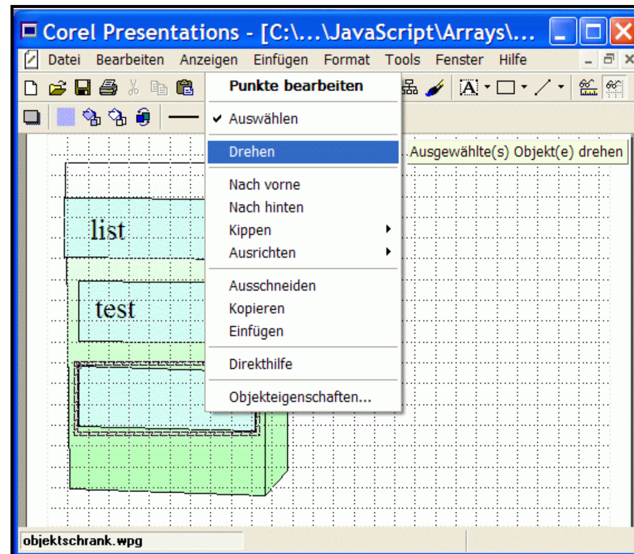

Abb. 1: Methoden bei einem Zeichnen-Objekt

Wir studieren Objekte

Einen Schrank wie in Abb. 1 können wir als ein Objekt ansehen. Er hat verschiedene **Eigenschaften**, z. B. die Größe, die Breite, seinen materiellen Wert oder seine Farbe. Auch die Schubladen lassen sich als Eigenschaften ansehen; sie stellen aber auch ihrerseits wieder Objekte dar, die durch Eigenschaften charakterisiert werden. Diese Verschachtelung hatten wir schon bei HTML-Objekten bemerkt. Der Schrank besitzt daneben auch **Methoden**: Er altert zum Beispiel; dadurch verliert er an Wert. Und er kann angestrichen werden; dadurch verändert er seine Farbe.

> Objekte bestehen also aus Eigenschaften und Methoden. Eigenschaften sind Werte (oder auch Objekte), welche das Objekt beschreiben. Methoden sind objekteigene Funktionen, die die Eigenschaften oder andere Werte manipulieren.

Auch viele andere Gegenstände des täglichen Lebens können wir als Objekte ansehen mit gewissen Eigenschaften und Methoden. Diese Sichtweise erleichtert oft den Umgang mit diesen Gegenständen. So ist es nicht verwunderlich, dass auch Zeichenprogramm diese Objektvorstellung benutzen. In der Abbildung 1 kann das markierte Objekt, die Schublade, mit der Maus verschoben werden; wie das Kontextmenü zeigt, stehen aber auch noch eine große Zahl weiterer Methoden zur Verfügung, mit denen das Objekt in der Lage geändert oder sogar ganz entfernt werden kann. Wir sehen ferner, dass über dieses Menü offensichtlich auch Eigenschaften des Objektes geändert werden können. Dazu gehören u. A. die Füllfarbe und die Randlinie.

Bislang hatten wir schon mit verschiedenen Objekten gearbeitet. Dazu gehörten Formulare und Eingabefelder bei HTML-Dokumenten, aber auch Zeichenketten und Arrays. Bei diesen Objekten handelte es sich um sogenannte **vordefinierte Objekte**; das sind Objekte, von denen Java-Script schon einen Bauplan besitzt. In diesem Kapitel werden wir weitere vordefinierte Objekte kennen lernen. Wir werden aber auch erfahren, wie man Objekte nach eigenen Bauplänen entwickelt.

