Grundlagen - Kapitel 1: Die Scriptsprache und ihre Syntax

In diesem Kapitil (es könnte u.U. mehrere Artikel umfassen, wird gesondert gekenntzeichnet) werde ich einige grundlegende Aspekte der Script-Erstellung aufzeigen.
Gerade bei den Kleinigkeiten, die zu den Grundlagen gehören, kommt es häufig zu Fehlern, die man gerne vermeiden möchte und diese auch vermeiden sollte.

1.1 Integration von Scripts in einem Asset

Scripts werden in der config.txt eines Assets referenziert.

Code: config.txt

```
script
class                          "[NAME DER OBJEKT-KLASSE]"
```

Dabei werden die entsprechenden Einträge in der config.txt (s.o.) mit den jeweiligen Informationen zum Script gefüllt.

1.1.1 Einbindung externer Scripts

Ja, auch Scripts, die nicht im Asset liegen können referenziert werden. Dazu verwendet man in der config.txt den sog. "script-include-table".
Dieser erweitert den Standard-Verweis auf die in Trainz eingebauten Scripts um die des/der fremden Assets. Diese lassen sich dann mit einem "include"-Befehl im Asset-eigenen Scripts direkt einbinden.

Code: config.txt

```
script
class

script-include-table
{


}
```

Durch diese Einträge können wir nun ganz einfach aus unserem Script (hier: "meinScript.gs") die Scripts aus denen im "script-include-table" aufgeführten Assets einbinden:

Code: meinScript.gs

```
include                                                    "MapObject.gs"
include "[SCRIPT AUS ANDEREM ASSET]"
```

Damit wären wir schon beim nächsten Punkt: Dem Include-Befehl.

Mit dem Include-Befehl lassen sich oben im Script (ganz oben, als aller erstes) verschiedene andere Scripts einbinden. Im Normalfall wird es sich hierbei um die mitgeliefierten Scripts aus Trainz handeln, die uns die Grundfunktionen für unsere Objekttypen mitgeben. Aber auch fremde Scripts lassen sich hier ganz einfach einbinden.
Wir können damit zwar nicht direkt Einfluss auf das fremde Script nehmen, aber Funktionen aus diesem Script wieder verwenden und erweitern (nur für unser eigenes Objekt!).

1.2 Hinweise zur Art und Weise des "Codens"

Gerade am Anfang neigen viele Programmierer dazu, ihre Quelltexte alles andere als Übersichtlich zu verfassen. Dabei ist es von großer Wichtigkeit, daß gerade der Ersteller selbst seinen eigenen Quelltext lesen kann.

Code: Unübersichtlich

```
include                                                    "MapObject.gs"
                                                                        class
{
                                          int                       meinezahl
public                  void                  Init(Asset                  asset)
{
inherited(asset);
}
                                              public                      string
{
string                   html                     =                  inherited();
                                  if(meinezahl       ==       0)html       =       html
           else            html           =           html           +           "Ciao!";

}
public                              Soup                            GetProperties()
{
Soup                    soup                    =                  inherited();
                                                                meinezahl               =
return                                                                      soup;
}
public                void                SetProperties(Soup                soup)
    {
  inherited(soup);
                                        soup.SetNamedTag("meinzahl",
}
};
```

Alles anzeigen

Code: Übersichtlich

```
include                                                          "MapObject.gs"

class                CMeinObjekt                isclass                MapObject
{
                                                                            int

                                                                 public                voi
    {
        inherited(pAsset);
    }

                                                                                public
    {



        else


    }

    {
                                                          m_iMeineZahl       =       pSoup.

    }

                                                         public              void
    {
        inherited(pSoup);

    }
};
```

Alles anzeigen

Viel mehr Erläuterungen braucht es wohl nicht: Wenn man Einrückungen und neue Zeilen geschickt einsetzt, kann man auch einen komplexen Quelltext sehr leserlich gestalten.

