UniCryptJS

Marcel Portillo

Biel, Spring 2016

Index

1 Einleitung

1.1 Aufgabenstellung

UniVote ist ein bestehendes Online

2 JavaScript

2.1 Objekte erstellen und Closures

Funktionen in Javascript können instantiiert werden. Sie sind die objektorientierte Entsprechung zu Klassen in Java. Werden Funktionen mit dem new Schlüsselwort aufgerufen liefern sie eine Objektreferenz. Allerdings verhält sich das Scope der Variablen anders, als es von Java her erwartet werden würde. Es gibt keine Sichtbarkeitsmodifizierer und auch kein blockscope. Stattdessen gibt es sogenannte Closures.

2.1.1 Closures

Wird in Javascript eine Funktion innerhalb einer Funktion definiert, so wird eine Closure erstellt.

2.1.2 Objekte erstellen

2.2 Prototypen Vererbung

3 Liste der Konzepte von Unicrypt in Java

3.1 Compiler

3.1.1 Anforderung

Ein Compiler prüft zur Entwicklungszeit bereits auf mögliche Fehler. So lassen sich Probleme zur Laufzeit vermeiden. In der Analysephase werden drei Tasks ausgeführt[2]:

- Lexikalische Analyse
- Syntaktische Analyse
- Semantische Analyse

In Javascript (Interpreter Sprache) werden diese Funktionen erst bei der Ausführung zur Laufzeit aufgerufen. Um gewisse Funktionen aus Java in Javascript umsetzen zu können muss eine Semantikprüfung eingeführt werden für die nicht in Javascript enthaltenen Prinzipien. Beispielsweise könnte dies bedeuten, dass bei dem Versuch eine als abstrakte Klasse deklarierte Funktion beim instantiieren einen Error wirft. Dies muss von Hand implementiert werden, da Javascript das Konzept der abstrakten klasse nicht kennt. Eine weitere Möglichkeit wäre, dass eine abstrakte klasse immer ein leeres Objekt zurück liefert, solange sie nicht vererbt wurde.

3.1.2 Implementierung in Javascript

3.2 Klassen, Scope und Zugriffsmodifizierer

3.2.1 Anforderung

Javascript kennt nur die Prototypen Vererbung (ECMAScript 6 beinhaltet allerdings bereits ein natives Klassenkonzept[6]). Diese entspricht aber auch einem objektorientierten Konzept. Deshalb gibt es in einigen Bereichen grosse Ähnlichkeiten.

Eine Klasse hat Attribute und Methoden. Die Attribute sind die Eigenschaften der Klasse. Sie beinhalten Objekte oder primitive Datentypen. Methoden bestimmen die

Verhaltensweise der Klasse. Es werden Operationen definiert, die Attribute verwenden, zuordnen oder Variablen deklarieren. Wichtig zu beachten ist der Scope.

Mehtoden in Java können überladen werden, je nach Anzahl und Typ der Parameter. Nun ist es allerdings möglich nur eine der Überladungen in einer Unterklasse zu überschreiben.
Ein Attribut darf nur bei der Definition der Klasse deklariert werden und nicht durch Methoden zu einem späteren Zeitpunkt.

3.2.2 Anforderungen Scope

In Java gibt es verschiedene Arten von Sichtbarkeit.
Wird eine Variable im Klassenkontext deklariert so ist sie immer der gesamten Klasse zugänglich (d.H. sie kann in Methoden und in Schleifen abgerufen werden).
Wird eine Variable innerhalb einer Methode deklariert, so ist sie auch nur dort verfügbar. Von ausserhalb der Methode kann nicht darauf zugegriffen werden. Dasselbe gilt für Variablen in einer Schlaufe.

3.2.3 Anforderungen Zugriffsmodifizierer

Attribute und Methoden einer Klasse können von außerhalb der Klasse eingesehen werden. Dies ist abhängig von der Sichtbarkeit des Attributes oder der Methode.
Das Schlüsselwort private bestimmt, dass die Methode oder das Attribut ausschliesslich innerhalb der Klasse verfügbar ist.
protected bedeutet, dass die Methode oder das Attribut innerhalb desselben Paketes verwendet werden kann (oder auch von einer Subklasse).
public erlaubt den Zugriff von überall her.

Das Schlüsselwort static erlaubt einer Methode von außerhalb der Klasse aufgerufen zu werden, ohne dass die Klasse instantiiert werden muss.

Mit final lässt sich eine Variable nur genau einmal initialisieren. Spätere Änderungen sind nicht mehr möglich. Eine Klasse mit final kann nicht ererbt werden. Eine final Methode darf nicht überschrieben werden.

3.2.4 Implementierung in Javascript

3.3 Vererbung

3.3.1 Anforderung

Eine Klasse darf von einer und nur genau einer anderen erben. Dabei werden alle Variablen und alle Methoden der Basisklasse in der Unterklasse verfügbar. Mit dem Schlüsselwort super kann der Konstruktor der Basisklasse aufgerufen werden. Es können damit aber auch Methoden der Basisklasse aufgerufen werden, die von der Unterklasse überschrieben wurden.
Eine Klasse darf nur von einer abstrakten oder von einer normalen Klasse erben. Allerdings darf eine klasse mehrere Interfaces implementieren.
Zu beachten sind bei der Vererbung verschiedene Konzepte. Methoden können überschrieben werden. Dazu wird eine identische Methode deklariert. Diese erhält nun den Vorzug gegenüber derjenigen, der Basisklasse. Die zu überschreibende Methode darf nicht final sein. Nur eine Klasse darf vererbt werden. Wurden die abstrakten Methoden implementiert? Darf die Klasse überhaupt vererbt werden (Basisklasse nicht final)? Finale Klassen sind allerdings ein Konzept, dass bei Unicrypt nicht vorkommt und deshalb nicht wichtig ist. Alle Attribute müssen zugänglich sein und alle Methoden vorhanden oder überschrieben werden.

