create table if not exists sensordaten  
(  
id integer primary key autoincrement,  
wassertemperatur float default 0,  
lufttemperatur float default 0,  
zeitpunkt timestamp default current\_timestamp  
);

create table if not exists users  
(  
id integer primary key autoincrement,  
username varchar(8) not null,  
password varchar(32) not null  
);

create table if not exists configuration  
(  
id integer primary key autoincrement,  
string varchar(999) not null  
);

insert into users (username, password) values (“foo”, “bar”);

**Methode “recreate”**

void recreate()  
Löscht die Tabellen „sensordaten”, „users” und „konfiguration” und erzeugt diese neu.

**Methode „close“**

void close()  
Beendet die Datenbankverbindung.

**Methode „login“**

boolean login(String username, String password)  
Kontrolliert, ob der eingegebene Username mit dem eingegebenen Passwort in der Datenbank existiert. Dazu muss das eingegebene Passwort vorher gehasht werden um es mit der Datenbank zu vergleichen. Ist der Username mit dem dazugehörenden Passwort in der Datenbank vorhanden wird „true“ returniert. Ansonsten wird „false“ returniert.

**Methode „speichern“**

void speichern(Daten d)  
Speichert die Eigenschaften des Objekts „Daten“ in die Datenbank“.

**Methode „getAll“**

List<Daten> getAll()  
Returniert eine Liste mit allen Daten aus der Datenbank.

**Methode „getLastEntries“**

List<Daten> getLastEntries(int count)  
Returniert eine Liste der letzten „count“ Daten aus der der Datenbank. Wird für die Erzeugung des Temperaturgraphen benötigt.

**Methode „getConfig“**

String[] getConfig()  
Liest die Zeichenkette „string“ aus der Tabelle konfiguration aus, erzeugt ein String-Array und returniert dieses.

**Methode „setConfig“**

void setConfig(String s[])  
Schreibt das String-Array „s“ in Datenbank-Tabelle „konfiguration“.

**Klasse Daten**

Dieses Objekt stellt einen Datensatz für die Datenbank dar. Es beinhaltet die Eigenschaften id, wtemp, ltemp, zeitpunkt. Diese können mit Getter- und Setter-Methoden abgefragt und gesetzt werden.

**Klasse „JavaMailThread“**

Ist für das Senden von E-Mails verantwortlich. Da dies im Hintergrund passieren soll, wurde hierfür ein Thread programmiert. Dieser wird mithilfe der Methode „sendMail“ in der Klasse Tool erzeugt und gestartet. Wurde ein E-Mail gesendet wird dieser automatisch wieder gestoppt.

**Klasse Data**

Diese Klasse liest die Datei server.properties ein und speichert deren Eigenschaften. Diese Klasse wird immer dann benötigt, wenn z.B. eine Datenbank-Verbindung, eine Socket-Verbindung oder Ähnliches gestartet wird.

**Klasse MainThread**

Dieser Thread stellt das Hauptprogramm dar. Wenn der Thread gestartet wird, wird vorerst sichergestellt, dass überhaupt eine Verbindung mit dem AVR-NetIO-Board hergestellt wurde. Dafür wird die Methode „ping“ gestartet. Danach wird gewartet bis eine neue Minute beginnt. Ist dies der Fall wird die Methode „fetch“ aufgerufen und die Sensor-Daten in die Datenbank gespeichert. Falls in der Fischfütterungsdatei, die aktuelle Uhrzeit als geplante Fütterungszeit gespeichert ist, wird zusätzlich die Methode „feed“ aufgerufen und die Fische werden gefüttert.

**Klasse Laucher**

Diese Klasse ist für das Starten des „MainThread“ verantwortlich. Der „MainThread“ wird gestoppt durch Eingabe eines beliebigen Zeichens oder Worts in die Konsole.

**Servlets**

Servlets sind Java-Klassen, die auf Webservern liegen und Anfragen von Clients entgegennehmen und beantworten. Hauptsächlich werden sie dazu verwendet Session-Daten auszulesen und weiterzuverarbeiten. Für den grafischen Aufbau verwendet man eher JavaServer-Pages. Deshalb wird auch nach dem Entwurfsmuster „Model-View-Controller“ programmiert, wobei sich die JavaServer-Pages im „View-“ und die Servlets sich im „Controller-Package“ befinden.

Das Konfigurations-Servlet

Ein tolles Feature unseres Web-Interfaces ist der Konfigurationsmanager. Eine Kombination der Vorteile von Servlets, Java Server Pages und AJAX-Scripts bietet optimalen Komfort beim Konfigurieren der Fischfütterungszeiten.