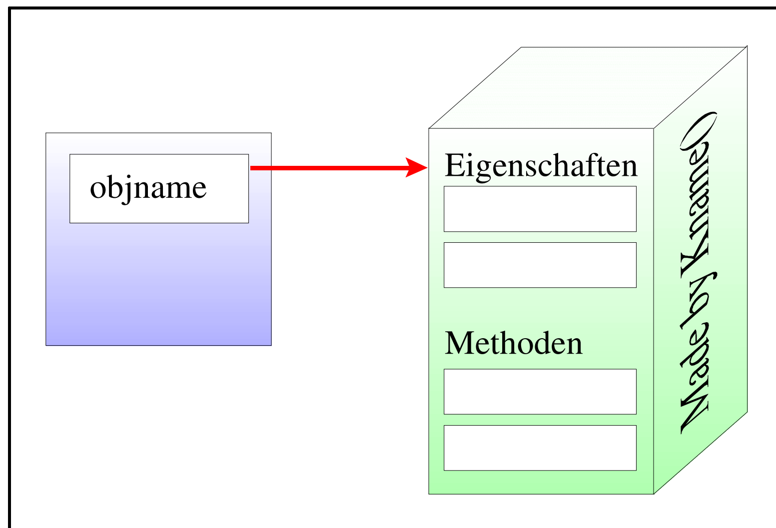

Abb. 2: Die Variable `ojname` zeigt auf eine Instanz von `Kname`.

Der Konstruktor

Vor der Nutzung müssen Objekte erst erzeugt werden. Dies geschieht mithilfe eines Konstruktors. Bei einem **Konstruktor** handelt es sich um eine spezielle Funktion, welche den Bauplan des Objektes beinhaltet. Der Bauplan gibt an, welche Eigenschaften das Objekt besitzen soll und welche Anweisungen durch die einzelnen Methoden ausgeführt werden sollen. Man bezeichnet einen solchen Bauplan auch als **Klasse**. Die Objekte, welche nach einem solchen Bauplan hergestellt werden, nennt man auch **Instanzen** dieser Klasse. Manchmal ist es nicht erforderlich, streng zwischen dem Bauplan und der Realisierung dieses Bauplans, also zwischen Klasse und Instanz zu unterscheiden. In diesem Fall wollen wir von Objekten sprechen.

Es ist üblich, der Klasse und dem Konstruktor denselben Namen zu geben. Bei unserer Array-Klasse heißt der Konstruktor demnach `Array(`Anzahl`)`. Mit der Anweisung

```
var liste = new Array(5)
```

wird eine Instanz dieser Klasse mit 5 Elementen erzeugt.

Allgemein wird durch die Anweisung

```
var objname = new Kname()
```

eine Instanz der Klasse `Kname` erzeugt. Genauer gesagt ist es das Schlüsselwort `new`, welches JavaScript anweist, nach dem im Konstruktor `Kname()` festgehaltenen Bauplan diese Instanz herzustellen. Wie bei den Arrays kann man sich vorstellen, dass bei dieser Anweisung zunächst ein Schrank mit einer Reihe von Schubladen erzeugt wird. Einige dieser Schubladen werden die Eigenschaften aufnehmen. In den anderen liegen die Methoden. (Genauer gesagt findet man in den Methodenschubladen Zeiger, die ihrerseits zeigen, wo die entsprechenden Anweisungen zu den Methoden zu finden sind.) Anschließend wird die Position dieses Schrankes in Form eines Zeigers in der Variablen `objname` abgelegt (Abb. 2).

Auch das String-Objekt hat einen Konstruktor. Ein neuer String (eine neue Instanz) wird durch

```
s = new String()
```

erzeugt. Wir haben diesen Konstruktor bislang nie benutzt (und werden ihn auch nie benutzen), weil JavaScript ihn automatisch als Erstes aufruft, wenn eine Anweisung wie

```
s = "Dies ist eine Zeichenkette."
```

erfolgt.

Von einigen vordefinierten Objekten braucht man gewöhnlich nur eine einzige Instanz. Ein Beispiel ist das Math-Objekt. Als Eigenschaften stellt es eine Reihe von Konstanten wie SQRT2 (Wurzel von 2) oder PI (Kreiszahl $\pi \approx 3{,}14159$) zur Verfügung. Neben den schon bekannten Methoden sqrt(x) für die Wurzelfunktion und der Potenzfunktion pow(x,n) findet man noch zahlreiche andere Funktionen wie die trigonometrischen Funktionen sin(x) und cos(x). Standardmäßig stellt JavaScript schon eine Instanz mit dem Namen Math bereit. Einen Konstruktor gibt es hier nicht. (S. Anhang!)