3.3.2 Implementierung in Javascript

3.4 Abstrakte Klassen

3.4.1 Anforderung

Abstrakte Klassen sind Klassen, die nicht Instantiiert werden können. Sie sind erweiterte Interfaces oder umgekehrt ausgedrückt sind Interfaces rein abstrakte Klassen. Abstrakte Klassen können sowohl Methoden, wie auch reine Methodensignaturen definieren. Damit eine von einer abstrakten Klasse erbende Klasse nicht mehr abstrakt sein muss, müssen alle Methodensignaturen implementiert werden[3].
Eine abstrakte Klasse verhält sich also wie ein Interface mit deklarierten Variablen und implementierten Methoden.

Was darf also nicht vorkommen? Es muss verhindert werden, dass eine abstrakte Klasse instantiiert wird. Sie muss vererbt werden können.

3.4.2 Implementierung in Javascript

3.5 Interfaces

3.5.1 Anforderung

Die Anforderung an ein Interface ist eine Vereinfachung der Anforderung an abstrakte Klassen. Allerdings darf ein Interface ein anderes Interface durch extends erweitern. Es muss also grundsätzlich zwischen Klasse, abstrakte Klasse und Interface unterschieden werden können.

3.5.2 Implementierung in Javascript

3.6 Typsicherheit und Generic

3.6.1 Anforderung

In Javascript ist dynamisch typisiert. Es kann jederzeit überprüft werden welchen Typ eine Variable hat. Diese Prüfung muss allerdings von Hand vorgenommen werden und erfolgt erst zur Laufzeit.
Java erlaubt durch Interfaces eine Referenz zu brauchen, die auf verschiedene Objekttypen zeigen kann. Allerdings wird sichergestellt, dass die aufzurufenden Methoden in beiden Klassen vorkommen. In Javascript kann jedes Objekt in einer Variable enthalten sein. Erst der Methodenaufruf zeigt, ob die Funktion vorhanden ist oder nicht. Dies zerstört die Notwendigkeit von Interfaces. Von Hand lässt sich aber die Implementierung der Methoden prüfen oder auch das Vorhandensein des Interfaces.

Generische Klassen können unterschiedliche Typen aufnehmen. Der Wildcard Parameter kommt dabei Javascript am nächsten. Eine generische Klasse in Java hat mehrere Optionen bei der Spezifizierung der Typen bei der Instantiierung. Bei der Invarianz kann ein spezifischer Datentyp angegeben werden. Bei der Kovarianz ist die Bedingung das jedes Element von einer spezifischen Klasse erbt. Bei der Kontravarianz muss jedes Element eine Superklasse einer spezifizierten Klasse sein. [8]
Bei der Definition einer generischen Klasse kann angegeben werden, welches Interface die generischen Parameter implementieren müssen.

Eine Umsetzung sowohl der Typisierung wie auch der generischen Klassen bedeutet eine Restriktion (Spezialisierung) in Javascript, wohingegen diese Konzepte in Java eine Erweiterung (Generalisierung) darstellen.

3.6.2 Implementierung in Javascript

3.7 Unveränderlichkeit

3.7.1 Anforderung

Unicrypt setzt auf das Konzept der immutable objects. Sobald ein Objekt instantiiert wurde, gibt es keine Änderungen der Attribute mehr. Wird eine abgeändertes Objekt benötigt, so muss ein neues erstellt werden. Um dieses Konzept umsetzen zu können, müssen gewisse Design Richtlinien befolgt werden [10]:

- Keine setter Methoden. Es dürfen keine Objektreferenzen oder Werte verändert werden können
- möglichst alle Attribute als private und final deklarieren.
- Unterklassen sollten keine Methoden überschreiben dürfen. Entweder wird dabei die gesamte Klasse auf final gesetzt oder aber es wird mit Factory Methoden gearbeitet (Ansatz in Unicrypt). Wie bei einem Singleton wird der Konstruktor privat gemacht. Um eine neue Instanz zu erhalten muss eine public Methode getInstance() aufgerufen werden. Dabei müssen alle Parameter übergeben werden, da es keine setter Methoden gibt.
- Beinhaltet ein Attribut eine Referenz zu einem veränderbaren Objekt, so soll die Klasse keine Änderungsmethoden zur Verfügung stellen und nicht die Referenz des mutable object freigeben.

Damit ein solches Konzept umgesetzt werden kann, müssen gewisse Funktionen verfügbar sein. Ein Attribut muss als privat und als final deklariert werden können. Dasselbe gilt für den Konstruktor (mit der Implikation, dass das Objekt **nicht** instantiiert werden kann ausser über getInstance()).

