Softwaretechnik

Vorlesung 02: Spezifikation mit Typen

Peter Thiemann

Universität Freiburg, Germany

SS 2008

Inhalt

Exkursion in eine Welt ohne Typen: Skriptsprachen

Skriptsprachen

- kleine Programmiersprachen
  entstanden aus Befehls-Sprachen
- einfache Datenstrukturen
  Hashmap (Objekt), Strings
- einfache Syntax
  geläufig, keine Semikolon, häufig nicht klar definiert, ...
- einfache Typisierung dynamisch, schwach, *duck tying*
- einfache Metaprogrammierung
- einfache Implementierung
  interpretiert, wenige Tools

JavaScript, eine typische Skriptsprache

- Anfänglich entwickelt von Netscape's Brendan Eich
- Standardisiert durch ECMAScript (ECMA-262 Edition 3)
- Anwendungsgebiete:
  - Skripten auf Client Seite (dynamisch HTML, SVG, XUL)
  - Skripten auf Server Seite (Whitebeam, Helma, Cocoon, iPlanet)
  - Skripten von Animationen (diablo, dim3, k3d)
  - u.v.a.

JavaScript, Technisch

- Java-style Syntax
- Objekt-orientierte imperative Sprache
  - kein Klassen, aber Prototypen
  - Objekte sind Hashtabellen
- First-class functions (Funktionen sind Werte der Sprache)
  - ein funktionale Sprache
- schwaches, dynamisches Typesystem

Slogan: Jeder Typ kann in jeden anderen typen konvertiert werden, solange es vorstellbar ist, wie dies zu tun ist.

```
node.onmouseout =
  function (ev) {
    init();
    state++;
    node.className =
      "highlight-"
      + state;
    ev.stopPropagation();
  };
```

Probleme mit JavaScript

Systematisch für anderen Skriptsprachen

- Kein Modulsystem
  - Keine Möglichkeit Name zu verwalten
  - Keine Beschreibung der Schnittstellen (Interfaces)
- Kein statisches Typesystem
- keine Anwendungsspezifischen Datentypen
  primitive Datentypen, Strings, Hashtabellen
- Type Konvertierungen sind teilweise überraschend
  "A scripting language should never throw an exception [the script should just continue]" (Rob Pike, Google)
  (Eine Skriptsprache soll nie Fehler werfen, die Skripte sollen einfach weiterlaufen.)

$\Rightarrow$ eingeschränkt auf sehr kleine Anwendungen

Probleme typisch für JavaScript

- Das am meisten verbreitete Anwendungsgebiet
  - Skripten auf Client Seite
  - AJAX
- Dynamische Veränderung von Webseiten per DOM Interface
  - DOM = document object model
  - W3C Standard Interface für Verarbeitung von XML
  - Hauptsächlich in Browsern eingesetzt

Probleme typisch für JavaScript

- Das am meisten verbreitete Anwendungsgebiet
  - Skripten auf Client Seite
  - AJAX
- Dynamische Veränderung von Webseiten per DOM Interface
  - DOM = document object model
  - W3C Standard Interface für Verarbeitung von XML
  - Hauptsächlich in Browsern eingesetzt
- Nicht kompatible DOM Umsetzungen in den Web Browsern

$\Rightarrow$ Programm Rezepte anstelle von konzeptuellem Vorgehen

Ist es möglich, verlässliche Programme in JavaScript zu schreiben?

- Kampf *z.B.* mit dem fehlenden Modulsystem
  - Ad-hoc Strukturierung großer Programme
  - Namenskonventionen
  - Arbeiten im Team
- Arbeiten mit den DOM Inkompatibilitäten
  - Benutzung von Frameworks (widgets, networking)
  - Framworks sind inkompatible
- unerwartete Resultate

Sehr kurze JavaScript Einführung

Regel 1:

JavaScript ist objekt-orientiert. Ein Objekt ist eine Hashtabelle, die Eigenschaften auf Werte abbildet.

Sehr kurze JavaScript Einführung

Regel 1:

JavaScript ist objekt-orientiert. Ein Objekt ist eine Hashtabelle, die Eigenschaften auf Werte abbildet.

Regel 2:

Jeder Wert hat einen Typ. Für die meisten denkbaren Kombinationen werden Werte von einem Typ in einen anderen konvertiert.

Sehr kurze JavaScript Einführung

Regel 1:

JavaScript ist objekt-orientiert. Ein Objekt ist eine Hashtabelle, die Eigenschaften auf Werte abbildet.

Regel 2:

Jeder Wert hat einen Typ. Für die meisten denkbaren Kombinationen werden Werte von einem Typ in einen anderen konvertiert.