Das Date-Objekt

Ein weiteres vordefiniertes Objekt ist das Date-Objekt. Als Eigenschaft hat es lediglich die Zeit, genauer gesagt die sogenannte Systemzeit, welche der Rechner von einem eingebauten Uhrenbaustein bezieht. Zahlreiche Methoden werden zur Verfügung gestellt, um aus dieser Systemzeit das aktuelle Jahr, die Nummer des Wochentages oder Stunden und Minuten zu erhalten. Einige von diesen Methoden sind hier aufgelistet:

Methoden	**Bedeutung**
getDate()	liefert den Tag als Rückgabewert
getDay()	liefert die Nummer des Wochentags
getHours()	liefert die Stunden
getMinutes()	liefert die Minuten
getSeconds()	liefert die Sekunden
getMonth()	liefert den Monat
getYear()	liefert das Jahr
getTime()	liefert die Anzahl der Millisekunden seit 1.1.1970 0:00 Uhr.

Wie können wir uns damit die Tageszeit in Stunden und Minuten anzeigen lassen? Zunächst erzeugen wir eine Instanz:

```
var jetzt = new Date();
```

Dann rufen wir die Funktion `zeitAnzeige()` auf:

```
function zeitAnzeige()
{
	var t = jetzt.getHours() + ":" + jetzt.getMinutes();
	alert(t);
}
```

Wir erhalten eine Meldung wie in Abb. 2. Rufen wir die Funktion `zeitAnzeige()` jedoch einige Minuten später auf, erhalten wir wieder dieselbe Zeitanzeige. Offensichtlich wird der Zeitwert nicht laufend aktualisiert, vielmehr wird immer der Zeitwert bei der Erzeugung der Instanz angezeigt.

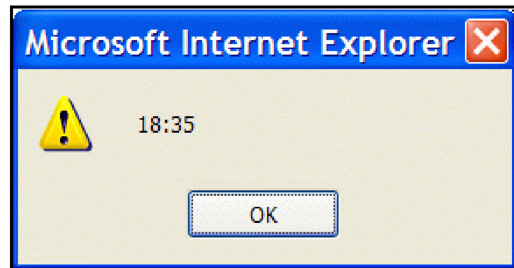

Abb. 2: Uhrzeit

Dies kann man ausnutzen, um Zeitlängen zu messen. Wir erzeugen zwei Instanzen unseres Date-Objekts, eines zur Start- und eines zur Stopp-Zeit; wir nennen sie `start` und `stopp`. Zur Auswertung muss anschließend nur die Differenz dieser Zeiten gebildet werden. Hier bietet es sich an, mit der Methode `getTime()` zu arbeiten:

```
zeitspanne = stopp.getTime() - start.getTime()
```

Es gibt auch Methoden, welche es gestatten, die Angaben eines Datum-Objekts zu verändern. Eine davon ist die `setTime`-Methode. Mit ihr kann man z. B. die Zeitangabe (in Millisekunden) seit dem 1.1.1970 festlegen. So liefern die folgenden Zeilen ein Date-Objekt, welches die aktuelle Zeitangabe besitzt, aber das Datum des folgenden Tages besitzt.

```
var spaeter = new Date(); // Zeit und Datum aktuell
spaeter.setTime(spaeter.getTime() + 24*3600*1000);
```

Aufgaben

1. Schreibe eine Funktion `zeitAnzeige()`, welche in einem Textfeld die Uhrzeit in der Form hh:mm:ss angibt.

2. Warum ist es schwierig, wenn man zur Bestimmung von Zeitspannen die Methoden `getMinutes()` und `getSeconds()` benutzen möchte?