3.7.2 Implementierung in Javascript

4 Op Framework

4.1 Klassen

In Java lassen sich Klassen definieren. JavaScript unterstützt diese Funktionalität bisher noch nicht nativ. In ECMAScript 6 werden neue Keywords eingeführt, die eine Klassen basierte Programmierung unterstützen[6]. Da allerdings diese Spezifikation neu ist und noch von wenigen Browsern unterstützt wird, verwendet das Op Framework diese nicht. Stattdessen werden Klassendefinitionen in Prototypen basierte Vererbung umgewandelt.

```
Op.Class('Klassname', {
    //Deklarationsoptionen
}, {
    //Klassendefinition
});
```

Snippet 1: Klassendefinitionen

4.1.1 Klassenname

Der Klassenname bei der Deklaration ist ein einfacher String. Intern wird die Konstruktorfunktion umbenennt, so dass bei der Prüfung der Typen bei der Parameterübergabe die gesamte Prototypenkette untersucht werden kann.

4.1.2 Deklarationsoptionen

Das Objekt Deklarationsoptionen enthält Informationen zur Vererbung und zu generischen Typen. Die Eigenschaft 'extends' spezifiziert von welcher Klasse geerbt wird. 'implements' definiert ein Array aus Interfaces die implementiert werden müssen. 'generic' stellt ein Array aus Typen dar die zum Instanziierungszeitpunkt festgelegt werden müssen. Deklarationsoptionen sind optional. Sollten keine angegeben werden, kann als zweiter Parameter entweder null oder ein leeres Objekt übergeben werden.

```
demo = {};
demo.BaseClass = Op.Class('BaseClass', {
},
{
    testFunction: function() {
        return 10;
    }
});
var ChildClass = Op.Class('ChildClass',  {
        'extends': demo.BaseClass
}, {
});
var instance = new ChildClass();
instance.testFunction(); // 10
```

Snippet 2: Klassendeklarationen

Wie im Snippet 2 ersichtlich wird mit dem Schlüsselwort 'extends' der voll qualifizierende Name der Mutterklasse angegeben. Das Framework kümmert sich anschliessend um die Vererbung.
Dasselbe gilt auch für die Implementierung von Interfaces Snippet 3.

```
demo.Interface = Op.Interface('Interface', null, {
        funcOne: function() {

        }.paramType(['int']).returnType('string'),
});
demo.InterfaceClass = Op.Class('InterfaceClass', {
        'implements': [demo.Interface]
},{
        funcOne: function() {
                return 10;
        }.paramType(['int']).returnType('string')
});
```

Snippet 3: Implementierung von Interfaces

Die Deklarationsoption 'generic' gibt die generischen Typen der Klasse an Snippet 4. Diese müssen bei der Instantiierung mit dem ersten Parameter typisiert werden. Die Eigenschaft 'generic' muss ein Array darstellen (Objekte sind nicht geordnet in JavaScript, Arrays hingegen schon). Innerhalb dieses Arrays dürfen per Konvention nur Großbuchstaben enthalten sein, die der Kondition str.match(/^[A-Z]$/) entsprechen. Eine Klasse kann also maximal 26 generische Parameter enthalten.

Die Eigenschaft 'extends' kennt eine zweite Form. Falls eine Kindklasse von einer generischen Mutterklasse erbt müssen die generischen Parameter im 'extends' Objekt spezi-

fiziert werden. Dies kann entweder durch direkte Zuweisung eines Klassennamen oder durch Weitergabe der generischen Parameter erfolgen. Anders als in Java muss beachtet werden, dass für die selben Buchstaben immer auch das selbe Objekt verwendet werden muss.
Dahingehend ist Snippet 5 nicht zulässig. In diesem Beispiel wäre trotz der Umkehrung der Reihenfolge der Parameter im 'extends' Objekt T und V in Funktionen aus beiden Klassen gleich. Wurde bei der Instantiierung von GenericChildClass T als string und V als int festgelegt, so gilt dieselbe Typenzuweisung auch für Funktionen, die in GenericClass deklariert wurden und eigentlich durch die Vertauschung eine Typisierung im 'extends' Objekt V als string und T als int erwarten würden.

```
var GenericClass = Op.Class('GenericClass', {
    'generic': [
        'T', 'V'
    ]
},{
});
var GenericChildClass = Op.Class('GenericChildClass', {
    'generic': [
        'T', 'K'
    ],
    'extends': {
        'class' : GenericClass,
        'generic': [
            'T', 'string'
        ]
    }
},{
});
```

Snippet 4: Generische Typen

```
var GenericClass = Op.Class('GenericClass', {
    'generic': [
        'T', 'V'
    ]
},{
});
var GenericChildClass = Op.Class('GenericChildClass', {
    'generic': [
        'T', 'V'
    ],
    'extends': {
        'class' : GenericClass,
        'generic': [
            'V', 'T' // Dürfen nicht vertauscht werden.
        ]
    }
},{
});
```

Snippet 5: Achtung: Nicht zulässige Deklaration; Ergibt nicht gewünschtes Resultat

Die Deklarationsoptionen entsprechen einer vereinfachten Verison der Klassendefinitionszeile in Java Snippet 6.

```
public class GenericChildClass<E, V>
           extends GenericClass<E, V>
           implements InterfaceX<V> {
}
```

Snippet 6: Beispiel einer Klassendeklaration aus Java

4.1.3 Klassendefinition

Die Klassendefinition ist ein Objekt mit allen Methoden und Variablen. Auf dem Objekt werden Eigenschaften definiert, die vom Op Framework in Methoden und Attribute umgewandelt werden. In Java entspricht dies dem eigentlichen Körper einer Klasse.