Regel 3:

Als Typen gibt es u.a. `null`, `boolean`, `number`, `string`, `object`, and `function`.

Sehr kurze JavaScript Einführung

Regel 1:

JavaScript ist objekt-orientiert. Ein Objekt ist eine Hashtabelle, die Eigenschaften auf Werte abbildet.

Regel 2:

Jeder Wert hat einen Typ. Für die meisten denkbaren Kombinationen werden Werte von einem Typ in einen anderen konvertiert.

Regel 3:

Als Typen gibt es u.a. `null`, `boolean`, `number`, `string`, `object`, and `function`.

Regel 4:

`undefined` ist ein Wert und ein Typ.

Einige kurze Fragen

Definiere ein Objekt `obj`:

```
js> var obj = { x: 1 }
```

Was sind die Ausgaben von:

- `obj.x`
- `obj.y`
- `print(obj.y)`
- `obj.y.z`

Antworten

```
js> var obj = {x:1}
js> obj.x
1
js> obj.y
js> print(obj.y)
undefined
js> obj.y.z
js: "<stdin>", line 12: uncaught JavaScript exception:
 ConversionError: The undefined value has no properties.
 (<stdin>; line 12)
```

Schwaches, dynamisches Typsystem in JavaScript II

Regel 5:

Ein Objekt ist eine dynamische Abbildung von Strings auf Werte.

```
js> var x = "x"
js> obj[x]
1
js> obj.undefined = "gotcha"
gotcha
js> obj[obj.y]
```

Was für einen Effekt/Ergebnis hat der letzte Ausdruck?

Schwaches, dynamische Typsystem in JavaScript II

Regel 5:

Ein Objekt ist eine dynamische Abbildung von Strings auf Werte.

```
js> var x = "x"
js> obj[x]
1
js> obj.undefined = "gotcha"
gotcha
js> obj[obj.y]
    == obj[undefined]
    == obj["undefined"]
    == obj.undefined
    == "gotcha"
```

Schwaches, dynamische Typsystem in JavaScript III

Betrachte Regel 2:

Jeder Wert hat einen Typ. Für die meisten denkbaren Kombinationen werden Werte von einem Typ in einen anderen konvertiert.

```
js> var a = 17
js> a.x = 42
42
js> a.x
```

Was für einen Effekt/Ergebnis hat der letzte Ausdruck?

Schwaches, dynamische Typsystem in JavaScript III

Wrapper Objekte für Zahlen

```
js> m = new Number (17); n = new Number (4)
js> m+n
21
```

Schwaches, dynamische Typsystem in JavaScript III

Wrapper Objekte für Zahlen

```
js> m = new Number (17); n = new Number (4)
js> m+n
21
```

Wrapper Objekte für Booleans

```
js> flag = new Bool(false);
js> result = flag ? true : false;
```

Ergebnis?

Schwaches, dynamische Typsystem in JavaScript IV

Regel 6:

Funktionen sind *first-class*, aber verhalten sich anders wenn Sie als Methoden oder Konstruktoren verwendet werden.

```
js> function f () { return this.x }
js> f()
x
js> obj.f = f
function f() { return this.x; }
js> obj.f()
1
js> new f()
[object Object]
```

Unterscheidung zwischen Existenz und Undefiniertheit I

```
js> obju = { u : {}.xx }
[object Object]
js> objv = { v : {}.xx }
[object Object]
js> print(obju.u)
undefined
js> print(objv.u)
undefined
```

Unterscheidung zwischen Existenz und Undefiniertheit II

Regel 7:

Das `with`-Statement fügt das Objekt, das es als Argument erhält, auf den aktuellen Stack der Umgebung hinzu.

```
js> u = "defined"
defined
js> with (obju) print(u)
undefined
js> with (objv) print(u)
defined
```

Unterscheidung zwischen Existenz und Undefiniertheit III

Regel 8:

Das `for`-Statement hat einen `in` Operator, der den Durchlauf über alle Indexe ermöglicht.

```
js> for (i in obju) print(i)
u
js> for (i in objv) print(i)
v
js> delete objv.v
true
js> for (i in objv) print(i)
js> delete objv.v
true
```

These

- Gebräuchliche Fehler wie
  - verwende Werte die keine Objekte sind wie Objekte *z.B.* benutze Zahlen als Funktionen
  - aufruf nicht vorhandener Methoden
  - Zugriff auf nicht vorhandene Eigenschaften
  - überraschende Konvertierungen

  können durch ein

  statisches Typsystem

  gefunden werden.
- u.v.m.