3. Programmiere eine Stopp-Uhr.

Wir entwickeln einen Konstruktor

In diesem Abschnitt wollen wir ein kleines Autorennspiel programmieren. In diesem Rennen sollen verschiedene Fahrzeuge gegeneinander antreten und eine Strecke von 100 km fahren. Jeweils für ein Zeitintervall von 6 Minuten kann der Spieler eine Geschwindigkeit vorgeben. Wenn ein Spieler nun einfach mit möglichst großer Geschwindigkeiten davon eilt, kann es passieren, dass er bald ohne Benzin stehen bleibt. Die Wagen bekommen nämlich zu Beginn des Rennens nur eine bestimmte Benzinmenge zur Verfügung gestellt, z. B. 20 Liter; und der Benzinverbrauch wächst mit steigender Geschwindigkeit überproportional!

Die Rennwagen wollen wir als Objekte darstellen. Natürlich gibt es in JavaScript keinen vorgefertigten Konstruktor dafür; deswegen müssen wir selbst einen solchen entwickeln. Zunächst überlegen wir: Welche Eigenschaften soll unser Rennwagen-Objekt besitzen? Wir beschränken uns hier auf die folgenden:

Name	Bedeutung
s	gefahrene Strecke in km
v	Geschwindigkeit in km/h
name	Name des Wagens
benzin	Benzinvorrat in Liter

Nun gilt es, sich Gedanken zu den Methoden zu machen. Wir wollen hier mit zwei Methoden auskommen:

Name	Bedeutung
einSchritt	berechnet, wie sich die Eigenschaften s, v und benzin in einem Zeitintervall von 6 min verändern
status	liefert als Rückgabewert eine Zeichenkette mit einem Statusbericht über s, v und benzin

Unser Konstruktor sieht so aus:

```
function Auto(startpos, benzin, startgeschwindigkeit)
{
  this.s = startpos;
  this.benzin = benzin;
  this.v = startgeschwindigkeit;
  this.name = prompt("Gib den Namen Deines Autos ein:", "");
  this.step = einSchritt;
  this.status = status;
}
```

Konstruktoren sind aufgebaut wie gewöhnliche Funktionen. In unserem Fall besitzt der Konstruktor 3 Parameter: Die Position beim Start, die Benzinmenge beim Start und die Geschwindigkeit beim Start. Was geschieht nun durch die einzelnen Anweisungen im Funktionsrumpf?

Die erste Anweisung lautet

```
this.s = startpos;
```

Hierduch wird der Eigenschaft `s` des Objekts der Parameterwert `startpos` übergeben. Auffällig ist hier das Schlüsselwort `this`. Dies steht stellvertretend für den Namen der Instanz, die in unserem allgemeinen Bauplan nicht benutzt werden kann; schließlich werden wir mit diesem Bauplan i. A. verschiedene Instanzen mit verschiedenen Namen erzeugen.

Auf die gleiche Weise werden die Parameterwerte für Benzinmenge und Startgeschwindigkeit an die entsprechenden Eigenschaften unseres Auto-Objekts übergeben. Durch die Anweisung

```
var meinAuto = new Auto(0, 20, 0);
```

wird also eine Instanz unserer Auto-Klasse erzeugt, bei der die Eigenschaften `s`, `benzin` und `v` die Werte 0, 20 und 0 zugewiesen bekommen. Damit ist unsere Instanz aber noch nicht fertig; durch die Anweisung

```
this.name = prompt("Gib den Namen Deines Autos ein:", "");
```

wird zunächst nach einem Namen für das Auto gefragt und die Eingabe unter der Eigenschaft `name` gespeichert.

Eigenschaften lassen sich also schon bei der Erzeugung einer Instanz festlegen; sie können dabei auf zwei verschiedene Art eingegeben werden: Entweder legt man sie während der Programmierung als Parameter des Konstruktors fest, oder der Konstruktor bietet die Möglichkeit, sie während des Programmablaufs über Eingabefenster einzugeben.