Die Eigenschaften sollten die in JavaScript übliche Syntax aufweisen Snippet 7. Das Framework bietet keine Handhabung für private Klassenmitglieder. Soll ein Attribut oder eine Methode den Zugriffsmodifizierer private erhalten, so wird der Variablenname mit einem vorgestellten _ (underline Symbol) versehen. Auch wenn das Framework dies nicht verhindert, so sollte doch per Konvention niemals auf einer Instanz etwas verwendet werden, dem ein _ vorgestellt ist. instance._privateAttribute = 4 ist also verboten, während (innerhalb einer Methode derselben Klasse) this._privateAttribute = 4 erlaubt ist.

Im Klassendefinitionsobjekt gibt es einige reservierte Schlüsselwörter. Diese magischen Keywords verhalten sich nicht wie erwartet, sondern weisen ein spezielles Verhalten auf. Niemals ist es möglich im Op.Framework eine Methode init, _init, $$super oder static zu definieren und diese als solche aufzurufen! instance.init() ist nicht zulässig (wirft einen Fehler).

Für den internen Gebrauch im Framework gibt es einige Framework-Variablen und Framework-Funktionen. Diese dürfen niemals verwendet werden. Auch dürfen in Klassendefinitionen keine solche definiert werden. Framwork spezifische Funktionen und Variablen folgen der Konvention _Variablenname_. Diese beginnen mit einem underline Symbol und enden auch mit einem solchen. Sollte eine solche Variable ausserhalb des Frameworks verändert werden, so wird das gesamte Programm vermutlich nicht mehr funktionieren und nur noch Fehler werfen.

```
// Funktioniert, da es der JavaScript Syntax entspricht
// Hat aber rein gar nichts mehr mit Java Syntax zu tun und ist deshalb nicht zu empfehlen
var obj = {};
obj['_x'] = 10;
obj['func'] = function(int1) {
    return int1 + this._x + this._privateFunc();
}.paramType(['int']).returnType('int');
obj._privateFunc = function() {
        return 10;
}.returnType('int');

var Class1 = Op.Class('BaseClass', null,obj);

//So wird eine Klasse richtig definiert
//Die Eigenschaften werden direkt in dem Objekt definiert
var Class2 = Op.Class('BaseClass', null,{
        _x: 10, //Per Konvention eine Private Variable
    func: function(int1) {
        return int1 + this._x + this._privateFunc();
    }.paramType(['int']).returnType('int'),
    _privateFunc: function() { // Per Konvention eine private Funktion
            return 10;
    }.returnType('int')
});

var c1 = new Class1();
var c2 = new Class2();
console.log('C1: ' + c1.func(10));
console.log('C2: ' + c2.func(10));
```

Snippet 7: Schlechte und gute Art eine Klassendefinition vorzunehmen

4.1.4 Konstruktor

Ein magisches Schlüsselwort ist init. kein Attribut sollte init genannt werden, init muss zwingend immer eine Methode sein (JavaScript typeof init === 'function'). init repräsentiert den Konstruktor aus Java in JavaScript. Innerhalb der init Methode darf der Superkonstruktor aufgerufen werden mit this.$$super(). $$super ist ebenfalls Schlüsselwort. Er darf keine Methode mit diesem Namen definiert werden.
Ist keine init Methode definiert wird der leere Standard Konstruktor verwendet. Dies ist im Framework wie folgt hinterlegt: obj.init = function init()
Eine spezielle Form des Konstruktors ist die _init Methode. Dieses underline hat hier eine technische Bedeutung. Ein privater Konstruktor verhindert die Instantiierung der Klasse. Dies bedeutet, dass die Klasse eine statische getInstance() Methode enthalten muss, die eine Instanz zurück liefert. Dies ist eine weitere magische Methode. Wird sie als statisch definiert und nur dann, weist sie ein spezielles Verhalten auf. Wird das Schlüsselwort getInstance() verwendet, so kann die Klasse innerhalb und nur innerhalb dieser Methode instantiiert werden, sollte der Konstruktor privat sein 8. Im Framework wird nach Aufruf dieser die Blockade aufgehoben für die Ausführungsdauer.

Es dürfen nie gleichzeitig eine _init und eine init Methode definiert sein.

```
demo.PrivateConstructor = Op.Class('PrivateConstructor', null, {
        _init: function(constructorParam) {
                this.x = constructorParam;
        }.paramType(['int']),
        static: {
                getInstance: function() {
                        return new demo.PrivateConstructor(20);
                }
        },
        x: 10
});

var privateTester = demo.PrivateConstructor.getInstance();
//var fehlerWerfen = new demo.PrivateConstructor(20); // wirft einen Fehler wenn diese Zeile
console.log(privateTester.x); // Log: 20
```