Das Datenbank-Servlet

Der „Datenbankviewer“ unseres Webinterfaces beinhaltet neben der Funktion zum Anzeigen von Sensorwerten aus einer SQLite-Datenbank eine innovative Methode zur grafischen Darstellung von Temperaturdaten. Dieses Feature wurde mithilfe eines transparenten Pixels und einer Farbenskala (blau = zu kalt, grün = normal, rot = zu heiß) realisiert. Unterschreitet beispielsweise die gemessene Temperatur einen Temperaturmindestwert wird die grafische Darstellung für diesen Datensatz in blauer Farbe durchgeführt.

****  
  
Das Anmelde-Servlet

Aufgabe dieses Servlets ist es, den generellen Zugriff auf das Web-Interface zu gewähren. Dazu werden die Userdaten, die beim „login.jsp“ eingegeben wurden mit der SQLite-User-Datenbank verglichen. Das eingegebene Passwort muss jedoch zuerst mit einem MD5-Hash gehasht werden um eine Abfrage zu ermöglichen.

Das Abmelde-Servlet

Dieses Servlet erfüllt lediglich die Aufgabe, den Zugang auf das Web-Interface wieder zu sperren und die Session zurückzusetzen.

Das ChartViewer-Servlet

Dieses Servlet ist für die Darstellung des Temperaturgraphen auf der „graph.jsp“ Seite verantwortlich. Hierbei wird die Grafik direkt auf den Output-Stream geschrieben.

Das LuftThermo-Servlet

Dieses Servlet ist für die Darstellung der Luftthermometer-Grafik auf der „index.jsp“ Seite verantwortlich. Hierbei wird die Grafik direkt auf den Output-Stream geschrieben.

Das WasserThermo-Servlet

Dieses Servlet ist für die Darstellung der Wasserthermometer-Grafik auf der „index.jsp“ Seite verantwortlich. Hierbei wird die Grafik direkt auf den Output-Stream geschrieben.

**JavaServer Pages**

Mit JavaServer Pages (JSP) ist es möglich, dynamischen Webinhalt in einer Kombination aus HTML und der Programmiersprache Java zu erstellen. Da HTML eine Markup-Language ist und somit nicht in der Lage ist Variablen zu deklarieren geschweige denn zu nutzen, stellen für uns JavaServer Pages die optimale Alternative zu HTML dar. JavaServer Pages erlauben es Java-Code in ein HTML-Dokument zu implementieren. Dabei ist es jedoch nicht empfehlenswert den ganzen Java Code in ein JSP zu verlegen. Das führt zu Geschwindigkeitsverlusten und zu Unübersichtlichkeit.

Beschreibung: C:\Users\Berny\Diplomarbeit\Diplomschrift\Bilder\212px-Apache_Software_Foundation_Logo.svg.png

**Apache Tomcat Server**

Der Apache Tomcat Server ist ein Open-Source-Software-Projekt der Apache Software Foundation. Dieser Webserver ist in der Lage Java Servlets und JavaServer Pages zu verwalten. Dabei verfügt der Tomcat Server über einen JSP-Compiler namens Jasper. Dieser übersetzt JavaServer Pages in Servlets und bringt diese zur Ausführung. Die aktuellste Version des Apache Tomcat Servers ist 7.0.25.

Mit der Java-Entwicklungsumgebung ist das Arbeiten mit Apache Tomcat besonders konfortabel. Ohne Entwicklungsumgebung wäre es relativ umständlich ein Servlet am Apache Tomcat zu registrieren und lauffähig zu machen. Mit Eclipse ist es möglich einen Server ähnlich wie eine gewöhnliche Klasse zu erstellen.  
Mit Rechtsklick 🡪 New 🡪 Other 🡪 Server wird ein neuer Server erstellt. Im darauffolgenden Konfigurationsdialog kann man sich für eine gewünschte Version des Apache Tomcat Servers entscheiden. In unserem Fall war das die Version 7. Ist ein Server erstellt kann dem Server ein „Dynamic Web Project“ hinzugefügt werden. Wenn man anschließend den Server startet ist wird gleichzeitig die Webanwendung am Server deponiert und lauffähig gemacht.

Soll die Webanwendung irgendwann zu produktiven Zwecken genutzt werden ist es mit Eclipse möglich das Projekt in eine WAR (Web Archive)-Datei zu exportieren. Diese Datei kann dann im Apache Tomcat Verzeichnis unter Webapps deponiert werden.

Um einen Apache Tomcat Server manuell zu starten muss man im bin-Verzeichnis des Tomcats die Batch-Datei startup.bat starten. Dazu muss jedoch im Vorhinein bereits der Java-Pfad als Systemvariable deklariert worden sein. Zum Stoppen des Tomcat Servers ist lediglich die Batch-Datei shutdown.bat auszuführen.

Beim Starten des Tomcats sowie zu seiner Laufzeit wird jede Änderung einer Datei im Verzeichnis Webapps erkannt und nötige Schritte durchgeführt. Wird beispielsweise ein Web Archiv im Verzeichnis Webapps deponiert entpackt der Tomcat Server dieses Archiv und bereitet die Webapplikation für die Ausführung vor.