Nach den Eigenschaften müssen jetzt noch die Methoden in unserem Bauplan erfasst werden. Dabei werden die Anweisungen, welche von einer solchen Methode ausgeführt werden, in eine Funktion ausgelagert. Im Fall der Methode `step` haben wir diese Funktion `einSchritt` genannt. Im Konstruktor wird jetzt nur vermerkt, dass diese Funktion `einSchritt` ausgeführt werden soll, wenn die Methode `step` aufgerufen wird:

```
this.step = einSchritt;
```

Man kann die Namen für die Funktionen und Methoden gleich wählen, so wie wir es im Fall der Methode `status` auch gemacht haben. Beachte, dass bei dieser Zuweisung nicht die üblichen Funktionsklammern stehen dürfen.

Nun müssen wir nur noch die beiden Funktionen `einSchritt` und `status` programmieren. Wie schon beim Konstrukor selbst benutzen wir wieder das Schlüsselwort `this`, um auf die Objekteigenschaften zuzugreifen.

```
function einSchritt()
{
  var text =  "neue Geschwindigkeit von ";
  this.v = prompt(text + this.name, this.v);
  var benzinverbrauch = Math.pow(this.v,2)*0.00015;
  this.benzin = this.benzin - benzinverbrauch;
  if (this.benzin < 0)
  {
    this.benzin = 0;
    this.v = 0;
    alert(this.name + " hat kein Benzin mehr")
  }
  this.s = this.s + this.v * 0.1;
}
```

Die Methode `einSchritt` fragt zunächst einen Geschwindigkeitswert für das nächste Zeitintervall ab. Der Benzinverbrauch für das nächste Zeitintervall wird daraus über die Formel

$$b = v^2 \cdot 0{,}00015$$

berechnet. Durch das Quadrat bei der Geschwindigkeit steigt der Benzinverbrauch überproportional; wenn die Geschwindigkeit verdoppelt wird, vervierfacht sich der Benzinverbrauch.

Dieser Benzinverbrauch wird nun von der aktuellen Benzinmenge subtrahiert. Sollte dann die so berechnete neue Benzinmenge unter 0 sein, werden Benzinmenge und Geschwindigkeit auf 0 gesetzt und es wird eine entsprechende Warnung ausgegeben.

Zuletzt wird die neue Position berechnet, dazu wird die in diesem Zeitintervall zurückgelegte Strecke über die Formel Geschwindigkeit mal Zeit berechnet und zu der letzten Position addiert. Bedenke dabei, dass das Zeitintervall hier immer 6 min = 0,1 h ist.

Die Status-Funktion ist einfacher; deswegen wird sie hier ohne weitere Kommentare angegeben.

```
function status()
{
     var a = "Status von "+ this.name + ": \n";
     a = a + "v= " + this.v + " // " ;
     a = a + "s= " + this.s + " // "
     a = a + "Bezinmenge = " + this.benzin + "\n";
     return a;
}
```

Nun gilt es den Konstruktor auszutesten. Wir entwerfen ein HTML-Dokument mit zwei Schaltflächen und einem Textbereich. Mit der ersten Schaltfläche rufen wir die Funktion erzeugen auf:

```
function erzeugen()
{
     meinAuto = new Auto(0, 20, 0);
     rform.ergebnisse.value = meinAuto.status();
}
```

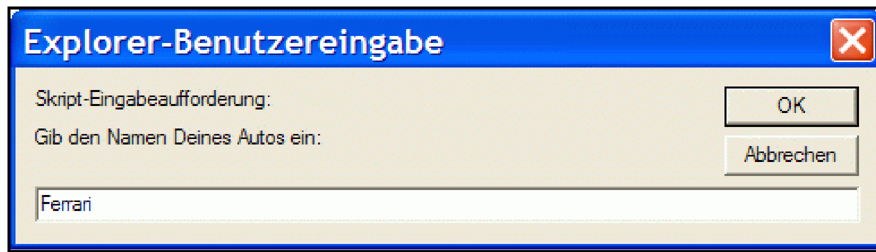

Abb. 4: Hier verlangt der Konstruktor nach einer Benutzereingabe.