Snippet 8: Ein privater Konstruktor wird definiert

4.1.5 Statische Attribute und Methoden

Enthält die Klassendefinition eine Eigenschaft static in Form eines Objektes, werden die so deklarierten Methoden und Attribute statisch verfügbar gemacht. Für den Aufruf einer statischen Methode muss der vollständig qualifizierende Name angegeben werden 9. Aus Gründen der Bequemlichkeit kann innerhalb der Methoden der Klasse mit this.static[Methodenname] direkt zugegriffen werden.

```
demo.StaticClass = Op.Class('StaticClass', null, {
        static: {
                z: 0,
                y: 0,
                increment: function() {
                        demo.StaticClass.z += 1;
                }
        },
        setY: function(y) {
                this.static.y = y;
        }
});

var staticClass = new demo.StaticClass();
demo.StaticClass.increment();
console.log('Z: ' + demo.StaticClass.z); // Z: 1
demo.StaticClass.increment();
console.log('Z: ' + demo.StaticClass.z); // Z: 2
staticClass.setY(10);
console.log('Y: ' + demo.StaticClass.y); // Y: 10
```

Snippet 9: Statische Attribute und Methoden

4.1.6 Funktionsdefinition und typen Sicherheit

JavaScript ist eine dynamisch typisierte Programmiersprache. Funktionen können erst zur Laufzeit den Typ ihrer Parameter und ihres Rückgabewertes kennen. Um dieses Konzept vom Framework abbilden zu können wurde der Funktionsprototyp erweitert.

Funktionen werden in der Klassendefinition als Eigenschaft definiert. Jede Funktion kann,

muss aber nicht, die Typen der Parameter und des Rückgabewertes vorgeben. Hierbei kommen die Erweiterungen des Funktionsprototypen reutrnType und paramType zum Tragen Snippet 10. Falls die Parameter oder der Rückgabe Wert auf diese Weise spezifiziert wurden, übernimmt das Framework bei jedem Funktionsaufruf die Überprüfung der Typen. Auch dies kann erst zur Laufzeit stattfinden, wie bei Interpreter Sprachen üblich. Gegenüber der klassischen Verwendung von nicht typisierten Parametern bietet diese Vorgehensweise zwei Vorteile:

- Die Fehlermeldungen sind übersichtlicher
- In vielen Fällen konvertiert JavaScript nativ die Typen dynamisch ohne einen Fehler zu werfen. Daraus resultieren schwer einzuordnende, falsche Werte Snippet 11. Mit dem Framework kann dem Vorgebeugt werden.

```
func: function(param1, param2) {
        return param1 + param2;
}.paramType(['int', 'int']).returnType('int'),
```

Snippet 10: Beispiel einer typisierten Methode

```
func: function(param1, param2) {
        return param1 + param2
}
console.log('Ex1: ' + func(10, 20)); // Log: Ext1: 30
console.log('Ex2: ' + func(10)); // Log: Ext2: NaN
console.log('Ex3: ' + func(10, null)); // Log: Ext3: 10
console.log('Ex4: ' + func(10, '20')); // Log: Ext4: 1020
console.log('Ex5: ' + func(10, {x:20}.x)); // Log: Ext5: 30
```

Snippet 11: Unkontrollierbares Verhalten ohne Typisierung

Die Funktionserweiterung paramType nimmt als Parameter ein Array aus strings entgegen und returnType einen einzelnen string. Die Anzahl an Werten im Array für paramType entscheidet über die Anzahl an möglichen Argumenten für einen späteren Funktionsaufruf. Es können sowohl native Typen geprüft werden (wie Beispielsweise int, string(ist in JavaScript keine Objekt, weshalb es klein geschrieben wird), long, array, boolean), als auch selbst definierte Klassen.
Überprüft werden sowohl Anzahl an Parametern wie auch deren Typ. Bei der Angabe des Typs wird auch die Vererbungshierarchie berücksichtigt. Bei einem paramType(['ParentClass']) wird als Parameter eine Instanz der Klasse ChildClass akzeptiert, solange ChildClass von ParentClass erbt.

4.1.7 Funktionen überladen

Funktionen können überladen werden. JavaScript unterstützt diese Funktionalität nativ nicht. Durch die Typisierung der Parameter durch das Framework kann eine Überladung unterstützt werden. Ein JavaScript Objekt verhält sich wie ein associative array. Es findet eine Zuweisung 'Key' -> 'Value' statt. Es ist also nicht möglich zwei gleiche Schlüssel zu definieren. Da im Op Framework die Methoden als Eingeschaft eines Objektes abgelegt werden, können nicht zwei den gleichen Namen besitzen.

Um trotzdem überladen zu können, müssen die Bezeichnungen mit einer Nummer versehen werden. Diese müssen nicht zwingend in einer aufsteigenden Reihenfolge gewählt werden, entscheidend ist nur, dass die Methodennamen dadurch eindeutig werden (keine

Zahl doppelt gewählt ist).
Der Aufruf verhält sich schliesslich wie erwartet: instance.methodname(param1). Das Framework erkennt anhand der Parameter die gewünschte Methode und führt diese aus 12.