1.2.1 Kommentare

Im Script lassen sich Zeilen definieren, die von Trainz ignoriert werden um Hinweise zur Funktion von Abschnitten o.ä. zu geben. Diese nennt man Kommentare.
Es gibt ein- und mehrzeilige Kommentare, dazu mal ein Beispiel:

Code: Kommentare

```
include                                                                    "MapObject.gs"

/*
    CMeinObjekt
    ===========
    Erbt von MapObject und stellt weitere Funktionen für MeinObjekt zur Verfügung
*/
class                CMeinObjekt                isclass                MapObject
{
             int    m_iMeineZahl    =    0;                //    Eine    Variable    für    eine

    /*
       Init
       ====

    */
                                                                          public                voi
    {
        inherited(pAsset);


    /*
       GetDescriptionHTML
       ==================
                       Erstellt    aus    HTML-Code    das    Konfigurationsfenster    des    Ob
    */
                                                                                    public
    {




        else




    /*
       GetProperties
       =============                                                                        Lädt
    */

    {
                                                                 m_iMeineZahl       =       pSoup.



    /*
       SetProperties
       =============
                                                                   Speichert         Daten         f
    */
                                                                public               void
    {
        inherited(pSoup);

}; // CMeinObjekt
```

Alles anzeigen

Wie man sieht ist es nicht sonderlich schwierig. Wir entdecken, wie angekündigt, zwei Arten von Kommentaren.
Bei einem mehrzeiligen Kommentar wird alles, was zwischen "/*" und "*/" steht einfach ignoriert. Dieser wird oftmals auch dafür verwendet, um eventuell fehlerhaften Code auszukommentieren, um sich ihm später wieder zu widmen.
Bei einem einzeiligen Kommentar wird alles, was in der gleichen Zeile hinter "//" steht einfach auskommentiert. Dieser kann auch dann folgen, wenn in der gleichen Zeile vorher gültiger Code steht. Erst ab dem "//" wird Trainz den Rest
der Zeile ignorieren.

1.3 Die Bestandteile der Scriptsprache

TrainzScript bringt viele Aspekte aus den bekannten Programmiersprachen C++ und Java mit. Wer jetzt den Schritt wagen möchte zu resignieren, weil er entweder noch nie etwas davon gehört hat oder jemans eine Code-Zeile in diesen Sprachen geschrieben hat, den kann ich beruhigen: Auch ohne Vorkenntnisse kann man in Trainz Scripts schreiben. Meiner Meinung ist es hier sogar einfacher einen Einstieg in die Programmierung zu finden. Vorsicht sollte man dabei allerdings doch walten lassen: TrainzScript ist sehr vereinfacht gegenüber C++ oder Java. Viel tiefgründige Thematik in den Computer und die Programmierung werden hier nicht benötigt, was zwar den Einstieg erleichert, nicht aber später die Weiterentwicklung raus aus Trainz.

1.3.1 Klassen und Objekte

Leider gehört dieses Thema, das in Lehrbüchern zu anderen Programmiersprachen meist erst ab der Mitte oder sogar weit hinten thematisiert wird, hier an den Anfang.
Ohne Klassen geht in Trainz nämlich gar nichts. Machen wir es also kurz und schmerzlos. Wobei das reine Auslegungssache ist...

Eine Klasse ist nichts anderes als die Definition für eine Art von Objekt. Damit dies hier nicht allzutrocken bleibt fange ich einmal mit einem Beispiel an, das in Trainz schon existiert.
Jedes Objekt in Trainz muss einer bereits vorhandenen Kategorie zugeordnet werden. Am Anfang beschränkt sich das meist auf Objekte, die statisch herumstehen:
-> Szenerie (config.txt: kind "scenery")
-> Gleisrandobjekte (config.txt: kind "scenery" mit "trackside"-Eintrag)

Darüber hinaus gibt es natürlich noch etliche Kategorien und auch nicht jede Kategorie in Trainz, wie z.B. "Splines", können nicht durch Script-Funktionen erweitert werden.
Bleiben wir aber mal bei den beiden genannten:

Für jede dieser Kategorien stellt Trainz eine eigene Klasse zur Vefügung:
-> "MapObject" für [freie] Szenerie-Objekte
-> "Trackside" für [gleisgebundene] Szenerie-Objekte

Diese Klassen enthalten für die jeweiligen Objekte genau die richtigen Funktionen. In Trainz sind diese zwar fest im Programm eingebaut, sodaß wir die grundlegenden Funktionen derer nicht überschreiben können, aber dieses Beispiel gibt uns einen guten Einblick in die Funktion von Klassen.
Wir wissen also jetzt, daß man Klassen dazu verwendet ein Objekt zu definieren. Natürlich wissen wir jetzt noch nicht, was das für uns, den Scripter, genau bedeutet.