Nach dem Anklicken der Erzeugen-Schaltfläche erscheint zuerst eine Eingabeaufforderung wie in Abb. 4, dann wird der erste Statusbericht geliefert (Abb. 5). Beachte, dass vor der Variablen `meinAuto` das Schlüsselwort `var` fehlt; die Variable `meinAuto` wird außerhalb der Funktion als globalen Variable deklariert und kann somit auch von anderen Funktionen aus angesprochen werden.

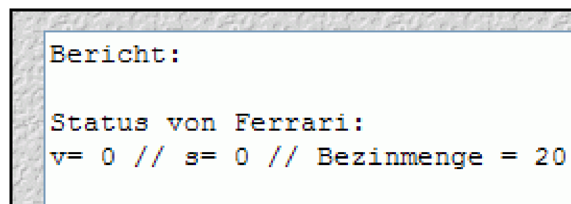

Abb. 5: Statusbericht nach dem Erzeugen des Objekts

Die zweite Schaltfläche soll die Methode `einSchritt` aufrufen und anschließend wieder einen Statusbericht liefern:

```
function naechsterZyklus()
{
      meinAuto.step();
      rform.ergebnisse.value = meinAuto.status();
}
```

Nach dem Betätigen der zweiten Schaltfläche wirst du aufgefordert, die Geschwindigkeit für das nächste Zeitintervall einzugeben. Wenn du z. B. den Wert 180 eingibst, erhältst du einen Statusbericht wie in Abb. 6. Man sieht sofort, dass man bei dem Tempo die 100 km - Strecke nicht bis zum Ende durchhalten kann, der Benzinverbrauch ist zu hoch. Bei den nächsten Schritten sollte man also die Geschwindigkeit etwas kleiner wählen.

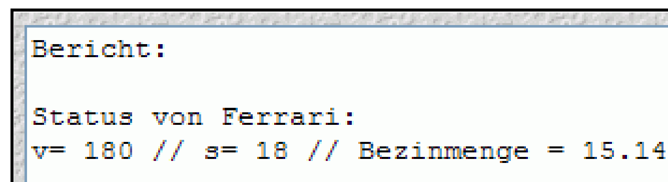

Abb. 6: Statusbericht nach dem ersten Zeitabschnitt

An dieser Stelle sehen wir schon, wie praktisch der Umgang mit Objekten sein kann – wenn der Konstruktor erst einmal vorliegt. Um das Auto-Objekt zu erzeugen oder um es fahren zu lassen, brauchen wir nur jeweils zwei Anweisungen zu geben und erhalten obendrein noch unseren Statusbericht.

Noch deutlicher wird der Vorteil der objektorientierten Programmierung, wenn wir ein Rennen mit einer größeren Anzahl von Autos fahren wollen. In diesem Fall sehen wir die einzelnen Autos als Elemente eines Arrays mit dem Namen `autos` an, welches wir als globale Variable deklarieren. Die gewünschte Anzahl der Autos muss natürlich vor dem Start in einem Textfeld eingegeben werden.

```
var autos;

function anDenStart()
{
     var bericht = "Bericht: \n \n";
     var anzahl = rform.anzahl.value;
     autos = new Array(anzahl);
     for (var k=0; k<anzahl; k++)
     {
          autos[k] = new Auto(0, 20, 0);
          bericht = bericht + autos[k].status() + "\n";
     }
     rform.ergebnisse.value=bericht;
}


function naechsterZyklus()
{
     var bericht =  "Bericht: \n \n";
     for (var k=0; k<autos.length; k++)
     {
          autos[k].step();
          bericht = bericht + autos[k].status() + "\n";
     }
     rform.ergebnisse.value = bericht;
}
```

Aufgaben

1. Erläutere die Funktion `anDenStart()`.

2. Vervollständige das Autorennprogramm und teste es aus. Versuche Rennstrategien zu entwickeln, die zum Sieg führen. Gibt es eine optimale Geschwindigkeit? An welchen Stellen lässt das Programm sich noch verbessern?

3. Untersuche den Quelltext des Igel-Programms aus dem Kapitel „Wir lassen zeichnen“. Überlege, wie man mit mehreren Igeln gleichzeitig arbeiten kann. Teste deine Überlegungen aus.