Eine Eigenheit des Frameworks ist, dass die Methoden immer noch über ihren ursprünglichen Namen ansprechbar sind. Diese beiden Schreibweisen führen zum selben Resultat: instance.method(singleIntParam) und instance.method1(singleIntParam). Solange die Definition von class.method1 = function(int1) .paramType(['int']) lautet.

```
var BaseClass = Op.Class('BaseClass', null,{
    func1: function(int1) {
        return int1;
    }.paramType(['int']).returnType('int'),
    func2: function(int1, int2) {
        return int1 + int2;
    }.paramType(['int','int']).returnType('int'),
    func2: function(int1, int2, string1) {
        return string1 + (int1 + int2);
    }.paramType(['int','int', 'string']).returnType('string'),
    func4: function(string1) {
        return string1 + ' : one single argument';
    }.paramType(['string']).returnType('string')
});

var ChildClass = Op.Class('ChildClass',  {
        'extends': BaseClass
}, {

});

var childClass = new ChildClass();
var result = childClass.func(10, 20, 'Result: ');
var result2 = childClass.func('Result: ');
var result3 = childClass.func1(10);
console.log('1: ' + result); // Log: 1: Result: 30
console.log('2: ' + result2); // Log: 2: Result:  : one single argument
console.log('3: ' + result3); // Log: 3: 10
```

Snippet 12: Überladung von Methoden

4.2 Generische Klassen

Das generische Konzept wurde auf Klassenebene realisiert, nicht aber bei generischen Methoden. Letzteres wird grundsätzlich nicht unterstützt, da es auch in UniCrypt nicht vorkommt.

Es werden in den Deklarationsoptionen die generischen Typen genannt und zum Instantiierungszeitpunkt festgelegt. Diese müssen zwingend aus nur einem Großbuchstaben bestehen und in einem Array abgelegt sein. Handelt es sich bei einer Klasse um eine generische Klasse muss bei der Instantiierung als erster Parameter ein Array aus Typen dem Konstruktor übergeben werden, so dass die generische Klasse typisiert werden kann. Der erste Parameter für die init() Methode ist also beim Aufruf mit dem new Operator erst der zweite Parameter.

Die generischen Typen können fortan als Parametertyp angegeben werden und das Framework überprüft deren Korrektheit.

Die Typen einer generischen Klasse können also auf zwei Arten festgelegt werden:

- Beim Aufruf durch den new Operator.
- Durch eine Tochterklasse in den Deklarationsoptionen mit der Eigenschaft 'extends': {'generic': ['typ']}

```
var BaseClass = Op.Class('BaseClass', {
        'generic': ['E', 'V'],
}, {
        x: null,
        init: function(val) {
                this.x = val;
        }.paramType(['E']),
        testFuncOne: function(val) {
                return this.x + val;
        }.paramType(['V']).returnType('E'),
        testFuncTwo: function(val) {
                return val + this.x;
        }.paramType(['V']).returnType('V'),
});

var ChildClass1 = Op.Class('ChildClass1', {
        'extends': {
                'class': BaseClass,
                'generic': ['string', 'int']
        }
}, {
        init: function(val) {
                this.$$super(val);
        }
});

var ChildClass2 = Op.Class('ChildClass2', {
        'generic': ['E', 'V'],
        'extends': {
                'class': BaseClass,
                'generic': ['E', 'V']
        }
}, {
        init: function(val) {
                this.$$super(val);
        }
});

var childClass1 = new ChildClass1('Nr. ');
var childClass2 = new ChildClass2(['int', 'string'],2);
console.log(childClass1.testFuncOne(1)); // Log: Nr. 1
console.log(childClass2.testFuncTwo('Nr. ')); // Log: Nr. 2
```

Snippet 13: Zwei Varianten zur Verwendung generischer Klassen

Das Konzept der generischen Typen ist für UniCrypt entscheidend. Auf einer hohen Abstraktionsebene, versucht die Bibliothek viele kryptographischen Operationen abbilden zu können. Klassen werden dazu generisch gehalten, sodass derselbe Programmcode für verschiedene Zwecke verwendet werden kann. Aus diesem Grund muss dieses Konzept

zwingend vom Framework korrekt angewandt werden, um die Ähnlichkeit zur Java Bibliothek erreichen zu können.

4.3 Abstrakte Klassen

4.3.1 Deklaration

Java kennt abstrakte Klassen. Diese unterscheiden sich durch zwei wesentliche Punkte von gewöhnlichen[4]:

- Eine als abstrakt deklarierte Klasse kann nicht instantiiert werden
- Eine abstrakte Klasse kann keine oder mehrere abstrakte Methoden enthalten

Da eine gewöhnliche und eine abstrakte Klasse in allen übrigen Verhaltensmustern identisch sind, behandelt auch das Op Framwork die beiden mehrheitlich gleich. Anstatt des Aufrufs Op.Class('',{},{}) wird Op.AbstractClass('',{},{}) verwendet.
Da es sich dabei nur um einen Wrapper handelt ergeben sich als Nebeneffekt zwei andere Arten eine Klasse als abstrakt zu definieren. Von diesen wird allerdings stark abgeraten.

In einer gewöhnlichen Klasse die mit Op.Class() definiert wurde, lassen sich ebenfalls abstrakte Methoden deklarieren. Ein vorgestelltes $ Symbol hat den zu erwartenden Effekt. Enthält eine Klasse eine abstrakte Methode wird sie automatisch abstrakt.