Ganz einfach: Alles, was wir im Script schreiben, bis auf die Include-Anweisung (siehe: *1.1.1 Einbindung externer Scripts*), schreiben wir immer innerhalb einer Klasse. TrainzScript ist somit eine [nur] objekt orientierte Sprache.
Logisch ist das ganze, weil wir mit unseren Scripts auf die bereits in Trainz vorhandenen Funktionen aufbauen **müssen**. Es stehen uns keine Funktionen zur Verfügung, um ganz neue Arten von Objekten in Trainz zu schaffen.

1.3.1.1 Vererbung

Nein, wir beschäftigen uns jetzt nicht mit Lamarck und Darwin. Wir haben bereits geklärt, dass wir mit unseren eigenen Scripts auf die Funktionen von durch Trainz bereitgestellte Klassen aufbauen müssen. Das geht aber natürlich nur,
wenn wir Funktionen aus anderen Klassen für unsere eigenen übernehmen können. Stichwort: "Vererbung"!
Denn genau das ist möglich: Wir können sagen, dass unsere Klassen von einer oder mehreren Klassen Funktionen erben können.

Grundsätzlich müssen wir immer mind. einmal von einer anderen Klasse erben:
Wenn wir das Script für unser Objekt schreiben, so müssen wir immer von der zum Objekt passenden Trainz-Klasse erben.
Das sähe so aus:

Code: Vererbung

```
include                                                    "MapObject.gs"

class              CMeinSzenerieObjekt              isclass              MapObject
{
    [...]
};
```

Wir müssen bekanntlich für unser eigenes Objekt immer eine neue Klasse anlegen, die wir dann in der config.txt (siehe: *1.1 Integtration von Scripts in einem Asset*) referenzieren.

Das geschieht, indem wir, wie in Zeile 1, einmal das Script laden, das unsere Spiel-Klasse enthält von der wir erben. Und einmal durch das Wörtchen "isclass" in Zeile 3.
Zeile 3 sagt aus:
-> "class": Hier kommt eine neue Klasse
-> "CMeinObjekt": Mit dem Namen "CMeinObjekt"
-> "isclass": die erbt von
-> "MapObject": Szenerie-Objekt
Also: Neue Klasse, mit dem Namen "CMeinObjekt", die von der Klasse "MapObject" (also "Szenerie-Objekt") erbt.

Damit haben wir "MapObject" zu unserer Elternklasse gemacht und können nun alle Funktionen nutzen, die in ihr, oder in einer ihrer Elternklassen definiert worden sind.
Vererbung geschieht also nicht nur zur Übergeordneten Generation, sondern zieht sich, wie in echt, bis zum Ursprung fort.
[Der Ursprung von "MapObject" ist "GSObject": *Wer sich darin prüfen möchte, ob er das Prinzip der Vererbung vertanden hat, kann ja mal schauen, ob er den Weg von "MapObject" zu "GSObject" selbst findet.*
]

Das war es aber erstmal mit Klassen, später werden wir dieses Thema noch etwas genauer beleuchten. Vererbung hört nämlich bei "isclass" und "include" nicht auf.

Wenn du das ganze hier noch nicht verstanden hast, so mache dir darüber keine Gedanken. Später, wenn du etwas mehr Erfahrung hast, so wirst du dies irgendwann einmal verstehen. Es wird Klick machen!
[*Bei mir kam der Klick als ich einmal Abends meine Freundin abholen wollte und ich grübelte, wie ich ein Problem lösen kann. Aufeinmal machte es eben "klick".*]

Im nächsten Artikel geht es dann weiter mit den Datentypen, Variablen und Co.