Aufgrund der Aufbauart vom Framework akzeptiert Op.Class('Klassenname',,,) einen vierten Parameter Snippet 14.

```
Op.Class('Klassname', {
    //Deklarationsoptionen
}, {
    //Klassendefinition
}, {
    //Optionen
    'abstract' = true;
});
```

Snippet 14: Definition von abstrakten Klassen wie im Framework

Wird bei einer normalen Klassendefinition ein Optionsobjekt mit 'abstract' : true angehängt so wird die Klasse ebenfalls abstrakt. Abstrakte Klassen können nicht instanziiert werden.

4.3.2 Abstrakte Methoden

Eine abstrakte Methode wird mit einem vorgestellten $ definiert. Ähnlich dem underline Symbol _ für private Variablen ist das $ Symbol eine Namenskonvention, die eingehalten werden muss. Das vorgestellte $ hingegen hat eine technische Wirkung. Methoden die auf diese Art benannt wurden, können nicht ausgeführt werden und müssen zwangsläufig von einer Tochterklasse überschrieben werden. Wird eine solche Methode nicht durch die Vererbungshierarchie an einer Stelle implementiert, so wirft das Framework einen Fehler bei der Instantiierung.
Es wird allerdings nicht nach Parametern unterschieden, weshalb eine Überladung von abstrakten Methoden nicht möglich ist.

Während abstrakte Klassen ein entscheidendes Konzept von UniCrypt darstellen, so ist die Verwendung von abstrakten Methoden in UniCryptJS nicht notwendig. Der Vollständigkeit halber und zum sicherstellen, dass der Programmcode komplett übernommen wurde können sie allerdings hilfreich sein.

Da JavaScript nicht zum Entwicklungszeitpunkt überprüft, ob alle aufgerufenen Methoden innerhalb einer Klasse auch tatsächlich vorhanden sind, müssen in dem konkreten Fall von UniCrypt keine abstrakten Methoden deklariert worden sein.

Abstrakte Methoden werden nach dem Schema $Funktionsname -> Funktionsname überschrieben. Eine implementierende Methode hat kein vorgestelltes $ Symbol. Achtung: Attribute sind von dieser Namenskonvention nicht betroffen. Werden diese mit vorgestelltem $ Symbol definiert, so können diese normal verwendet werden. Op.Class('Test', null, $x: 5,) resultiert in einem gültigen instance.$x (ist gleich 5).

4.4 Interfaces

Interfaces weisen die gleiche Syntax wie Klassen auf. Variablendeklarationen werden aber ignoriert. Op.Interface(String Klassenname, Objekt Deklarationsoptionen, Objekt Interfacedefinition); Als Deklarationsoption wird das 'extends' Attribut verwendet. Da ein Interfaces allerdings mehrere Interfaces implementieren kann, wird hier anders als bei Klassen ein Array von Interfaces erwartet.

5 Framework Mängel und Probleme

5.1 TODO: Wichtige aber nicht implementierte Funktionalität

5.1.1 Überladen von statischen Methoden

Statische Methoden können zum momentanen Zeitpunkt nicht überladen werden. Dies ist grundsätzlich kein grosses Problem, da dies in UniCrypt selten vorkommt. Eine statische Methode wird allerdings sehr oft überladen: getInstance().
Aus Bequemlichkeitsgründen ist es möglich eine Klasse mit diversen verschiedenen Eingabeparametern zu instantiieren. Um dies auch in UniCryptJS abbilden zu können, müsste auch das Op Framework eine Überladung sowohl der speziellen getInstance() wie auch der anderen statischen Methoden erlauben.

5.1.2 Überladen der Konstruktorfunktion

In der Regel wird in UniCrypt die getInstance() Methode überladen. Diese übernimmt die Parameterprüfung und ruft sich selbst so lange auf (die anderen Überladungen), bis alle Variablen vorhanden sind. Sobald alle notwendigen Werte mitgegeben oder erzeugt wurden, wird der private Konstruktor aufgerufen. Deshalb haben die meisten Klassen auch nur genau eine solche Methode.

Der Vollständigkeit halber wäre es trotzdem wichtig, dass diese Option besteht. Java unterstützt die Überladung von Konstruktoren und deshalb sollte auch das Op Framework diese Möglichkeit anbieten.

5.1.3 Einbeziehen von Interfaces bei der Typisierung

Das Op Framework unterstützt Interfaces. Da UniCrypt auf abstrakte Klassen setzt, ist deren Einsatz für eine erfolgreich Umsetzung von UniCryptJS nicht notwendig. Ich habe also bisher bewusst auf Interfaces verzichtet.

Das Op Framework besitzt allerdings diese Funktionalität. Trotzdem ist sie zum momentanen Zeitpunkt nur eingeschränkt nutzbar. Klassen können Interfaces implementieren. Wird allerdings bei function(classThatImplementsCorrectInterface).paramType('interface'); ein Interface Namen angegeben, anstelle eines Klassennamens, so kann das Framework die Prüfung nicht durchführen. Der Parameter classThatImplementsCorrectInterface wird zurückgewiesen mit der Bemerkung, dass die Typen nicht übereinstimmen. Die Typisierung respektiert also die Vererbungshierarchie, nicht aber die Implementierungshierarchie.

5.1.4 Kein Try {} catch(err){} im Framework

Im Op Framework wird die überladene Methode mit einem Try-Catch-Statement ausgewählt. Dies kann dazu führen, dass Codeblöcke ausgeführt werden, die nicht ausgeführt werden sollte, ehe ein Fehler auftritt und die nächste Methode ausprobiert wird. Mit wenig Aufwand könnte dieses Verhalten optimiert werden. Es sollte kein Try-Catch-Block im gesamten Op Code vorkommen. Das Framework wirft kontrolliert Fehler. Diese sollten nicht bereits innerhalb abgefangen werden.

5.2 Nicht vorhandene, aber vermutlich unwichtige oder nicht implementierbare Funktionalität

5.2.1 Typisierung von Variablen

Ich habe ehrlich gesagt keine Ahnung, wie eine Typisierung von Variablen elegant umgesetzt werden könnte ohne Hilfe eines Präprozessors. Ein etwas seltsamer Lösungsvorschlag könnte folgendermassen aussehen Snippet 15.

Neue Variablen werden zuerst typisiert. Dies geschieht durch den Aufruf ans Framework var x = Op.Var('typ'). x ist fortan eine Instanz vom Typ 'Variable'. x verfügt über genau 2 Methoden: get() und set(assignment). get() gibt das gekapselte Objekt zurück, wohingegen set(assignment) einen Wert zur Zuweisung erwartet. 'assignment' muss vom Typ 'typ' sein der in Op.Var('typ'); spezifiziert wurde. Das Resultat ist ein unleserlicher und unverständlicher Code.

```
var Calculator = Op.Class('Calculator', null,{
        y: Op.Var('int').set(null),
        x: Op.Var('int').set(null),
        setter: function(x, y) {
                this.x.set(x);
                this.y.set(y);
        },
        divide: function() {
                var result = Op.Var('float').set(this.x.get() / this.y.get());
                return result;
        }.returnType('float');
});

var Divide10 = Op.Class('Divide10', {
}, {
        x: Op.Var('int').set(10),
        result: Op.Var('float'),
        init: function(val) {
                var calc = Op.Var('Calculator');
                calc.set(new Calculator());
                calc.get().setter(this.x.get(),val);
                this.result.set(calc.divide());
        }.paramType(['int']),
        getDivisionResult: function() {
                return this.result.get();
        }
});
```

Snippet 15: Typisierte Variablen Lösungsvorschlag

5.2.2 Typisierung bei generischen Parametern

In Java ist folgendes zugelassen:

```
public abstract class ClassName<E extends InterfaceName<V>, V>
extends ParentClass<E, V> implements otherInterfaceName<V>
```

Beim Setzen des Typs von E muss also überprüft werden, ob der Typ von E das Interface InterfaceName vom Typ V implementiert. Dies könnte im Op Framework wie folgt aussehen. Trotzdem erscheint diese Funktionalität für den Einsatzbereich zu übertrieben. Selbst ohne diese zusätzliche Überprüfung kann die für UniCrypt wichtige Überladung von Methoden mithilfe generischer Parameter vorgenommen werden. Es würde das Framework nur unnötig komplizierter machen, im Vergleich zum entstehenden Nutzen einer zusätzlichen Sicherheit.

Die Gefahr besteht also, dass auf einer Variablen x vom Typ E (in Java: extends Interface i methodX(param)return param;) die Methode x.methodX(value) aufgerufen wird. Da aber x vom Typ E genanntes Interface nicht implementiert wird dies eine JavaScript Fehlermeldung 'methodX is not a function' auslösen. Dies sollte aber mit dem Debugger (oder der Zeilennummer bei Ausführung in Node) gut zu finden sein.

5.2.3 Zugriffsmodifizierer: Private, Protected und Public

Fast an jedem Ort im Java Code lässt sich ein private, protected oder public Schlüsselwort verwenden. Einige Beispiele:

- Klassenattribute
- Methoden (statisch und auf Instanz bezogen)
- Klassendefinitionen
- Innere Klassen

Von Interesse für das Framework sind die Attribute und die Methoden. Per Konvention werden diese mit einem underline Symbol gekennzeichnet. Dieses hat allerdings ausser bei der _init() Methode keine technische Relevanz. Selbstredend könnte diese Funktionalität nachträglich noch eingebaut werden. Es gibt allerdings wichtige Gründe die dagegen sprechen.

JavaScript kennt nur die Closures

Table of Illustrations

Liste von Code Snippts

References

[1] Berner Fachhochschule BFH. Univote. https://www.univote.ch/voting-client/, 06. 2016.

[2] Diverse. Abstrakte klasse. https://de.wikipedia.org/wiki/Compiler, 03. 2016.

[3] Diverse. Abstrakte klasse. https://de.wikipedia.org/wiki/Abstrakte_Klasse, 03. 2016.

[4] diverse. Abstrakte klassen. https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html, 06. 2016.

[5] Diverse. Closures (funktionsabschlüsse). https://developer.mozilla.org/de/docs/Web/JavaScript/Closures, 03. 2016.

[6] Diverse. Defining classes. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes, 03. 2016.

[7] diverse. Ecmascript 6 klassen. https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Klassen, 06. 2016.

[8] Diverse. Generische programmierung in java. https://de.wikipedia.org/wiki/Generische_Programmierung_in_Java, 03. 2016.

[9] Diverse. Javascript closures. http://www.w3schools.com/js/js_function_closures.asp, 03. 2016.

[10] Diverse. A strategy for defining immutable objects. https://docs.oracle.com/javase/tutorial/essential/concurrency/imstrat.html, 03. 2016.